To
my wife Ulla

for helping me through difficult times
and for all the support and encouragement
which she has given me.

SPEEDOS:

Making Computers Secure

volume 2

James Leslie Keedy

Table of Contents

List of Figures

Preface

Clarification: We first clarify what is meant in this book by computer security. When used in the context of computer systems, and in particular computer operating systems, the word "security" can have (at least) three quite different meanings.

It can mean that the operating system code has been proven "correct", in the quasi mathematical sense that a specification exists and that the code of the operating system has been proven to conform to the specification. This is the sense in which the word "secure" is sometimes used, for example, in association with the claim that Sel4 (https://sel4.systems/) is the "world's most highly assured OS kernel". This is not the meaning of "secure" when we describe SPEEDOS as secure.

Similarly the reliance on encryption techniques to guarantee security is not the sense in which the word security is used here, although SPEEDOS actually uses such techniques for transferring information over the Internet and for accessing discs.

In this book and in other documents on SPEEDOS the word security is used in the architectural sense, i.e. with respect to the hardware instruction set design and the operating system design (especially but not exclusively the design of the kernel). As will become evident, the SPEEDOS architecture is radically different from that of conventional systems.

This book records the main results of an Odyssey which has lasted for more than fifty years of my life, beginning with my work in the design team of the VME operating system for the ICL 2900 Series of computers in Kidsgrove, England. This was followed by my founding the MONADS operating system group at Monash University in Melbourne Australia, with follow up work on MONADS in the groups which I later led at the University of Darmstadt in Germany, the University of Newcastle, N.S.W., Australia and the University of Bremen in Germany. My final professional move was to the University of Ulm in Germany, where I founded the SPEEDOS project and the Timor project[1] in the Department of Computer Structures. Since my retirement I have continued to develop the SPEEDOS ideas, considerably extending and improving on the original version and working out how to implement some of the wilder concepts, such as the world-wide unique virtual memory and addressing incorporated into SPEEDOS.

[1] Timor is an object-oriented and component-oriented programming language designed to accompany SPEEDOS, see the Timor website https://www.timor-programming.org/

Whereas my team at Monash actually built several prototypes for the MONADS-PC system which were then used later in Newcastle, Bremen and Ulm, there is no prototype implementation of SPEEDOS, partly due to a lack of funding. Nevertheless I have formulated a plan which I believe will convince computer manufacturers to make a small modification to their RISC computer designs which will both (a) enable SPEEDOS systems to be built and (b) at the same time allow existing RISC applications to execute without modification except a re-compilation. This hardware modification is particularly significant since it allows capability systems (such as SPEEDOS) to be built which not only improve the way that access rights can be formulated and controlled but also can provide a solution for the confinement problem, thus making computers far more secure than conventional systems. This modification is described in detail in [1], which can be downloaded from the SPEEDOS website[2].

It need hardly be said that current systems are riddled with security loopholes and that attempts to close these are usually only partially successful. This is a nuisance for normal users (to say the least), but it is far more serious in some areas, especially national security, where espionage and cyber warfare could at any time lead to a total disaster, and in hospital systems, in electricity supply systems and similar public utilities which are vulnerable to attack. For this reason I would recommend that the first SPEEDOS systems are built with such applications in mind.

The book is in two volumes. The first volume is an introductory walkthrough of most of the fundamental technical ideas that form the basis upon which the SPEEDOS design is built. Some of the ideas are well known and a few are less well known. What makes them interesting is that almost none of the best of them are to be found in the major operating systems in current use. I explain a concept, e.g. virtual memory, which is in use but where several decisions are possible. I explain why one choice is better for security than the others, and yet almost invariably a worse alternative has been chosen for implementation in current systems. And it also turns out, almost without exception, that the good choice for security is the most efficient solution!

For this reason volume 1 can have a dual purpose. It serves first as my explanation why I chose particular ideas to form the basis for SPEEDOS. In this sense it serves as an important introduction to SPEEDOS. But second, it can provide additional material for a first computer science course in computer architecture and operating system design. In fact it is to a considerable extent based on undergraduate courses which I have given in the past.

[2] https://www.speedos-security.org/

The tenor of the second volume is quite different. Relying on the information in the first volume, it provides a technical introduction to the SPEEDOS kernel and an operating system built on the kernel, explaining in some detail how a real SPEEDOS system can be designed and built. The second volume is suitable for graduate courses in the same area, and will certainly give good students ideas for writing their own PhD theses in this area.

At the outset I would like to make clear that the emphasis in the book is largely on the design of computers and their basic software. There are some areas, in particular those concerned with computer graphics and with the functionality of the Internet, where my expertise is limited to that gained as a user of such systems. Although I have attempted to show the relationship between these fields and SPEEDOS in the second volume, the main emphasis in the book is concerned with the design of the computers themselves and on the basic structure of the operating systems which control them. I believe that this is the best basis on which to improve Internet security.

Volume 1 can be read independently of volume 2, but the reverse is not the case, even for computer scientists and programmers.

In order to simplify cross references between the volumes, the chapters for both volumes are numbered as a single sequence, but each volume uses separate page numbers.

Readers who already have experience in operating systems and in computer architecture will probably be familiar with Parts 1 and 2 in volume 1. I suggest that such readers can skim through these two parts, but Parts 3 and beyond contain much new material which is essential for an understanding of the SPEEDOS ideas. Among the highlights of these chapters I draw special attention to chapter 13, which explains how the confinement problem can be solved.

Finally, I should mention that this work would never have existed except for a piece of advice given to me by the late Professor Chris Wallace, former Head of the Department of Computer Science at Monash University. When I first arrived at Monash I mentioned to him that it would be nice for me to do some research in natural language systems. But he wisely said that it would be sad for me to throw away the experience I had gained at ICL. He was right!

I hope that someday a SPEEDOS system will be built, and I would very much like to lead a project to do so, but that depend whether I will be successful in convincing computer manufacturers to modify the designs of their RISC systems. Meanwhile, I hope that you will enjoy reading both volumes.

Leslie Keedy
BREMEN 2023

Part 5
The Basic Kernel Design

Chapter 17
A Modular Kernel Design

Around the turn of the millennium I decided that it would be worthwhile to clarify how we might design a successor for the Monads systems. In principle Monads had solved one of the key security problems, viz. how to control access rights such that Lampson's matrix [2] could be fully implemented, allowing individual users to determine for each of their files (individually in a finely grained manner) what access rights could be granted to other users. But I was concerned that a further serious security problem, known as the confinement problem, remained unsolved (see volume 1 chapter 3 section 1). I had already found a starting point for such a solution at the programming language level in a paper published in 1997 [3], so the time seemed ripe to initiate a new project under the name SPEEDOS[3] together with my Ph.D. student Klaus Espenlaub. The first task was to determine how to structure a kernel for SPEEDOS.

As Espenlaub points out in his thesis [4][4], security kernels in the past have not been very successful, largely for two reasons. First, they have usually supported the out-of-process model for their clients. Second, they have been implemented monolithically. We now look at these issues in turn.

1 Kernel Support for In-Process Design at the Application Level

We need say little about this first point, because the in-process model has already been described in detail in volume 1 and was shown to be superior to the out-of-process model from various viewpoints, including security and charging[5] [4]. The SPEEDOS kernel and thread scheduler therefore provide a rigorously

[3] SPEEDOS is an acronym for 'Secure Persistent Execution Environment for Distributed Operating Systems'.

[4] Espenlaub's thesis can be downloaded at http://vts.uni-ulm.de/doc.asp?id=5333); it describes our first attempt at designing a SPEEDOS kernel. Many of the ideas reappear, often in a modified form, in this and the following chapters, but some significant ideas have radically changed.

[5] See chapters 8 and 15 in volume 1.

in-process model for the user threads which it supports.

Espenlaub also proposed an early kernel design which itself was in-process. This was partly accepted (see below), but some of Espenlaub's idea proved to be extremely difficult to implement, as was will be explained in chapters 20 to 22. Consequently there remains a central core of kernel activities which might be regarded as out-of-process, though their design by no means follows the standard out-of-process model. The SPEEDOS kernel in fact adopts the basic idea developed for the MONADS systems, as originally developed by John Rosenberg [5], which proved to be very efficient to implement, as will be shown in chapter 22. These are handled in a structured manner via separate kernel processes.

2 Modularity via Co-Modules

In order to avoid the need for a monolithic kernel, a strategy has been adopted whereby some activities usually considered to be kernel activities have been outsourced to individual user-level modules which are trusted by the kernel, thus allowing these functions to be handled in-process.

The SPEEDOS kernel adopts a policy with respect to modularity which differs quite substantially from those found in other kernels. The only static modular structure which the kernel provides for its users is based on information-hiding (as was described in volume 1 chapter 14).

In order to provide greater flexibility, and in order to keep the basic kernel code as small as possible, the core code of SPEEDOS concentrates on executing only the most essential security code. Ancillary functions associated with particular applications, including the organisation of the persistent data structures which are needed to carry out security-related functions, are delegated to *co-modules*. These are placed in the same container as that which holds the information-hiding application module for which they are needed (see Figure 17.1). The organisation of co-modules is described in detail in chapter 19.

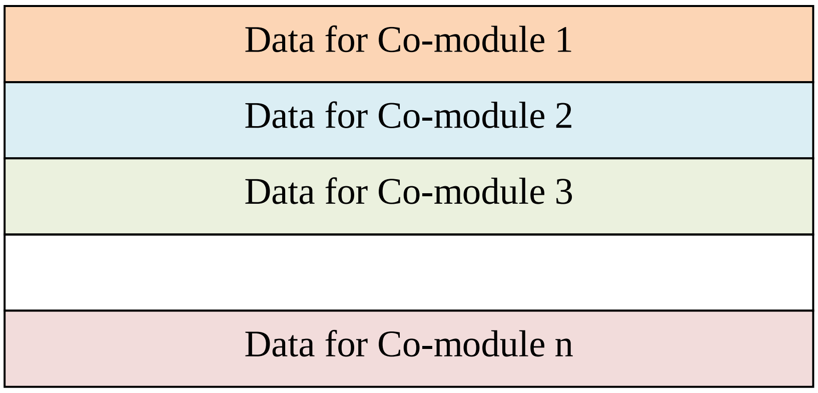


Figure 17.1: Data of the Co-modules in a Container

This organisation has a number of advantages. First it permits the kernel's most essential functions which are executed with special privileges (e.g. the right to load segment registers) to be implemented in the core kernel. These can be separated from other security activities which do not (and should not) need such privileges. This contrasts with many operating system designs, in which far too much code is executed in privileged mode, thus increasing the risk of misuse and therefore potentially opening up security loopholes.

Second, it supports the idea, first formulated as a design principle for the Hydra capability system [6], of separating mechanism and policy. The core code of the SPEEDOS kernel provides the security *mechanisms* (which remain fixed), while its co-modules support *policy decisions* reflected in the persistent data structures which the core kernel uses. These can be varied by those users responsible for security policy.

Third, it means that large sections of security relevant code (which do not have special kernel privileges) can be protected using the basic SPEEDOS security mechanisms (e.g. module capabilities, inter-module calls). To achieve this, kernel co-modules (but not the core kernel) are subject to the rules of normal modules, which were outlined in volume 1 Chapter 14.

Fourth, as a result of the kernel's use of persistent information stored in co-modules, the core kernel code does not need to create persistent data structures. Any information which it generates while carrying out its duties is temporary. In the case of a kernel failure a restart can therefore be achieved much more easily.

Fifth, although the core kernel code cannot be qualified by bracket routines[6], its co-modules can. This means that further measures can be taken to protect them, as will be discussed in chapter 24.

Finally, this design technique avoids the need for a large monolithic kernel.

3 Kernel Use of Co-Modules

Espenlaub proposed that the kernel should access the information held in its co-modules in two different ways. His preferred way (because it conforms better to the information hiding principle) is to invoke predefined routines of its security-related co-modules in order to take advantage of their functionality. For this purpose it does not store capabilities in its own (non-persistent) memory, but obtains the information needed to make such calls from information which is directly accessible when needed. He referred to such calls as *forced method calls* [4, p. 156]; their return values are inspected and acted upon by the kernel. However, he did not describe how such calls could be implemented, and it has become clear that such forced calls are difficult and inefficient to implement. The

[6] see Volume 1 Chapter 13.

author therefore reached the conclusion that, despite their aesthetic appeal, a much simpler and more flexible mechanism should be provided to handle the kernel's need to execute modules at the user level. I have called this mechanism "surrogate threads", which are described in more detail in chapter 22.

There are some circumstances in which it is desirable, or even necessary, for the kernel directly to access the pre-defined persistent data structures of its co-modules. This allows the co-modules which set up the data structures to base the information in these data structures on policy decisions made outside the core kernel code, though the data structures themselves have a predefined format.

It can be argued that direct access to data is a violation of the information hiding principle. However, it is important in this context to remember that this is the mechanism via which the information hiding principle is established for all other modules in the system. The kernel's core code is not a module.

4 Composite Modules

In accordance with the description in Volume 1 Chapter 14, a module is implemented using two containers, a data container (for its persistent data) and a code container (for its code). This concept is now extended to encompass the idea of *composite* modules, i.e. units which consist of a number of co-modules stored in a single container. In fact all modules in SPEEDOS are composite modules. Although the co-modules often provide ancillary security functions to assist the kernel, they usually perform tasks which are closely associated with a particular application module. Their persistent data is stored in the same container as that used for the data of the application module (which is itself considered to be a co-module). In other words, what we have so far called a module is in fact a cluster of related information hiding modules with their persistent data stored in a single container, which is owned by the user who created the application module. However the code of these co-modules can be distributed over different code containers, which provide the algorithms relevant to the purpose of the specific co-module. An important reason for this arrangement is to allow co-modules to provide functionality (often security sensitive functionality) associated with specific application modules.

Since individual kernel co-modules have all the properties previously described in connection with normal modules they are accessed via their own separate module capabilities. In volume 1 chapter 14 the impression was given that a container holds only one module and that the unique identifier in a module capability is the unique module container number. It now becomes necessary to distinguish between the persistent data of different co-modules within a single container. Hence the unique identifier in a module capability is in fact defined as a unique container number *and a small index number*. The kernel uses this to select the required co-module (see Figure 17.2).

The indices of those co-modules which the kernel needs to use are fixed, thus enabling the latter to access them without having to possess a capability for them. Additional user modules have higher indices which are determined dynamically.

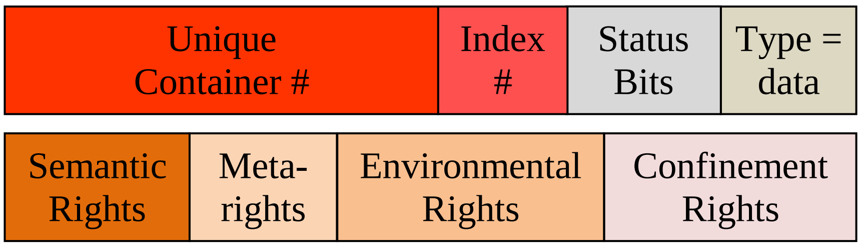


Figure 17.2: The Basic Structure of a File Capability

A subfield in the status bits, hereafter called the *type field*, is set to "data" in capabilities which provide access to co-modules. The rights fields are described in more detail in Chapter 26.

5 Functions Delegated to Co-Modules

In this section a few (but not necessarily all) examples of security-sensitive co-modules are briefly described.

5.1 Co-Module Management

The creation and management of co-modules in a container is clearly a sensitive activity. As we shall see shortly, different policies can be associated with different containers. The management activity is itself carried out in a co-module, the *Co-Module Manager*[7]. This maintains a central table of co-modules (including the main application module) for the container in which it resides, known as the Co-Module Table (CMT). For each co-module this has an entry which contains information such as a pointer to the persistent state data of the module, a capability for the code module which is bound to the co-module and a module capability for the list of qualifiers which bracket the co-module, as well as some status information. This information enables the kernel to execute inter-module calls to co-modules.

5.2 Segment Management

The creation and further management of individual segments in a container is a very sensitive activity, because it is necessary, for example to guarantee that different segments do not overlap in the container memory space. In each container

[7] The names of specific individual modules are capitalised for clarity.

there is a *Segment Manager*. Segment Managers associated with different containers can, at least in part, be programmed differently. They create and delete segments in the container for its other co-modules and may be responsible for garbage collection in the container.

5.3 Virtual Page Table Management

The mapping from virtual page numbers to disc addresses for the pages of a container is held in a co-module of the same container, and the code for this determines how the table is organised. The core kernel accesses data prepared by the *Virtual Page Table Manager* co-module to obtain the information which it needs, for example, for managing the TLB. Different containers can have different page table structures and paging strategies.

5.4 Qualifier List Management

The SPEEDOS kernel supports the use of qualifiers with bracket routines (cf. volume 1 chapter 13), but delegates the management of the qualifier lists, etc. to a co-module associated with the qualified co-module, the *Qualifier List Manager*. Each co-module in a composite module can be separately qualified and can have multiple qualifiers associated with it.

5.5 Debugging Modules

Debugging is the activity of finding and correcting errors in programs. For this purpose a *Debugger* co-module needs access to the data structures which the program has created and used in an attempt to establish the cause of an error. For this purpose it needs to have *white-box* access to a module.

The information hiding principle is sometimes described as a *black-box* model, because the clients of such a module are unable to see how the module works. This is normally one of the strengths of the information hiding principle, but there are some tasks, especially system tasks such as debugging, which require special code to access the internal structure of a module. Allowing this to happen is called *white-box* functionality. We shall see shortly how this can be organised for some co-modules.

6 The Kernel User Interface

The interface presented to the users by the core kernel is relatively simple, and is best regarded as an extension of the hardware instruction set. No special privilege is required to call some kernel instructions, but the kernel carries out checks in order to establish whether the user request can be validly carried out.

There are some cases (e.g. the Segment Manager) where the kernel needs to be sure that certain of its instructions are being called only by its own security-sensitive co-modules, and for this purpose special *kernel capabilities* (with type field set to "kernel" and access rights showing which kernel instructions are permitted by the holder) must be presented as operands. For such instructions any module can attempt to execute the instruction, but if an appropriate kernel capability is not provided as an operand the instruction generates an interrupt. Examples of this will become evident in later chapters.

Since the operands of kernel instructions must be separated from the parameters for inter-module calls, the user sets up a segment containing the operands for a kernel instruction addressable via segment register 15 before executing the instruction. The kernel then uses segment register 15 while carrying out the instruction. The segment itself is a normal segment which can hold capabilities (e.g. kernel capabilities, pointers and data) in which the kernel can also return results on completion of the instruction.

7 Kernel Modules and Processes[8]

Not all modules associated with the kernel can be directly identified with particular application modules, e.g. device drivers, spooler modules, resource allocation modules, the thread scheduler. These modules are stored in containers which are typically owned by the system manager. Like application modules, they are implemented as composite modules with their own co-modules, and they can be invoked in-process by user level threads.

Similarly some activities not associated with particular applications may need to be executed in separate threads not belonging to a particular application. Kernel processes are discussed in more detail in chapters 20 to 22.

8 Cut, Copy and Paste

Many operating systems provide a very useful *clipboard*[9] to support a general cut, copy and paste facility[10]. When used to transfer information from one module to another, this provides users with a potential way to avoid privacy checks. Consequently the SPEEDOS kernel does not provide such a mechanism. However, individual applications can easily support such a facility for use within the program, and where appropriate a buffer can be organised in a sharable file to allow users who have a capability for the file to use this as a clipboard.

[8] To distinguish activity in the core kernel from user activities, we use the term *process* for each kernel activity (see chapter 20) while the term *thread* is used in the case of user level activity outside the kernel. However the term process is also used in SPEEDOS to signify a collection of threads designed to co-operate with each other, see chapter 20.

[9] see https://en.wikipedia.org/wiki/Clipboard_(computing)

[10] see https://en.wikipedia.org/wiki/Cut,_copy,_and_paste#Origins

9 Conclusion

SPEEDOS has introduced a significant new kernel structuring principle, the use of co-modules in a composite module [4], chapter 5. Notice that this is unrelated to the hierarchical operating system structuring techniques described in volume 1 chapter 9. The flexibility of this technique will become more evident in the following chapters, which discuss various aspects of the kernel design.

It can be argued that as a consequence of this technique, the core kernel does not manage all the security features of a system directly and therefore, strictly speaking, that it cannot be considered as a security kernel. However, the classification as a security kernel or not is not as such important. It is important however that through this structure a minimum amount of code is actually executed in privileged mode, reducing the risks of misuse, and on the other hand that security-sensitive co-modules can support different policies.

Chapter 18 Module Variants and their Invocation

In previous chapters the impression was given that all modules[11] have a uniform structure and are invoked in an identical way. However, there are good reasons for introducing variations, while keeping the information-hiding principle intact. This chapter introduces some additional details regarding the organisation and invocation of modules. In order to do this it is necessary to have a basic understanding of the kinds of data which are associated with a module, both statically and dynamically.

1 Kinds of Data Associated with a Module

1.1 Inter-Module Linkage Data and Parameters

These data items are dynamically created and deleted on a (kernel) thread stack in a process container by the kernel as part of the inter-module call/return mechanism, which was introduced in chapters 14 and 15, as well as similar call mechanisms which are described later in the chapter.

1.2 Persistent Data

The lifetime of persistent data is independent of the threads which use it. It comes into existence as a result of a request to create an information–hiding file, and it ceases to exist when this is explicitly deleted[12] or can be garbage collected. It is held as a heap in a file container.

[11] In principle all modules are co-modules, as was described in the previous chapter. However, where nothing is to be gained from emphasizing the significance of co-modules we frequently revert to the simpler and more conventional word "module".

[12] Prof. Roger Needham [27] argued that in a capability-based system an object should persist until capabilities for it no longer exist, but that view is unrealistic in a system where capabilities can be distributed worldwide.

1.3 Temporary Data

For subroutine and similar calls *within* a module the kernel thread stack is *not* used. This decision, which differs from Espenlaub's design [4, pp. 167-8 and 182] and most in-process systems, was necessary partly because of the problem of creating persistent segments by linking a temporary segment into a persistent structure (which is the way compilers typically create the code for this purpose) and partly because of garbage collection issues. This approach also gives compilers freedom to implement languages with varying scoping strategies and at the same time it simplifies the kernel.

A semantic routine, when activated in a module, calls the Segment Manager co-module in its own container to set up a separate root segment for its temporary data (i.e. the data which the thread creates for this call and which is deleted when the thread exits from the routine). This can be used to build an internal subroutine call stack, a temporary heap and/or for any other purpose for which it might need temporary data. When the semantic routine exits, this temporary data is deleted (either explicitly by the routine calling the Segment Manager or implicitly via a garbage collector).

In the case of a call to a file module the temporary data is stored in the same heap as the file's persistent data. In the case of a call to a program module (i.e. a module without persistent data) a single heap is used by all threads for this purpose. This is permanently associated with the program at the current node.

1.4 Retained Data

This is data which allows a thread to retain information relating to a sequence of calls between an open call and a close call (see below). This data might be useful for example to keep a note of the next record to be read, thus allowing a file to have a (very useful) semantic routine `get_next`, etc. This is stored in the heap associated with the called module, i.e. alongside the file data.

In summary, all forms of data for a file module (persistent data shared by all threads using the file, retained data for those threads which require non-persistent information to be retained between their inter-module calls, and temporary data for all the calls on semantic routines of the file) are held in the container for the file module. The associated code module should contain instructions which synchronise access to the persistent data. Since retained data segments are separately rooted for each thread, and temporary data segments have no saved root (except the segment register via which they are addressed) different threads cannot interfere with each other's retained or temporary data segments, although they are held in the same container.

The data for a program module (temporary data for all the calls on semantic routines of the program) are held in the single container associated with the program module, for all the routines which are (possibly concurrently) active within it. Protection between the segments of different threads is guaranteed in that they cannot load segment registers to address the segments of other threads.

2 Kinds of Module

The above overview of kinds of data supported by SPEEDOS allows us now to consider how these data kinds can be combined to produce various different patterns for SPEEDOS information hiding modules. Figure 18.1 shows and names the various combinations.

	Persistent	Retained	Temporary
File	√	x	√
Openable File	√	√	√
Program	x	x	√

Figure 18.1: Kinds of Data

All these module variants need temporary data in which the code can carry out its calculations.

2.1 File

The semantic routines of a file which does not require retained data can only be invoked using inter-module calls. Each call is completely independent of other calls (except that access to the persistent data may need to be synchronised). This form would be useful, for example, to access routines from a library of mathematical functions which *look up* their results in a table (such as a table of trigonometrical functions of the type which students might use in schools).

2.2 Openable File

This corresponds, for example, to conventional commercial data processing file, where a user may need to make a series of calls in sequence. For example a payroll file may be called multiple times in order for the payroll clerk (either human or another module) to calculate the weekly or monthly pay of employees.

2.3 Program

This serves a function similar to the file, except that it needs no persistent data. For example it may have semantic routines for *calculating* trigonometrical functions.

3 Types and Implementations

In programming languages such as Timor [7] a distinction is drawn between a *type*, which has a defined behaviour with respect to its users, and individual *implementations* of the type, which may use different techniques to implement this behaviour (see for example [8]). If two or more implementations can achieve the same behaviour, then from the user's viewpoint they are equivalent (except for performance).

In the examples of a trigonometrical library above, the same behaviour (from the user's viewpoint) might be achieved by a file module and by a program module, provided that they also offer the same set of semantic routines. In other words they might be considered to be modules of the same type (in the sense of programming language types). However, SPEEDOS (and other operating systems) do not offer a type concept, but merely implementations. The type issue is considered to be a matter purely for the programming language level, not the operating system level. Nevertheless one small concession will be made below to respect this concept (see section 5.2).

4 Implementing Programs

Given the protection provided by the SPEEDOS segmentation scheme, there is no problem in storing the temporary data of multiple programs in a single container associated with the code module of the program, which serves as a shared heap. This has the advantage that instead of creating a new container each time a thread activates a program a new container must be created only once, when the program is created. But more importantly it unifies the mechanisms for files and programs.

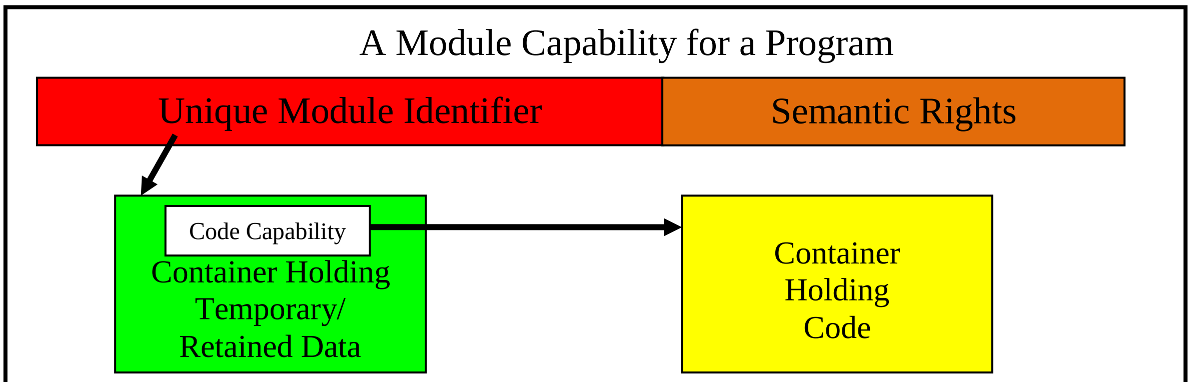


Figure 18.2: Calling a Program Module

A module capability for a program identifies the shared program heap, not the code module, while the latter is reached from a code capability stored in the program heap, as is illustrated in Figure 18.2. A comparison with Figure 14.8 (in volume 1) confirms that programs and files can be handled uniformly, also with respect to inter-module calls.

5 Creating the Different Kinds of Data

The precondition for creating the various kinds of data is that a heap container (i.e. a container in which all the various kinds of data except inter-module linkage and inter-module call and return parameters can be stored) exists which contains a capability for the associated code module. How this precondition is fulfilled will be described in later chapters in connection with the management of containers and the management of co-modules.

5.1 Creating Temporary Data

Whenever a thread becomes active in a semantic routine it needs a root segment for temporary data. For this purpose it calls the Segment Manager co-module to create a new segment and makes this addressable (by convention) via segment register 4. When the thread exits from the semantic routine the temporary data which it has created can be deleted.

5.2 Creating Persistent (File) Data

In order to create a root segment for persistent data a thread must make an inter-module call to a *constructor* routine of the module in question. This is always the semantic routine numbered 0. Its function is to initialise the persistent data of the file module.

The constructor routine calls the Segment Manager to request the creation of a segment and calls the Co-Module Manager to enter the address of this root segment in its co-module table (see chapter 19). It can then initialise this and create further segments linked to it.

When the file module is subsequently called by another module, the kernel automatically loads the address of this root segment into Segment Register 5. However, this segment register can be reloaded with other values by the module's code; Segment Register 5 can be reinitialised to address the persistent root segment using the kernel instruction `reload_persistent_root`, which has no parameters.

In modules without file data (i.e. program modules) there is no semantic routine 0 and module capabilities for program modules have the access right 0 set to 0 (corresponding to no access for a semantic routine 0, i.e. the call is invalid). When a program module is invoked, Segment Register 5 is invalidated, but can be used by the program.

If for a program module an attempt is made to call routine 0, the kernel will simply ignore the instruction and return to the next instruction of the caller. This allows different implementations of a type to function correctly regardless whether they use persistent data or not (see section 3 above).

5.3 Creating and Deleting Retained Data

Before a root segment for retained data can be created a thread must make an inter-module call to an *open* routine of the module in question. This is always semantic routine 1[13]. For openable modules other semantic routines (apart from a constructor) cannot be called until the module has been opened.

The open routine should call the Segment Manager's *create retained* routine, providing it with specifications for the retained root segment. The Segment Manager checks[14] that the module is an openable module and that this call is from an open routine. It then returns to the caller a pointer for the new retained segment *and* an identifier (the time of creation of the segment in milliseconds). After the open routine has initialised the retained segment (by convention addressed via Segment Register 6) it returns the identifier to its caller.

Once an openable module has been opened, further semantic routines can access the retained segment by calling the Segment Manager's *get retained* routine, providing the identifier of the retained segment. If the identifier is valid the Segment Manager returns a pointer to the retained segment. By convention the module loads this into Segment Register 6.

When a thread wishes to close a module it calls semantic routine 2 (the *close* routine), which deletes the retained segment and prevents further invocations of the module via the retained segment identified in the close call.

In modules without retained data (i.e. modules which cannot be opened) there are no semantic routines numbered 1 and 2, and module capabilities for program modules have the access rights 1 and 2 (corresponding to semantic routine 1 and 2) set to 0. Thus if for a non-openable module an attempt is made to call routine 1 or routine 2, the instruction will be ignored by the kernel.

6 Library Modules

One structural issue which has been difficult to manage elegantly in most operating systems is that of library routines/modules. In this context a library module is defined as a body of code (usually with multiple entry points) which performs useful related tasks for an application, e.g.

– a set of mathematical functions,

– a set of routines for manipulating strings,

– a library of routines for organising items within a program into collections such as automatically ordered lists (e.g. alphabetically), unordered lists, us-

[13] From the SPEEDOS viewpoint symbolic names such as "open" and "close" are not important.

[14] To do this it calls the Co-Module Manager, which checks the status word associated with the module (see chapter 19).

er ordered lists, sets (without duplicates),

– separately compiled user defined classes for use in different object-oriented programs,

– routines for synchronising access to data,

– a graphics library which draws and colours shapes, etc.

Some library routines, for example mathematical function libraries, can in fact be handled within the framework already discussed. The real difficulty occurs when a library routine is intended to perform tasks on the existing data of their client module. These are the kinds of library modules which are of interest in this section.

6.1 Libraries in Hierarchical Systems

One reason why library modules have caused problems is that they can be used by many programs which have different protection requirements. This leads to problems for example in connection with the hierarchical (ring) model for protection described briefly in Volume 1 Chapter 9. In order to conform to the hierarchical calling rules – all calls should normally be inwards, i.e. to a lower level in the hierarchical structure – library routines might appear to belong to the innermost layer. But that is the kernel layer where highest privilege applies; it is obviously not a good idea from the viewpoint of protection (or of flexibility) to place library routines there. Thus in hierarchical protection systems quite complicated rules were introduced to avoid this situation.

6.2 Library Routines as Information Hiding Modules

Libraries can normally easily be defined as independent information hiding modules but for many library modules (e.g. string libraries, collection libraries, independently developed OO libraries) that would create a problem, because it would prevent them from directly manipulating the data structures of their calling module. If they were invoked as normal SPEEDOS inter-module calls, which may not pass pointers as parameters, this would imply that the data which the library routines manipulate would have to be passed to them as values and then copied back to the application as return values. This would lead to much inefficient copying.[15]

[15] At this point we note that some library modules can have a dual role in a persistent system such as SPEEDOS. For example typical collection modules such as lists, sets, bags, etc. can be useful for organising data within a SPEEDOS module (and thus correspond to typical collection modules within an application module). But in a persistent system they can also be used as independent information-hiding modules corresponding to files in the file systems of conventional non-persistent systems.

6.3 Library Calls: Espenlaub's Solution

To solve this problem Espenlaub defined a *library call* (LC) instruction in his design for a SPEEDOS kernel. This allows a routine of a library module to receive a pointer as a parameter (in practice a segment register referring to a segment in the application module). Otherwise he described the library module as a normal module (which might for example have its own persistent data). This could theoretically introduce the risk that the pointer passed by the client module to the library module be stored in the latter's own persistent data. However, in practice this cannot occur, because the pointer parameter is defined to be passed simply as a valid segment register, which can only be stored in the container which it addresses.

Nevertheless Espenlaub's solution envisages that such a module can manipulate persistent data of two different modules in two different file containers, which raises protection issues that are not easy to anticipate and to solve. Furthermore the consequence would be that at the programming language and compilation level library modules would have to be treated differently from normal modules. (For example, in object-oriented libraries some segment of the calling module's data should be treated as the root segment of the library module, not as a parameter.) For such reasons we now define a safer alternative for SPEEDOS.

6.4 The New SPEEDOS Solution: Library Calls

It therefore appears to be more appropriate to view library code which is designed to operate within an application module simply as an extension of the application's code, and the library call as a simplified call which leaves the current data container active, but which switches the code to the appropriate entry point in the library module, based on a code capability (not to be confused with a program capability) provided as the main parameter.

We consider first the case of "library file" modules. These modules create new data structures for their client modules and view this as their persistent data (e.g. a collection library). From the viewpoint of the library module they need a constructor. Instead of an inter-module call (IMC) being used for this purpose the host module calls a *library file constructor* (semantic routine 0) using a new kernel library call (LC) instruction. The operands of this instruction are a *code module* capability for the library module, an entry point number and an optional root segment address (which is initially a null pointer). Like a normal file constructor this creates a root segment for what it regards as its persistent data. But instead of calling the Co-Module Manager to set up a pointer for this root segment in a CMT entry (see chapter 17 section 5.1), it returns the pointer to the constructor's caller. The reasons for this are that such a library module has no CMT entry, and more significantly, it has actually constructed an abstract data structure for its caller (in the same container) and the calling module can then root this into its own (persistent, retained or temporary) data structures (which might for example include several collections).

Thereafter further calls from the host module to the individual semantic routines (in Timor parlance, to the methods) of the library instance are also made by invoking the kernel's LC instruction, passing to it the appropriate code module capability, an entry point number and a pointer to the root segment library module instance's root module[16].

If the library module requires retained data, its host module must first use a kernel library call instruction (LC) to invoke the *open* routine (semantic routine 1) of the library module in question. For openable library modules other semantic routines (apart from a constructor) cannot be called until the module has been opened.

The open routine calls the Segment Manager's *create retained* routine (as for normal modules), providing it with specifications for the retained root segment. The Segment Manager returns to the caller a pointer for the new retained segment *and* an identifier (e.g. the time of creation of the segment in milliseconds). The open routine returns this to its client module which can store it in one of its own data structures for use in subsequent calls to the library module.

Once an openable library module is open, further semantic routines gain access to the retained segment by calling the Segment Manager's *get retained* routine, providing the identifier of the retained segment. If the identifier is valid the Segment Manager returns a pointer to the retained segment.

6.5 Evaluation

One aim of the design of library modules is to allow them to be programmed and compiled (almost) like normal modules. With this approach to libraries, the software for these can be written by programmers (e.g. in Timor) exactly like normal modules. They have to be compiled slightly differently, but provided that the compiler knows it is compiling a library module, it can easily hide these differences from the programmer.

One basic difference is that instead of finding the required code capability[17] in the CMT, the kernel receives it as part of the LC instruction. Such capabilities

16 In chapter 22 a special feature will be described allowing library methods also to be invoked via the kernel.

17 The evaluation of the code capability is via a Code Table (to be described in Chapter 19). Since this holds a qualifier list, the code of library routines can be bracketed separately. Since the data produced/manipulated by a library module is considered to be part of the client module, its methods can be bracketed using the qualifier list in the CMT (see chapter 19).

can be provided to the host module as parameters (either to its constructor or even individually with each independent library call or sequence of library file calls). This makes the use of libraries very flexible.

In contrast with inter-module calls, library calls can pass pointers to additional segments, thus enabling the library module to address several of its segments (e.g. if its function is to merge two lists into a third). These may also be set to "read-only".

7 Cooperating Co-Modules

While the information hiding principle is basically a very sound concept, there are some circumstances in which a limited amount of sharing of data between separate modules may be justified at the application level. For example, a central organisation (e.g. the headquarters of a company), with a central database of information, might have subsidiary organisations (e.g. franchisees, local branches) which have a legitimate need to share some (but not necessarily all) of the central information. Each sub-organisation may also have its own additional information in a further database, to which the central organisation may need (partial) access. In such a situation a strict adherence to the information hiding principle would result in an extremely inefficient design, with considerable copying of data (and the creation of duplicate copies which risk becoming out of step with each other).

7.1 Application Co-Modules

To facilitate such environments SPEEDOS allows related application modules to be implemented as co-modules in the same container. In this situation the individual co-modules can have separate databases and separate semantic routines with separate capabilities, thus providing them with individual control over their data and access rights. At the same time, by allowing them to make calls to each other which relax the normal calling rule forbidding the passing of pointers, they can share access to selected parts of each other's databases in a controlled way. For this purpose a further call instruction, a *co-module call* (CMC) is supported by SPEEDOS. This is like an IMC, except that pointers may be passed as parameters. The kernel checks that such calls only take place between co-modules in the same container. Because they are in the same container there is no formal problem with passing segments as parameters.

7.2 Passing Segments as Parameters

The following rules are used to ensure that segment pointers, passed as parameters via LCs and CMCs, are not misused.

a) The entry points of a module which expect to receive segment pointers as parameters are marked as such in the entry point list of the code of the module. The destination of the call must be either library code or a co-module in the same container. Other entry points may not accept pointer calls.

b) A pointer can only be stored in the container which holds the referenced segment.

c) A bit in a pointer indicates whether it can be loaded into a segment register in "read-write" or "read-only" mode. The latter indicates that the contents of the addressed segment can be read but not modified[18].

d) The bit in a pointer which indicates whether it is "read only" must always be stored when the pointer is copied. This bit is also copied to segment registers, indicating that the content of the addressed segment may not be modified.

The interface routines of co-modules which can pass pointers can be qualified with bracket routines.

7.3 Kernel Co-Modules

Co-module calls (CMCs) can of course also facilitate calls from application modules to kernel co-modules in the same container (e.g. when an application wishes to create a new segment or delete a segment it can use a CMC to call the Segment Manager for this purpose. Similarly one kernel co-module (e.g. the Segment Manager) may also need to call another kernel co-module (e.g. the Virtual Page Manager) in the same container and uses a CMC to achieve this.

7.4 Co-Module Calls, Library Calls and White Box Functionality

Finally, neither of these kinds of call provides normal white box functionality, i.e. access to the entire data of a module is not permitted, unless the pointer passed is the root segment of the caller. Otherwise white box functionality is only achieved by an inter-module call to a module which the Co-Module Manager has organised to share a root pointer with another module. How it does this will become clear in the next chapter.

8 Free Capabilities

There are two further situations in which the strict use of information hiding can lead to inefficiencies.

8.1 N-ary Operations on Files

The word n-ary is built on the pattern of words such as unary (for one), binary

[18] Module capabilities in the addressed segment can be copied, unless the generic access rights in the module capability itself prevent this.

(for two), etc.; it means "for n", where n is a number whose value is not fixed. The n-ary problem arises when a method wants to manipulate two or more similar objects together. For very small objects (e.g. integers) most operations are n-ary and they are built into the computer's basic instruction set. For example the addition operation takes two objects and produces a third object as a result. At the segment level, library modules which provide n-ary services (e.g. for merging or comparing two segments in the same container) solve the problem.

Most file operations are not n-ary, because their semantic routines normally involve accesses to a single file; consequently the information hiding principle is normally unproblematic and, as we have seen, brings many security and software engineering benefits. But it is nevertheless inconvenient in cases such as that of a routine which merges two files or compares their contents. This can theoretically be achieved in an information-hiding framework by using a routine of a third module, which reads from and/or writes to them individually, using their semantic routines. However, such an approach can be very inefficient because of the overhead of the many inter-module calls required. If all the objects concerned were created by the same code module, this restriction is not even necessary to preserve the information hiding principle! Furthermore, it is virtually impossible in situations where the content of a file is long and is not easily decomposed into small segments, as is the case for example with video files.

The passing of capabilities for such files as parameters is not a problem – a parameter segment for an inter-module call can always contain module capabilities. The issue is how the content of the corresponding files can be addressed. All that is needed in the module carrying out the n-ary operation is access to the root segment of each of the parameter files' root segments. For this purpose the kernel provides an instruction `load_free_cap` which takes as parameters the file capability passed and the number of the segment register which the n-ary routine chooses to use to address the file's segments.

We refer to capabilities which can be passed in this way as "free capabilities". These must have the metaright "permit free capability" set (see chapter 26). Their use can be further restricted to "read only" as defined in a further status bit in the capability. Using this feature an n-ary routine to merge two files into a third can for example be implemented by passing two read only parameters to a further module which creates a new file.

It is tempting to suggest that the kernel should also check that the code file of the parameter file is the same as that of the called module. But that would rule out a solution for the next problem.

8.2 File Conversion

Suppose that some file type can be implemented in two quite different ways. For example bank accounts files might be implemented using a B-tree technique or an indexed sequential (IS) technique, and a banking company might from time to time convert particular files from one implementation into another, e.g. because one of the techniques provides faster access or uses less space. Similarly video files can have different internal formats and conversion between these is quite common. After such a conversion users of a converted file could continue to use it as if nothing had happened, assuming that both implementations support the same type definition[19]. In that sense the possibility of making such conversions could be said to encourage the information hiding principle.

To implement this, a conversion module might for example take one module capability in read-only mode as the source of the conversion and create a second, in the new format, as the new file.

Such examples make it tempting to introduce the idea of *file types*[20] at the kernel level, and to allow only files of the same type to be attached to other files. But not only would that be extremely complicated (e.g. because of inheritance), it would also exclude more general conversions. For example it would virtually rule out in practice an efficient form of conversion from normal text to compressed text or encrypted text, etc. Consequently it is better not to enforce any information hiding rules in the matter of passing files as parameters to modules.

8.3 Precautionary Measures with Free Capabilities

To ensure that this feature is not misused (for example by a user with a restricted set of access rights for a file) the instruction which the kernel provides to make a free parameter capability for a file checks that the capability from which the free capability is copied, is an owner capability, thus ensuring that only the owner of a file can make free capabilities, marking them as such in the free capability's metarights field.

Input parameters can only be loaded into segment registers in read-only mode, and the segment register is marked as non-storable, thus guaranteeing that confinement measures cannot be avoided by using free capabilities. One implication of this is that if files are to be merged or converted, the module carrying this out must be the output module (or must call a further module to carry out the output operations).

There is one final security risk to be avoided: this facility should not be al-

[19] In a later chapter we discuss how they might obtain a capability for the new version.

[20] A file type here means a file module interface which can be implemented in different ways but provides the same functionality.

lowed to circumvent bracket routines associated with a module. This is discussed in Chapter 24 section 5.

9 Call Back Calls

There are situations in which a normal module may have a need to call its calling module back. This can happen at two levels, at the inter-module level and at the library level. These require different but similar SPEEDOS kernel instructions.

9.1 Inter-Module Call Back Calls

The SPEEDOS kernel supports a 'call back call' (CBC) instruction. This activates a call back routine in the application module by nominating an entry point number in a special call back entrypoint list. In contrast with normal inter-module calls, the caller does not provide a capability for the call back call; instead the kernel discovers the details of the CBC destination by examining which module called the currently active module and activates the nominated call back routine. When the call back routine terminates it returns back to its caller, which resumes execution at the instruction following the CBC instruction. There is no limit on the number of call back calls a module can make.

9.2 Library Call Back Calls

For this case the kernel provides a 'library call back' (LCB) instruction which discovers the destination routine by examining which internal module invoked the original library call (LC), see section 6.4.

10 Conclusion

This and the previous chapter have provided a basis for the kernel's structure and some of the kernel's most significant security functions. With this as background information we are now in a position to delve more deeply into the functionality of key kernel co-modules and how they interface with the kernel.

Chapter 19
Containers and their Contents

Containers play a central role in SPEEDOS. They serve as discrete address spaces for the persistent distributed virtual memory. They hold persistent data which might be used, for example, as databases for user applications. They can be used to hold executable code. Hence they serve a similar role in SPEEDOS to that of files in conventional systems. However, that is only part of the story. They can also hold information (persistently) about user processes and their threads, which in conventional systems is a function of the (non-persistent) virtual memory.

This chapter describes how containers are organised and provides an overview of their functionality, beginning with the way they are identified.

1 Container Identifiers

Section 2 of volume 1 chapter 16 described the SPEEDOS technique for supporting world-wide unique virtual addressing. Logically a world-wide unique virtual address consists of a unique container identifier concatenated with a within-container offset. The unique container number is expressed as three contiguous entities, as is shown in Figure 19.1 (which is a repeat of Figure 16.2). The actual sizes and further subdivisions of these fields are discussed in chapter 23 and Appendix 1.

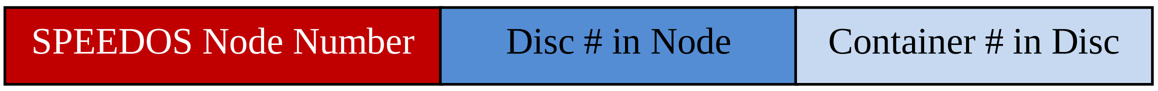


Figure 19.1: A SPEEDOS Container Identifier

Concatenating this identifier with a within container address results in a world-wide unique virtual memory address which is far too large to allow an ATU to be built cost-effectively. To avoid this we suggested in volume 1 chapter 16 (section 3.3) the use of 3 bit SCIDs (short container identifiers).

2 Container Red Tape

The *Container Manager* at each node is a security-sensitive co-module which performs node-wide functions and is therefore itself located in a container which is not associated with a particular application co-module. It is responsible for providing the functionality necessary to create other containers, as will be discussed in section 12 below.

When a new container is created several unique identifiers are stored in fixed locations within it which are known to the core kernel. These uniquely identify:

a) the user who created the container,

b) the container itself,

c) the thread which created the container,

d) the code module via which the container was created,

e) the co-module via which the container was created,

f) the user who currently owns the container,

g) the date and time of creation, and

h) its activity status (e.g. the count of currently open co-modules in the container).

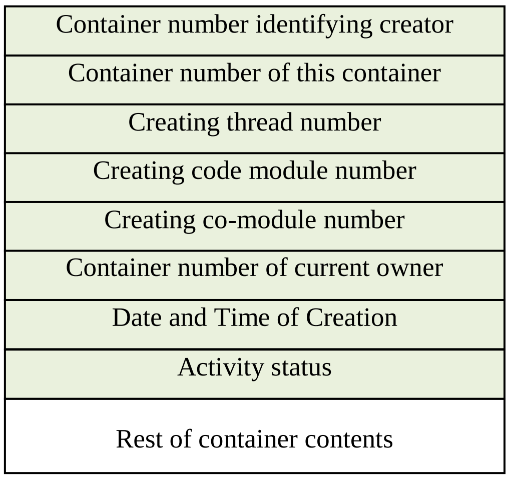

Container number identifying creator
Container number of this container
Creating thread number
Creating code module number
Creating co-module number
Container number of current owner
Date and Time of Creation
Activity status
Rest of container contents

Figure 19.2: Identification Fields of a Container

This information is set up in page 0 of the container by the kernel's `new_container` instruction.

When a new container is created, the semantic routine responsible for this returns a *container capability*, which entitles its possessors to load co-modules into the container (see Figure 19.3). The index field is set to -1.

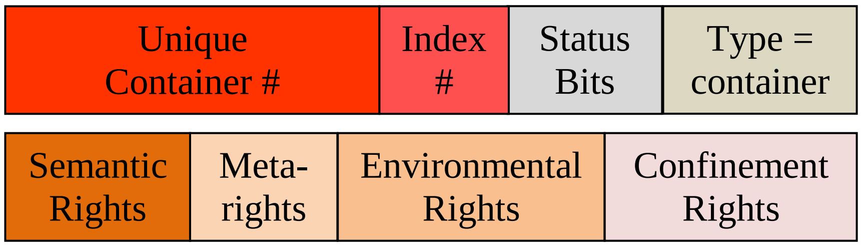


Figure 19.3: The Basic Structure of a Container Capability

3 Using Containers for Multiple Purposes

There are two senses in which containers are used for multiple purposes. First, as was described in the volume 1 chapter 18, a container can hold several co-modules; the kernel relies on some of these to carry out its privileged activities. Second, as foreshadowed above, containers can be used for three different primary purposes: as repositories for persistent data (henceforth called *data files*[21]), for code (*code files*) and for processes (*process files*). In fact, all modules are initially created and start their life as data files. The type field in a capability determines how the container should be viewed when accessed via that capability. Notice that when the type field is set to 'container' the rights fields indicate what actions are possible when the container is being viewed as an entire container (e.g. whether the entire container can be copied, etc.) The Container Manager is not a component of every container, unlike the co-modules about to be discussed.

4 Segment Management

Each container must have a Segment Manager co-module, and each Segment Manager must provide certain standard routines. These are provided as part of the SPEEDOS system, but this module can be extended in different ways (e.g. using the Timor inheritance and code re-use techniques) allowing different containers to manage segments differently, e.g. with respect to garbage collection. When a new container is created the Segment Manager, like the Co-Module Manager, is pre-installed in the Co-Module Table (see section 7). An important part of the segment manager's work is to ensure that the segments in a container do not overlap.

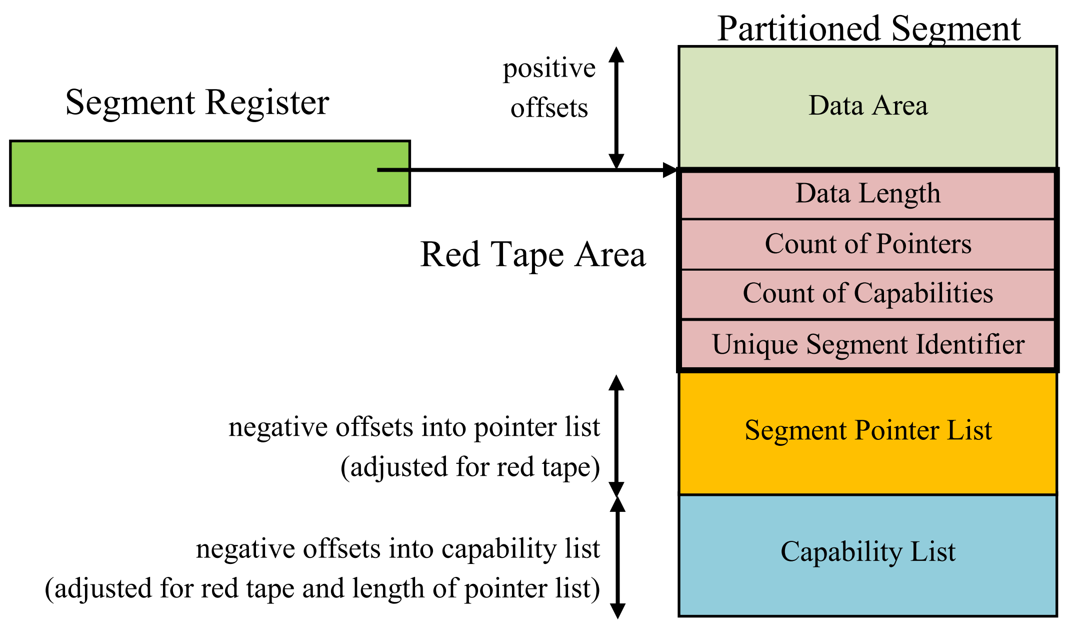
All modules in the container may need to create temporary segments and can access the appropriate Segment Manager routines using CMC calls (thus allowing pointers to be passed as parameters), provided that they can obtain a capability for the container's Segment Manager. Only the operations of file

[21] In this sense each co-module is a data file consisting of a persistent data structure pointing to a code file containing its associated semantic routines.

modules need to create file segments and to re-link temporary segments into existing persistent segments.

4.1 Segment Structure

SPEEDOS segmentation is based on the idea of partitioned segments, which was first proposed by Anita Jones [9]. Because SPEEDOS has two kinds of capabilities (i.e. segment pointers and module capabilities) the original structure proposed by Jones (see Figure 10.7) has been modified to that shown in Figure 19.4.



Partitioned Segment
Segment Register
positive offsets
Data Area
Data Length
Red Tape Area
Count of Pointers
Count of Capabilities
Unique Segment Identifier
negative offsets into pointer list (adjusted for red tape)
Segment Pointer List
negative offsets into capability list (adjusted for red tape and length of pointer list)
Capability List

Figure 19.4: SPEEDOS Partitioned Segments

They have four basic areas: a data area, a red tape area, a segment pointer list and a module capability list. The content of a segment can only be accessed via a segment register.

The user can address the data area directly using normal (non-privileged) CPU instructions, which contain (non-negative) offsets from the base address in the segment register. Because the segment register contains a length field the hardware can ensure that the user cannot address information outside the data area without causing an address violation interrupt.

Below the data area is a "red tape" area which can only be accessed by the kernel. Attempts by users to access this area via negative offsets cause an address violation interrupt. The red tape consists of a length field for the data area, a count of pointers, a count of capabilities and unique (within container) segment identifier. The red tape area can only be addressed by the kernel. Segment pointers and module capabilities can only be addressed indirectly via (separate subsets of) kernel instructions, using non-negative integers as offsets. The first pointer is numbered 0, the second is 1, etc. Module capabilities are also addressed in the same way, i.e. capability 0 is the first, capability 1 is the second, etc.

A pointer is a single word offset in the current container, which refers to the red tape area of a different segment in the same container. There is no way that pointers can refer to addresses in other containers. Pointers cannot be directly accessed by applications; they can only be manipulated via kernel instructions. One of the pointer instructions allows a segment register to be loaded from a pointer. In this case the kernel loads the address of the referenced segment into the specified segment register, using the current segment register's details to complete the address. It then follows the pointer to locate the addressed segment, and uses the information in the referenced segment's red tape to load the length field into the segment register. If the current segment register access rights are set to read-only, it copies this also into the referenced segment register. The user also has the option to set the referenced segment register to read only, regardless of the setting in the current segment register. If access is set to read only then the user can only read the information in the data part of the segment.

Since several pointers can be associated with a segment, the mechanism allows arbitrary linked lists, tree structures, etc. to be created. Alternatively a compiler can choose to store all the necessary persistent information in a single segment and manage this itself.

Module capabilities are always stored in the protected area of segments. All the instructions which use them are kernel instructions which have operands that specify a segment register and module capability number. A capability can be copied into the data area of a segment, but only if the corresponding metaright (`permit read`) in the capability permits this. Such a copy cannot be used as a capability.

5 Distributing Standard Capabilities

There are at least five cases in which a module executing in a particular thread may legitimately need a capability for the thread itself or for another module, where the normal distribution methods for capabilities would lead to clumsy, complicated and inefficient solutions. These are capabilities for the

- *Segment Manager* associated with the container of the current module, in order to create and delete segments in that module's container, on behalf of the module;
- *current thread,* i.e. the thread which is currently executing, needed by some modules for synchronisation purposes;

- *Thread Manager* for the current thread, to allow sub-threads to be created by an application module;
- *standard input and output modules* (if any) currently associated with the thread, to allow appropriate modules to communicate with the user, e.g. current keyboard and screen devices;
- *Print Request Manager*, i.e. the module which accepts print requests from other modules executing in a thread (see chapter 33).

These are stored at the base of each user thread stack in an area known as the capability accessibility area and are set up by the Thread Manager when it creates a thread.

The kernel provides instructions which allow the code of an executing module to obtain them. However, simply to make capabilities for the Thread Manager, the current thread (i.e. its thread capability) and/or the standard input and output modules available for all modules executing in a thread would potentially be very dangerous from a security point of view. To ensure that only the modules which really need these capabilities are provided with them, the kernel provides a mechanism which allows it to check whether the request made by a module should be honoured. This mechanism is described in Chapter 26.

The kernel provides two different capabilities for the Segment Manager with different access rights. These permit the possessor of the capability to create either

– persistent segments for *operations* of file modules[22] and to link temporary segments to persistent segments, or

– temporary and retained segments for *enquiries* and to link them to other temporary and retained segments.

Since the kernel can recognise whether a request comes from an operation or an enquiry, the appropriate capability can be issued without reference to any confinement permissions. For this purpose constructors are regarded as operations while open routines are regarded as enquiries.

6 Virtual Page Table Manager

Each container has a Virtual Page Table Manager, which is responsible for translating virtual addresses into disc addresses. It is one of the first co-modules to be loaded into a container, and its mode of activity can vary from container to container, depending on the planned content to be located in the container. Its role is described in more detail in chapter 23.

22 *Operations* in Timor and SPEEDOS are semantic routines which can create and/or modify persistent information. *Enquiries* can only read persistent information.

7 Data Files, the Co-Module Manager and the Co-Module Table

As we saw in chapter 18, access to co-modules, when they are viewed as data files, is achieved via a *file capability* which is passed as an operand to a kernel call instruction (see Figure 19.5). The kernel interprets this capability via the Co-Module Table (CMT), which is set up and managed by a Co-Module Manager in each container.

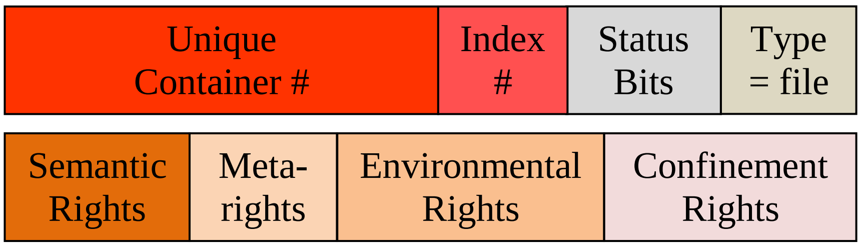


Figure 19.5: The Basic Structure of a File Capability

The structure of the CMT is fixed because it is one of the data structures accessed directly by the kernel[23]. The Co-Module Table is an array (i.e. a directly indexed list) of entries, one for each co-module in the container. Figure 19.6 illustrates the basic format of the table. Each entry is logically structured as a segment.

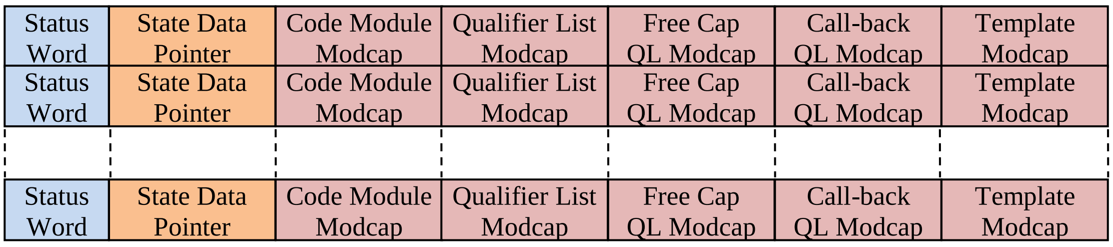


Figure 19.6: The Co-Module Table (CMT)

Since each data file is one of several co-modules in a container the container number part of the capability is modified by an 8-bit index value to designate which co-module in the container is intended.

The Co-Module Manager itself occupies the first entry in the CMT. The kernel can locate the CMT from a pointer (at a fixed position) in the red tape of the container.

The state data pointer contains the start address of the root segment of the persistent data of the co-module. (In the case of a program module this is a pointer to the single heap permanently associated with the program, see chapter 18, section 1.3.) When a constructor creates a root segment (using the Segment Manager's create_persistent_root routine) the Segment Manager advises the Co-Module Manager, which sets up the pointer in the CMT.

[23] This is necessary because inter-module calls are interpreted via the table.

White-box sharing of an application's data structures (e.g. for use by a debugger) can be achieved by setting the sharing co-module's State Data Segment Pointer to the same value as that of the application co-module. However, this can only occur when the application is not active. Special permission is required for a co-module which needs white-box functionality.

The *code module modcap* entry is used by the kernel to locate the code container of the module on file, program and cooperating co-module calls. (For library calls a code module is passed as a parameter to the library call.) Code modules are described in more detail section 9.

The *qualifier list modcap* provides the kernel with access to a further co-module which provides a list of the qualifiers for this co-module (cf. volume 1 chapter 13). Its routines are called by the kernel to control the sequencing of bracket routines associated with the module (see chapter 24).

The *free cap QL modcap* provides the kernel with access to a further qualifier list module which is used to bracket the module when it is being accessed via a free capability (cf. section 13.3 below).

The *call-back QL modcap* indicates the qualifier list module which is used to bracket call-back calls (see chapter 28).

The *template modcap* is explained in chapter 32 section 2.2.

Each co-module in a composite module can be separately qualified and can have multiple qualifiers associated with it. The kernel can locate the qualifier list for a co-module from the module capability in the CMT.

Each co-module in a container has a *status word* in the CMT indicating whether it requires a *constructor* call to initialise persistent data, whether it requires *open* and *close* calls (which can create and delete *retained* data), and further information about the status and current activity of the module.

There is also an open status for the entire container (not illustrated above), which is maintained by the kernel. This is not associated with a particular open/close call, but provides the kernel and its co-modules with the global status, e.g. whether a co-module is open at all and if so how many threads are active in the module as readers or writers. This is relevant, for example, to determine whether an external disc can be safely removed from the node to which it is currently attached.

8 Creating File and Program Modules

In order to create a (co-)module, an entry must be made in the co-module table of the container in which the module will be placed. The functionality for doing this is provided by the Co-Module Manager's semantic routine `create_module`,

It has four parameters:

– a code capability for the code file to be associated with the data file or program, which it places in the new entry;
– a file capability for a qualifier list (may be null);
– a boolean value indicating whether the new module is a program module (i.e. without persistent data) or a file module;
– a boolean value indicating whether the new module is openable.

The "pointer to root data" field in the CMT entry is set to null[24]. The Co-Module Manager routine then organises the manufacture of a file or program capability, which *inter alia* holds the container number, the file index used for the new table entry and access rights copied from the code capability; all rights and permissions[25] and the owner bit are set. The routine checks that the access rights for semantic routines 0 (constructor), 1 and 2 (open and close routines) are correctly set then returns this capability to the caller.

9 Organising Code Files

Code files (as in conventional systems) start life as data files – usually created as the output of compilers and/or linkers. They organise the code in normal segments, i.e. with data, pointer and capability partitions.

The code itself is stored in the data partitions, which can also be used to hold constant data segments.

The pointer partitions can be used to link code segments together (e.g. for different subroutines) and provide access to constant segments. Thus the compiler can organise the code such that individual code and constant segments are separately protected (e.g. with separate bounds checks).

The capability partitions can hold module capabilities, which can for example provide the code with access to library and other ancillary modules (e.g. a spelling checker module for editor code). The capability partitions of security-sensitive kernel co-modules can also be used to hold kernel capabilities which provide them with special privileges.

In SPEEDOS a data file can be converted into a code file by calls on the semantic routines of a *Code Manager* co-module, which must reside in the same container as the code of the new code file. Thus possession of a capability for a Code Manager is a prerequisite for creating code files. System administrators can therefore use the distribution of Code Manager capabilities to control the

[24] For a file module the persistent root is subsequently set up in the CMT when the Segment Manager's `create_persistent_root` routine is invoked, see section 7.

[25] The permissions are described in Chapter 26.

right to introduce executable code into a system. When the Code Manager has completed a conversion operation it returns a *code capability* to the caller, as illustrated in Figure 19.7.

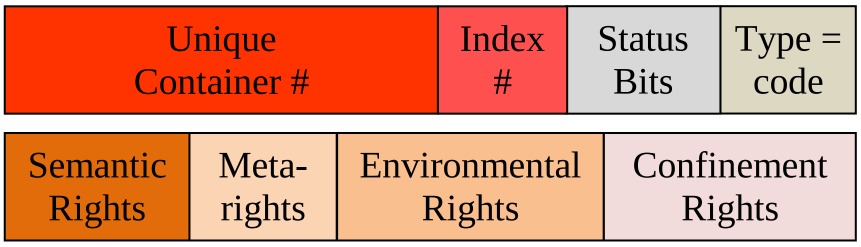


Figure 19.7: The Basic Structure of a Code Capability

A code capability has the same structure as a data capability except that (a) the index field is used to select one of several potential code files held in the same container and (b) the type field in the capability is set to "code".

The primary purpose of a Code Manager is to organise a Code Table in the container in which it resides. Since the core kernel relies on the correctness of this table, the Code Manager, like the Co-Module Manager, plays a central role in maintaining the integrity and the security of a system. (In order to establish its right to create the Code Table, the Code Manager must present a kernel capability with the appropriate access rights.)

As in the case of a Co-Module Manager, there is only one Code Manager in a container. This has a separate entry for each code file module. The structure of a Code Table, which differs substantially from that described by Espenlaub, is illustrated in Figure 19.8.

↑ External EPL	↑ Bracket EPL	↑ Internal EPL	Qualifier List Modcap	Free Cap QL Modcap
↑ External EPL	↑ Bracket EPL	↑ Internal EPL	Qualifier List Modcap	Free Cap QL Modcap
↑ External EPL	↑ Bracket EPL	↑ Internal EPL	Qualifier List Modcap	Free Cap QL Modcap

Figure 19.8: The Code Table

Each entry contains

– a pointer to the code module's external entry point list (i.e. the entry points used by the kernel to locate the appropriate semantic routines of the code file on inter-module calls);

– a pointer to an entry point list for the module's own bracket routines (if the code is for a qualifier module), to be discussed in Chapter 23;

– a pointer to an entry point list for internal calls (see below);

– a module capability for the list of qualifiers currently associated with the semantic routines of the module, to be discussed in Chapter 24.

Each entry in the Code Table provides information about one code file in a container.

9.1 Entry Point Lists

An entry point list pointer in a Code Table entry refers to a segment which describes the information needed by the kernel to activate the various entry points (semantic routines, bracket routines and internal calls) of the code. An Entry Point List (EPL) consists of a single segment containing in its pointer partition an array of pointers to segments in the container holding executable code, and in the data partition an array of integers which serve as offsets into the corresponding code segment pointer, each indicating the start address within the code segment for that routine[26] (see Figure 19.9).

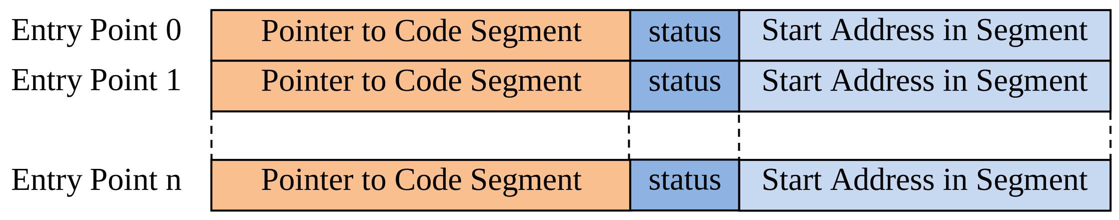


Figure 19.9: An Entry Point List

For each entry point there is a status bit indicating whether the entry point's routine is an enquiry or an operation. When an enquiry is activated the segment register which addresses the module's root data node is set to read-only access. For an operation it is set to read-write access. A further status bit indicates whether the routine can receive pointers as parameters.

Since some or all of the code segment pointers can point to the same segment, it is possible, for example, for a compiler to compile the code of all the routines into a single segment, or to have a separate segment for each routine. EPLs are created by the compiler/linker as part of normal compilation, but the Code Table is created by the Code Manager.

Within the same code container the structure allows different code modules to share code. This can be used for example to create a new code file which corrects errors in some segments of an earlier version, or to allow code re-use, e.g. between a queue module and a double ended queue module. For further advantages of this organisation of code modules see Espenlaub [4] chapter 7.3.

Based on the information in the appropriate EPL the kernel loads the in-

[26] In accordance with the RISC philosophy the SPEEDOS EPLs, in contrast with those used in the MONADS-PC, do not hold information about the parameters. Parameter checking is left to software.

formation from the appropriate code segment into the Code Segment Register. (This is a dedicated segment register which always refers to the currently active code segment.) The corresponding Start Address is loaded into the Program Counter register PC, which serves as an index into the code segment and is automatically incremented to the next instruction on the execution of normal instructions and modified on jump instructions. The hardware always checks that the PC remains within the bounds of the segment.

9.2 Inter Module Calls, Co-Module Calls and Library Calls

When an IMC or CMC instruction is executed, the kernel uses the appropriate entry in the Co-Module Table to locate the relevant code capability associated with the data or program file (see Figure 19.6). For LC instructions the capability is passed as a parameter to the kernel. With this capability the kernel can use (a) the container number in the code capability to locate the container holding the code and (b) the index field to locate the appropriate Code Table. The appropriate semantic routine is found by indexing into the code module's external entry point list. A more complete description of these and the following calls appears in Chapter 20.

9.3 Bracket Entry Point Lists

The second entry in a Code Table points to a further entry point list with the same format. This contains information about the code module's own bracket methods. These are not directly activated by inter-module and similar calls; the kernel activates them in the course of a call to another module, when it detects in the latter's Co-Module Table or Code Table that there is a module capability for a Qualifier List. The activation of bracket routines is described in more detail in Chapter 24.

9.4 Internal Entry Point Lists

An *internal call* (IC)[27] pushes the current value of the Code Segment Register and the Program Counter to the top of the current kernel thread stack and loads the details of the new code segment into these registers. The destination of the call is specified as an entry point number into a second entry point list, called the Internal EPL or IEPL. The IC instruction has an integer operand which serves as an index into the IEPL.

The kernel provides no support for passing parameters or return values, nor does it provide support for any particular high level programming language scope rules. These issues are best left to the compilers of the various languages.

[27] This is not described in Espenlaub's thesis.

Note: Internal calls are only needed when a subroutine is located in a different code segment from the calling subroutine. The compiler can completely take care of internal calls within the same segment, provided that it has conditional and unconditional jump instructions and a jump subroutine instruction, all of which use within container offsets for code addresses.

9.5 Subthread Entry Point Lists

Creating a new subthread (see chapter 20 section 8.2, chapter 31 section 2.5) requires that the kernel has access to a Subthread Entry Point List. This has the same structure as other EPLs, see Figure 19.8. How this is used is explained in chapter 31 section 2.5. Subthreads have no explicit parameters. Like other routines they can be given read-only or read-write access to the state data of a module. They can make calls (e.g. inter-module calls, internal calls). They are bracketed in the same way as the module in which they are activated.

9.6 Return Instructions

The same `return` instruction is used for all calls except for bracket routines; the latter are activated by the kernel, and must use a `bracket_return` instruction). These reload the stored values from the corresponding call. They can behave differently (see chapter 31 section 2.7), depending on the corresponding call and on whether they reach a backstop, i.e. a special marker on the thread stack which indicates the logical stack bottom.

10 Organising Process Files

Like code files, user process files start life as data files, which in this case contain a Thread Manager co-module. The latter creates a Thread Table (analogous to a Co-Module Table and a Code Table) with one entry for each thread within a process container. When a new thread is created (by invoking a further semantic routine of the Thread Manager) it returns a *thread capability* (Figure 19.10) to the caller.

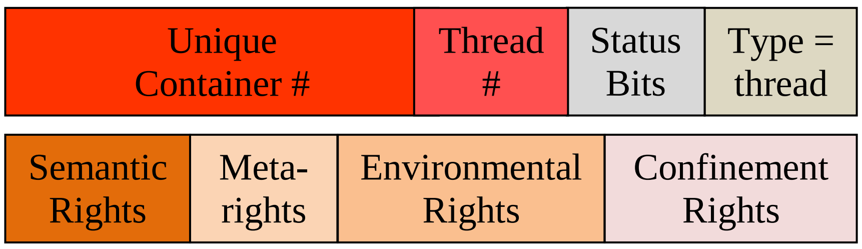


Figure 19.10: The Basic Structure of a Thread Capability

The semantic methods permitted in a thread capability can be used to control the thread (e.g. to suspend it or activate it). These differ from data capabilities and code capabilities in that the index number is used as an index into the

Thread Table (i.e. it is a thread number) and the type field indicates that the capability is a thread capability.

The thread table itself is simply a segment containing an array of pointers (one per thread of the process), which the kernel accesses to locate further information about each thread. This includes a linkage stack for the thread and the contents of all the CPU registers used by the thread, as well as some pseudo-registers which will be explained in later sections. The register values and other status information are stored when a further kernel co-module decides that a thread should no longer actively execute on a CPU, and are re-loaded when it decides to schedule the thread.

The linkage stack contains the linkage information which is extended when the thread executes a kernel call instruction and is contracted when the thread exits from the currently active module. The parameters and return values of inter-module calls are passed via an input segment and an output segment on the thread stack. The kernel ensures that pointers cannot be passed. The passing of parameters is described in more detail in Chapter 20.

11 Multiple types in a Single Container

The above discussion perhaps leaves the reader with the impression that file containers, code containers and thread containers (after the last two have been established in a data container) are quite separate container types, the functionality of which does not overlap. This is how the SPEEDOS operating system normally uses these containers. But such usage need not have this exclusive character in all operating systems which run under the SPEEDOS kernel. For example in an operating system which uses the out-of-process model (see volume 1 chapter 8) it is conceivable that a single container might hold the data and the code of a module and at the same time one or more threads which are dedicated to executing this module. Thus the coexistence of a Co-Module Table, a Code Table and a Thread Table within a single container is possible. Even in a SPEEDOS operating system environment it might, for example, be sensible to place the data and the code of a *singleton* module (a module where the code has only a single instance of the data) in a single container.

12 Creating a New Container

A new container is created at the request of a user with a capability for the Container Manager which includes the appropriate access right. One of its functions is to allocate a world-wide unique identifier for the new container.

In the course of creating a usable container a number of security sensitive co-modules (e.g. instances of the Co-Module Manager, the Segment Manager and the Virtual Page Table Manager) must be installed and instantiated.

12.1 Preparing the Security-Sensitive Co-Modules

Such modules typically consist of (a) a core part, which provides mechanisms that are vital for the kernel's correct functioning, and (b) a more flexible support part which can be tailored to suit the needs of particular environments and user requirements.

To maintain this flexibility without risking the danger that sensitive mechanisms are implemented either maliciously or erroneously by individual programmers, the core kernel distribution includes a rudimentary set of pre-prepared modules which can be *extended* in different ways at different nodes using the Timor type derivation and code re-use techniques. For this reason, most of what follows can be viewed as examples of how these co-modules might be extended, rather than as a definitive version of a SPEEDOS operating system.

12.2 Protecting the Code Capabilities for Security-Sensitive Modules

Because the code capabilities which are needed to construct these modules are themselves very sensitive, they should not be made generally available. A technique for restricting their circulation to the modules which need them is to store them in the constant segments of the modules which use them (e.g. a capability for the Co-Module Manager in the constant segments of the Container Manager's code). Hence when the Container Manager is called to create a new container, it already has a code capability for a Co-Module Manager, which it can use to construct an instance of the latter for the container. Similarly the Co-Module Manager already has at hand capabilities allowing it to create a Segment Manager and a Virtual Page Table Manager for a new container.

12.3 Constructing the Initial Data

Some of the kernel co-modules constructed in the course of creating a new container need to have a persistent data root in page 0 of the new container, in some cases at specific addresses known to the kernel. To allow them to set up their persistent roots, their constructors (invoked by calling routine 0 via the respective code capability) need direct access to page 0 of the new container. For this reason a special kernel instruction[28] (`load_page0`) requests the kernel to load segment register 5 (used normally to address the persistent root segment of a module[29]) such that it has read-write access to the new container's first page[30].

[28] This is one of several new kernel instructions, not mentioned and in some cases not envisaged by Espenlaub.

[29] see Chapter 18.

[30] It will become clear in chapter 23 how the first page of a container is physically allocated space on disc.

To ensure that this instruction cannot be misused by other modules, a kernel capability must be presented as an operand. This also is stored in the constant data of the respective code module. The remaining operands are passed in the general purpose registers. (At this stage in the construction procedure the module has no persistent data except in these constant segments.)

12.4 The Container Manager `newContainer` Routine

Like other sensitive routines, an interface routine of the Container Manager which creates a new container can only be called by modules which have a Container Manager capability in which the appropriate semantic right is set. The caller passes to the container creation routine

– an integer parameter, which is passed on to the constructor for the Co-Module Manager, defining the maximum number of co-modules to be created in the container (to determine the length of the Co-Module Table), and

– a file capability for a Disc Directory Manager, which determines the disc on which the new container is to be placed. If a null parameter is passed this may imply placement on a standard disc (e.g. the node's main system disc).

The container creation routine returns two capabilities to its caller:

– an owner capability for the container and

– a file capability with appropriately reduced rights for its Co-Module Manager.

The kernel's role in this activity is described in chapter 23 section 6.

12.5 The Disc Directory Manager

The capability for the Disc Directory Manager provides evidence that the caller can create a container on the appropriate disc. The container creation routine uses the capability to call the Disc Directory Manager's container creation routine, which allocates a new container number (within disc), enters the new container into its disc directory and allocates to the container a single page on disc which becomes its virtual page 0. This suffices to allow the Co-Module Manager to create a Co-module Table (CMT). If at a later stage page 0 is not in the main memory, the Disc Directory Manager resolves this page fault without reference to the Virtual Page Table Manager for the container. Hence the following steps can be carried out without requiring the intervention of the VPT Manager.

12.6 Installing the Co-Module Manager

The main function of the Co-Module Manager is to create and manage the CMT of the container. When the Container Manager calls the Co-Module Manager's constructor, this initialises the CMT for the container; the latter has a fixed position in page 0, which is known to and used by the kernel.

When the Container Manager's create routine (still executing in the new owner's thread) invokes routine 0 (the constructor) of the Co-Module Manager code capability, the latter gains access to page 0 of the new container, as described above. At this stage there is no physical memory behind virtual page 0. Hence the first attempt to access page 0 will cause a page fault interrupt, which can be handled by the Disc Directory Manager module without reference to a (still non-existent) Virtual Page Table Manager.

Thereafter the Co-Module Manager can access the page to create and set up the Co-Module Table (CMT). It creates a reference to its own file data (entirely contained in page 0) in the first entry (i.e. entry 0), such that the "Pointer to State Data Segment" field actually points to its own root segment[31].

It places a copy of its own code module capability in the appropriate field in the CMT of entry 0 (the "Modcap for Code Module" field). It can obtain this in a parameter to its constructor, since the Container Manager already has this capability. The semantic right to call routine 0 is removed from this capability, since the constructor should not be called a second time. Thereafter it is possible for other modules to call the Co-Module Manager instance for the new container, provided that they have a capability allowing this.

12.7 Installing the Remaining Security Sensitive Modules

At this point normal calls to the Co-Module Manager can take over the task of initialising the new container. At least the following additional security-sensitive modules need to be installed:

– a Segment Manager, and

– a Virtual Page Table Manager.

Further co-modules which may need to be created include a debugger module, a Code Manager, a Thread Manager and a Thread Control Manager, depending on the purpose of the container.

As these are all security-sensitive co-modules related to kernel activity, they must also be pre-approved modules. In order to install the required modules, a new container owner can make a selection from a pre-existing list of code capabilities (held in the Co-Module Manager's constant segments), using the `createSecureMod` routine. This returns a capability for the newly created module, which is typically not the owner capability, but a capability with reduced rights corresponding to the actions which normal users can invoke. The owner

[31] Espenlaub assumes that the CMT is the root segment, but this is not actually necessary, provided that the CMT is located where the kernel expects to find it.

capability is either retained by the Co-Module Manager or deposited in a special directory holding owner capabilities for all the security-sensitive modules associated with the container).

A `modType` parameter indicates which type of secure module is to be created (e.g. Segment Manager, Virtual Page Table Manager, debugger, etc.), and a `selector` parameter indicates which of the available alternatives for this type of module is to be installed.

For each relevant new co-module, the Co-Module Manager activates routine 0 (the constructor) of the corresponding secure code module, which in appropriate cases executes the kernel's `load_page0` instruction to gain access to page 0 of the container, using a similar pattern to that described above.

12.8 Creating Normal Modules

It is the Co-Module Manager's responsibility to ensure that the security sensitive modules required in the container are installed before its owner can create normal co-modules in the container.

In this case a `modType` parameter indicates whether the new module is a normal file module, a cooperating co-module or a library module which requires a persistent root.

Converting a normal data file into a code module is not an activity of the Co-Module Manager but of a separate Code Manager module (which must have been installed as a security sensitive co-module before the compiled data file can be converted into a code module).

13 Copying Containers

The copying of containers in SPEEDOS is a more complex issue than at first meets the eye. There are two reasons for this.

i) A SPEEDOS container does not simply hold data or code, but it also includes security-sensitive co-modules which cannot simply be copied. An obvious example is the Virtual Page Table Manager, which holds disc addresses relevant only to the current container. For this reason there are always two phases in copying a container. The first is to create a new file (for the copy), as described above in section 12. The second is to copy the required user information into this container. Relevant security related co-module information, in particular the Virtual Page Table Manager information, must be created anew for the copied module.

ii) The second reason for the complexity is that different operations must be provided for different purposes. For example a user's aim might be

– to make a temporary backup of changes as they are being made (e.g. when editing files), such that the file retains the same identifier, i.e. can continue to be accessed by the same capability).

– to make a copy which the owner, or some other user, can use independently of the original.

– to archive the container for use as a backup in case of failure in the original.

– to distribute software to purchasers.

– to replicate the container for use in parallel on different nodes at geographically distant locations (e.g. to reduce internet traffic).

– to replicate the container for immediate use on a backup machine when the main machine encounters a problem requiring it to close down.

Such different purposes can lead to copy operations acting differently, especially with respect to the handling of capabilities embedded in the container to be copied. Not all such copy operations need to be provided on every SPEEDOS system or for every container on the same system. The first three aims are likely to be needed on a regular basis for many containers on many systems and implementations of these are therefore now briefly discussed. Before doing so we discuss some fundamental issues relating to the copying of containers. Discussion of other aims, e.g. which involve the use of a network of computers, is postponed to later chapters.

13.1 Fundamental Copying Issues in a SPEEDOS Environment

In some systems copying is problematic because of the pointers which may be contained in a file. Some of the problems which these raise in other systems were discussed in volume 1 chapter 9.

In SPEEDOS there are two kinds of pointers

- internal pointers between the segments in a container, and
- capabilities (i.e. external pointers between different files).

Internal pointers in SPEEDOS would not be a problem if an entire container were to be copied page by page. The reason for this is that "short" pointers (i.e. within container addresses) used to provide the cross-references between the different segments in a container are relative pointers within the container.

However, the indiscriminate copying of capabilities can create protection problems. Capabilities are discussed in more detail in chapter 26 but at this stage it is essential to know that

a) only one *owner* capability can exist for each container and for each module within a container;

b) other capabilities can include restrictions which do not allow the capability to be copied.

To assist in controlling such situations, the Segment Manager provides a semantic routine which indicates how many owner capabilities exist in the user module segments of a container and how many other restricted capabilities exist. It also provides a semantic routine which locates them.

13.2 Temporary Backup of a Container (e.g. while Editing a File)

In conventional systems it is normal practice for text editors and similar programs to create a second copy of a file or to record changes, thus allowing the user the option to restore the original file content if he so decides when bringing his editing session to completion. Such programs often also have a facility which allows the programmer to return to the last state of the file recorded in the file system.

This raises two issues for SPEEDOS systems. First, because the virtual memory is persistent and no separate file system is provided, everything is automatically "saved" by the virtual memory system. Second, if the file is simply copied it will have a new identity and consequently capabilities for the file held by other users would be implicitly revoked. There are several ways to handle these issues. Here is just one of them.

When the editor is activated, it copies the file's segments and thereafter makes the user's modifications to this copy. But the copying of the existing file does not entail creating a new file. Instead both "files" are held together in the same co-module. This is achieved by the editor, when first creating the file, by attaching to the root node two new segments. The first of these is then actually used as if it were the root node. Then when the file has to be edited, it first makes a fully copy of the existing file to the second node and edits this version.

The user can subsequently use an editor-supplied "save" command, which causes the editor to write the second "root" to the main "root", leaving the file in the edited state. Whether it deletes the earlier version of the file or simply leaves it as a backup, thus allowing the user to revert to the unedited version if he wishes, is a decision for the editor design.

This solution also suggests how previous versions can be maintained. The persistent secondary root simply needs to maintain a series of pointers allowing access to previous versions of the file. At the start of an editing session the editor can then allow the user to select which version he wishes to open.

The approach described above has the advantage that a single file container is used throughout, and no new capabilities need be created and distributed.

No attempt has been made to optimise this solution, since this lies within the realms of normal application programming, and no additional system functionality is required.

13.3 "Simple" File-to-File Copying

There are two fundamentally different ways of approaching the task of copying a container or an individual file (which is addressed via a new capability). The simplest (but not necessarily the most efficient) way is to create a new container and then use a loop, which continuously reads a record (or other logical unit) from the file to be copied (via its semantic routines) and writes this to the new container which it has created, until the file has been completely copied. This requires no special precautions in the very simple case and all the usual safeguards for protecting files apply.

But what is the "simple case"? The following conditions must apply:

i) The file to be copied either (a) contains no pointers, or (b) where internal pointers are used these should be hidden via semantic routines.

ii) The copying program must either contain no owner capabilities or non-copyable capabilities or the program must know how these are to be handled.

Condition i)(a) will only be fulfilled in the simplest of cases (e.g. a small unstructured file, or one where the internal structure does not use pointers). However, there are many cases where condition i)(b) can be fulfilled (e.g. where the file is a sequential file with a semantic routine "get next" (and a corresponding "put next"), whereby internally the records may (or may not) be linked by pointers.

Condition ii) will often be met. The best way of course will be for the user to avoid such problems entirely by ensuring that all capabilities in the file are copyable[32]. While this is a secure way of copying files, it is not the most efficient, and it will not easily work for all kinds of file. For example long video files cannot usually be broken down into "records", and although they will rarely contain capabilities, using this method involves copying the file three times, i.e. once into the parameters for the copy routine, once from its input parameters to its output parameters and once into the final file! Clearly an alternative technique is needed.

13.4 Page by Page Copying

Such an operation cannot be trusted to unprivileged user level software since it implies direct access to the page tables of the container. This requires that the

[32] In a later part of the book, in a discussion of user-level activities, it will be suggested that users should in any case organise their capabilities in such a way that normal copying of most files will avoid these capability issues.

Virtual Page Manager's pages (including the page tables) can be distinguished from the rest of the container's content, and will also be dependent on the capabilities contained in the container; this implies that the Segment Manager must also be involved in the operation. For these reasons such a copy operation is a privileged operation, and must be provided by the system. Hence the Container Manager provides an operation, which gives the user a choice of actions to be taken with respect to "problematic" capabilities in the container. This method is explained in more detail in chapter 23 section 7.

13.5 N-ary Copying of a Container

N-ary operations are operations involving two or more files, often of the same type (see chapter 18 section 9). In SPEEDOS these are handled as operations in which files passed as capability parameters can be accessed via their root segment. The capability for an n-ary parameter, known as a free capability, must have a special access right set which allows this (see chapter 26). Further restrictions are that n-ary parameters allow only read access to the file data and that a module may not be used concurrently as a free capability and via a normal inter-module call, unless the inter-module call claims *read access* only to the file. For this purpose the status word in the Co-Module table (see Figure 19.6 above) holds a count of readers.

This copying method allows users to avoid the rule which forbids the passing of pointers between modules, but the rules regarding owner capabilities and non-copyable capabilities are strictly followed. This method is suitable, for example, for copying video files (and thereby sidesteps the need to copy such a file three times). But it is not reserved for video files; it can be used also for other files, e.g. those with complex internal segment structures, but also for simple files.

The actual copy procedure requires that a new file first be created as described in section 12 then when this is activated it calls the kernel instruction `load_free_cap` (see chapter 18 section 8) in order to gain access to the data of the file to be copied. If it contains "forbidden" capabilities then the operation will fail unless the copy operation (which must be a semantic routine of the file type being copied) knows how to deal with them.

13.6 Archival Backup

The issues involved here include recovery of an entire system after failure, which is a quite complex issue that cannot be considered without also considering other issues such as system start-up. A discussion of system backups is therefore postponed to a later chapter.

13.7 Other "Copying" Requirements

The remaining themes mentioned above (distribution of software to purchasers, replicating containers for use in parallel on different nodes at geographically distant locations and replicating a container for immediate use on a backup machine) are also advanced issues which cannot be dealt with at this point.

14 Converting Modules to a Different Format

The conversion of user data from one format to another can of course be undertaken internally, provided that the related code manager is also designed to cope with the new format, but the more challenging issue arises when the result of a conversion is to be accompanied by a change in the code manager.

In such cases the actual conversion operation requires a code manager which understands both the old and the new formats for the data. The best way to achieve this is to create an empty container for the new format and pass to its conversion routine a free capability for the file to be converted. Alternatively if the new module is to be placed in the same container as the old module, the file root for the old module could be passed as a parameter to a CMC call (see chapter 17 section 7), assuming that the code module also supports this facility.

15 Deleting Containers

The container manager provides a basic deletion interface routine which deletes an entire container. Its first parameter is a capability for the container to be deleted, which must include the generic right *delete*. This routine only deletes the container if the latter has no active co-modules.

A further parameter to the container manager's delete routine indicates whether deletion is to proceed if it contains owner capabilities. This is significant because if an owner capability is deleted, the basic SPEEDOS system provides no further way of controlling the container (including deleting it). However, individual systems could add a further routine in the container manager which manufactures and returns a new owner capability to replace those in the deleted module, returning this to the caller. In this case it is important that the extension (a) checks that the container still exists, (b) checks that the owner of the calling thread is in fact the owner of the container, (c) notes in the appropriate containers details indicating when the new capability was created and (d) provides a further interface routine providing the owner of the container with access to this information, to ensure that the interface has not been misused.

If a user attempts to use a capability for a deleted container or any item in it, this results in an error (signalled as a result of a failure to resolve container identifier in the capability).

Since the main purpose of a container is to hold a single application module and its related co-modules the system does not provide a mechanism for deleting a single module within a container. However, such a service could be added by extending the co-module manager and related modules.

16 Renaming Containers and Modules

One simple but drastic way of revoking capabilities is to have them renamed (i.e. provide them with a new virtual memory identifier). This automatically makes capabilities containing the old identifier unusable. From the viewpoint of a user owning such capabilities, it is as if the container or modules within it had been deleted.

To rename a container basically involves the Container Manager in invoking the disc manager of the disc on which the container resides, in conjunction with the Virtual Page Manager of the container concerned to allocate a new "Container # in Disc", then returning a new owner capability to the caller. As in the case of deleting a container, the activity can only be carried out if the container is inactive. No facility is provided to rename individual modules.

Notice that renaming, although it involves issuing a new container number, does not mean that the entire data must be moved. Instead it can be implemented by indirection, i.e. the disc directory entry for the new container number can be made to refer to the existing container and the disc directory entry for the old container number can be invalidated.

17 Changing the Ownership of a Container

One of the identification fields of a container (see Figure 19.2 'Container number of current owner') identifies the current owner of the container. The Container Manager provides a semantic routine `change_owner`, which normally allows the field to be changed. The instruction can only be carried out under the following circumstances.

a) The caller of the routine is the current owner.

b) The new owner is verified to exist.

If these checks are positive the container manager replaces the old entry with that of the new owner name in the appropriate identification field. The `change_owner` routine has two parameters, a capability for the container and a capability for the original container of the new owner, used to create him as a new user. The capabilities must contain a valid `change_owner` metaright. There is a protected kernel instruction which the container manager can use to effect the change.

Chapter 20
Managing User Processes

The management of processes and the scheduling of their threads (the active components of computer systems), are key aspects of a security kernel. The general principles governing these activities, including basic scheduling algorithms and synchronisation techniques, were discussed in volume 1 chapter 8, while chapter 15 outlined the role played by processes in the general model on which the SPEEDOS design is based. In this chapter we consider in more detail how the SPEEDOS kernel manages user processes and organises the calls which a thread makes as it moves from one module to another. Chapter 21 discusses how the activity of synchronising using threads is organised; in chapter 22 the scheduling of user and kernel threads is described.

1 Processes and Threads

Parallel computation (sometimes known as multiprogramming, multitasking or multithreading) is a feature of almost all modern computer systems. One of its aims is to ensure that the CPU is used to best advantage in parallel with the much slower disc and other input/output devices. For example if a thread[33] is executing on a CPU and this initiates a disc access (either explicitly to the file system in conventional systems or implicitly as a result of a page fault in SPEEDOS or in conventional systems), rather than let the CPU lie idle until the disc access has completed (which could result in the loss of millions of CPU instruction executions), another thread is selected to use the CPU. The decision about which SPEEDOS thread will run next is the job of the *User Thread Scheduler*, the role of which is discussed in chapter 22. The important thing about parallel computation is that it is a technique which attempts to optimise the use of the computer's hardware resources (CPU, discs, I/O devices).

[33] A thread was often called a process in earlier literature and research papers on operating system design (including the MONADS system).

A user process in SPEEDOS can be decomposed into multiple threads. The intention is to allow such optimisation to take place within what the user sees as a single computation. However the way this might happen is not determined by the system; the user decomposes his task into multiple threads as is most convenient for him. This is not unusual in modern operating systems, but the way it is organised in SPEEDOS is somewhat unconventional.

In the operating system literature a thread is usually considered to be a *lightweight process* because parts of its process state are shared with other threads of the same application, thus allowing scheduling switches between threads of the same application to be optimised. This is *not* how threads in SPEEDOS are defined. While they may indeed cooperate to achieve a single application aim, in the SPEEDOS architecture they are not restricted to being executed in a single module but can invoke other modules independently of each other, as part of the in-process model.

2 Process Containers and Co-Modules

A process, like other major entities in SPEEDOS, is created in a container, which already holds a Co-Module Manager, a Segment Manager and a Virtual Page Table Manager co-module. What makes the container into a process container is that two further co-modules, a *Thread Manager* and a *Thread Control Manager*[34], are also loaded into it. The purpose of the Thread Manager is to create and manage one or more threads comprising the process; the Thread Control Manager is responsible for organising their dynamic execution, in association with the User Thread Scheduler.

There is no defined limit on the number of process containers which may be owned by a single user. But since each container always has a single owner, all the threads of a particular process belong to the same user. A process container can have up to 255 threads (and subthreads), which are distinguished in thread capabilities by their index number (see Figure 19.10).

3 Thread States

At any point in its execution a thread has a *state*, which progresses with each instruction that it executes. From the viewpoint of the User Thread Scheduler, the state of a particular thread is defined primarily in terms of the values in its CPU registers (e.g. the general purpose and the segment registers) and in its kernel's pseudo-registers (e.g. its container registers) while it is executing.

When the User Thread Scheduler decides that a thread should no longer ex-

[34] These modules, which appear in process containers, should not be confused with the User Thread Scheduler mentioned earlier.

ecute on the CPU, the state of its CPU registers and pseudo-registers must be saved for restoration when it later continues, and the state of the newly selected thread must be loaded into the CPU registers from the memory locations where they were saved. This is part of the activity of thread scheduling, which is discussed in more detail in the chapter 22.

4 The Thread Manager Co-Module

A Thread Manager co-module is created in a new process container as a result of some other thread invoking the constructor of the Co-Module Manager in the new container[35]. When the constructor of the Thread Manager is invoked, this creates a *Thread Table* (Figure 20.1), which is managed by the Thread Manager and is used by the core kernel, which has a pointer to the thread table in page 0 of the container.

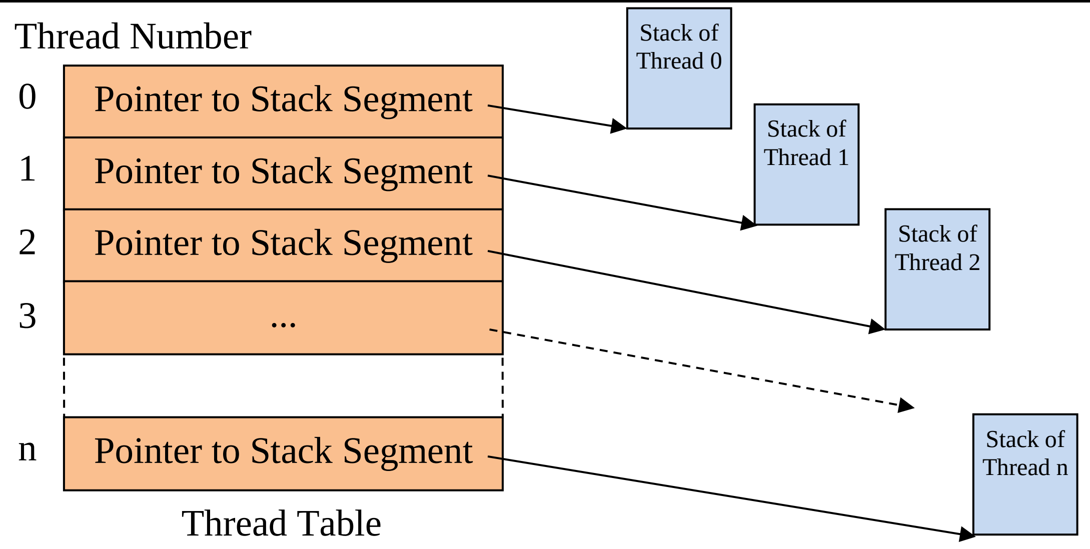


Figure 20.1: A Thread Table in a Process Container

A Thread Table consists of an array of pointers to the information describing each individual thread of the process[36]. The segment referenced by a thread table pointer, called a *Thread Stack*, holds the current state of the corresponding thread (its current register values when it is inactive, the addressing environment of the current module in which it is/was active) and its inter-module linkage (Figure 20.2). The thread number in a thread capability (see Figure 19.7) serves as an index into the Thread Table.

[35] When a Co-Module Manager creates a Thread Manager instance it prepares a thread capability for the latter which confers the rights to create and delete subthreads (and, as we shall shortly see, to make limited calls to the Thread Control Manager). It passes the thread capability to the kernel and the latter places it in the red-tape area of the container, whence modules with the appropriate permissions set can obtain a copy (see chapter 19 section 5).

[36] In reality it is a segment containing a pointer for each thread stack in the pointer section and an indication of which threads exist in the data section.

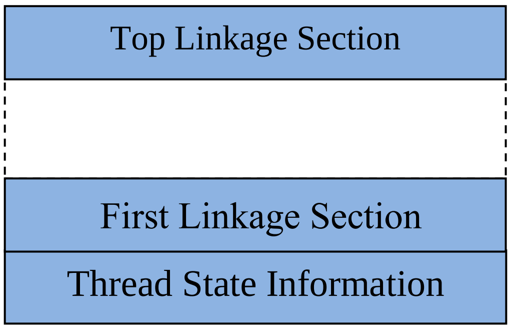


Figure 20.2: A Thread Stack

The Thread Manager offers, inter alia, an interface routine **`createThread`**, which expects as parameters a capability for a start module, an integer defining the semantic routine to be called, a string defining a name by which the thread is to be known and a capability which defines its root module. It creates a kernel stack for the thread, placing a 'thread backstop' at the bottom of the stack and prepares it to make its first inter-module call (including a parameter segment which provides access to the root module).

This routine invokes the Thread Control Manager to advise the existence of the thread and to activate it before it calls the User Thread Scheduler, which then calls the kernel (see section 8.2).

4.1 Thread Stacks

Each thread stack is a single segment and the information in it is held in the data section of the segment. Although this holds much security-sensitive data which would normally be held in separate segments (e.g. each linkage section could in principle be a separate segment, with pointers to parameter segments, etc.), this would be quite inefficient for the kernel to manage, and so it is simply treated as data by the kernel. This is not a security risk, since only the kernel can address information on a thread stack[37].

The kernel maintains two private registers for each thread stack: a Top of Stack Pointer and Current Local Base Pointer. The rest of the thread stack consists of linkage information and parameters relevant to the kernel's role in supporting inter-module and similar calls.

Chapter 31 describes in more detail how users and their processes, threads and subthreads are created.

[37] As we shall see in section 6, the kernel provides user threads with access to parameters for inter-module and similar privileged calls via segment registers, which are held on the thread stack.

5 Application Level Multithreading

Normally a user creates one or more main threads in a process container and provides them with a capability or capabilities via which they can make inter-module calls to application modules[38]. It is usually the application module's code (not necessarily the initial module of a thread) which decides how the computation should be further decomposed into parallel subthreads (more equivalent to threads in conventional systems), as often only the programmer of a module knows how to decompose a particular computation into subtasks.

Espenlaub's thesis does not discuss how a module can create parallel subthreads dynamically from within the code of the module itself. In order to do that, a semantic routine of the Thread Manager for the active thread must be called, and this requires a module capability for the appropriate Thread Manager. The difficulty is that threads of different processes which call the module each have a different Thread Manager instance, so that different capabilities are potentially required for different threads.

To make such capabilities accessible to application modules, the Co-Module Manager, as part of its Thread Manager creation activity, passes a capability for this to the kernel, which places it at the base of the container holding the process's Thread Manager. This can be made available to application modules as described in chapter 19 section 5, using the kernel instruction `get_subthread_cap` , which returns a capability for the Thread Manager to the requesting module, in which only the semantic right `create_subthread` is set, provided that the corresponding permission is set in the Thread Security Register (see chapter 26).

To create a subthread the application module can then invoke the Thread Manager's interface routine **`createSubthread`**. This routine expects as a parameter the number of an entry point in the Subthread Entry Point list (see chapter 19 section 9.5) for the currently active module. The Thread Manager then creates a thread stack for the new subthread, places a 'subthread' backstop on the stack and sets up a thread state for the subthread. It then calls the Thread Control Manager, which in turn calls the User Thread Scheduler to schedule the subthread.

Subthreads, like main threads, can have local call stacks, make inter-module calls, etc. If subthreads create further subthreads the rules above are applied as if the creating subthread were the main thread. Otherwise no special relationships are defined between main threads and subthreads, thus allowing different compilers to adopt different strategies. Any special rules concerning sub-

[38] This will be more fully described in chapter 31.

threads (e.g. that the thread which creates a subthread must be informed when a subthread is deleted, or that subthreads must be automatically deleted if their creating thread is deleted), are the responsibility of the application module. Extensions to the Thread Control Manager might be added in Timor to assist in this.

Creating threads and subthreads is described in more detail in chapter 31 section 2.

6 Parameter Passing Strategy

The RISC movement developed various strategies regarding the passing of parameters on procedure calls. However these strategies were concerned with the optimisation of procedure calls within a single program and were therefore not motivated by security concerns, unlike the SPEEDOS protected calls. Furthermore the RISC movement did not base protection on segment registers. Consequently the RISC techniques do not provide an adequate model which can simply be adopted for SPEEDOS calls.

6.1 Espenlaub's Attempt to Adapt a RISC Strategy

In Section 6.6.2 of his thesis Espenlaub defines a strategy for parameter passing and register management for inter-module calls in SPEEDOS based as far as possible on RISC ideas. This allows general purpose registers to be used for passing normal data values, and in theory for segments to be passed via segment registers, and for module capabilities to be passed via "module capability registers". However, the uncontrolled passing of valid segment registers (which in Espenlaub's design have full 256-bit virtual addresses) presents a potential threat to the strategy of ensuring that pointers do not escape from a module (as this would make garbage collection – and synchronisation of data accesses – impossible). Espenlaub goes on to say that "it is explicitly permitted by the SPEEDOS kernel design to omit the module capability registers completely and reference module capabilities with memory operands" [4, p. 167]. In the new SPEEDOS design module capability registers are *not* supported, as this would add considerable cost to a hardware design without offering commensurate benefits.

Espenlaub continues that "since the number of registers is in some cases not sufficient to pass the parameters and/or return values, it is also possible to pass exactly one segment register to the called module and/or to the calling module". In the new SPEEDOS strategy (which for example includes the decision to use addresses with SCIDs rather than full 256 bit virtual addresses) a variant of this becomes the rule rather than an option for inter-module and other kernel organised calls. Furthermore, the segments addressed are held on the thread stack, as will be described below. Consequently the difficulties mentioned above no longer exist. Furthermore this approach solves a potential problem which Espenlaub does not discuss, viz. that under certain circumstances – to be discussed in chapter 26 – bracket routines (which were introduced in chapter 13) may need to examine parameters[39]. If these can be passed in arbitrary registers it would be impossible to implement bracket routines.

6.2 The New SPEEDOS Strategy

There is only one way of passing all parameters for an IMC and other secure call instructions, i.e. via a stack-based "parameter segment", addressed by a dedicated segment register controlled by the kernel. (A thread's call stack, it will be recalled, is organised entirely by the kernel as a single segment, but the kernel creates for applications the impression that the parameters are passed in a separate segment accessed via a segment register.)

The access rights in the segment register for the input parameters are set to "read only" access by the kernel as part of the IMC instruction, because return parameters are passed in a separate segment, which is accessible to the caller via a further segment register after the return.

This strategy applies only to the kernel call instructions, and does not prevent compilers from using a RISC strategy for parameter passing between the internal procedure calls of a module.

6.2.1 Parameter Segment Registers

Four segment registers are used for the passing of parameters and cannot be used for any other purpose. They can only be set and invalidated by the kernel. Their uses are shown in Figure 20.3.

Register 0	the module's own input parameters (read-only access)
Register 1	parameters to be returned by the module to its caller (read-write)
Register 2	parameters being prepared for a module which it will call (read-write)
Register 3	the parameters from the last call which it made (read-only).

Figure 20.3: Dedicated Parameter Segment Registers

The following kernel instruction creates two "segments" at the top of the thread stack (with the return segment above the input segment[40]).

```
create_imc_params (int in_data_length, int in_modcap_count,
         int return_data_length, int return_modcap_count)
```

[39] On the other hand Espenlaub discusses the need to *protect* access to parameter lists from unauthorised bracket routines, which could be a threat to security (see Espenlaub section 4.3.2). This issue is discussed in chapter 24.

[40] The reason for this will become clear in chapter 24.

It is executed by a calling module in preparation for making an inter-module (or similar) call. The `in_data` and `in_modcap` items describe those parameters which a caller wishes to pass to the called module, while the `return_data` and `return_modcap` parameters define those which it expects to receive on the return.

The instruction is implemented directly by the kernel without reference to a Segment Manager. The kernel creates both segments on the current thread stack (with the return segment below the input segment) and initially loads Segment Register 2 to access the segment intended for the module to be called, with "read-write" access rights. The instruction invalidates Segment Register 3, but as part of the `return` instruction the latter is set to address the return segment (with "read-only" access). On return to the module Segment Register 2 is invalidated by the kernel.

If parameter segments already exist when this instruction is executed, these are deleted and a new pair of segments is created at the top of the kernel's thread stack.

When a calling module has set up its parameters it can execute the kernel's `inter-module call` instruction. The called module can read its parameters via Segment Register 0 (which the kernel sets to read-only mode) and can prepare its return values via Segment Register 1 (which the kernel sets to read-write mode).

Segment Registers 0 to 3 cannot be stored into the pointer partition of other segments, nor can they be copied into other segment registers. This check is carried out in the kernel `store_pointer` instruction, since the segment registers used to hold parameters have the "can be stored" access right unset.

6.2.2 Restrictions on Parameters for Inter-Module Calls and Returns

It has already been emphasized several times that pointer parameters may not be passed on inter-module calls and returns. But that is not the only restriction. More complex parameters (e.g. objects of user defined types in object oriented languages) should not be passed (also not by value), as this contravenes the information hiding principle. Although this cannot be fully controlled (e.g. if the programmer reverts to the use of integers as pointers) such parameters (between modules) are unnecessary since they are better handled by using library routines (see chapter 18 section 6.4).

Conventional programming languages are usually designed in such a way that they support only one return value. This restriction does not exist in the SPEEDOS architecture.

In general the standard types supported by higher level programming languages have a fixed length, the only common exception being character strings. When these are passed as parameters to another module, the caller generally knows the length and may need to indicate this (if the instruction set does not take care of this situation) in the data partition of the input parameters. Similarly programming languages must take care of the character strings for return parameters. In this case the caller determines the size of return parameters and may have to make allowance for a string of maximum length to be returned.

6.3 Library Calls and Co-Module Calls

The question arises whether parameters passed to library routines should be handled in a similar manner to those of inter-module calls. One difference is that there is no obvious problem in passing segments as parameters since these are held in the same container for client and library routine. This might suggest that the compiler should handle the parameters internally. A similar argument might be used for co-module calls.

However, in both cases the code is switched to a new code module, which might define that a qualifier list (i.e. bracket routines) be used in association with the code execution[41]. Qualifiers in this list may need to examine the parameters (especially the outgoing parameters) or might even totally prevent outgoing calls for security reasons. To make this possible the parameters must be available to the kernel. Consequently a modified form of the inter-module parameter passing technique is applied, the only difference being that segment pointers may also be passed as parameters. The corresponding kernel instruction is as follows:

```
create_pc_params (int in_data_length, int in_pointer_count,
     int in_modcap_count, int return_data_length,
     int return_pointer_count, int return_modcap_count)
```

(The abbreviation *pc* in the name refers to a *pointer call*, i.e. a library call or a co-module call.) When it receives a request to create parameters the kernel notes which kind of call to expect and raises an error if `create_pc_params` is followed by an inter-module call or if `create_imc_params` is followed by a pointer call.

7 Storing/Restoring Registers on Calls

Decisions regarding the conventions for storing and loading registers as part of the execution of procedure calls can dramatically affect the efficiency of systems. But in this respect some RISC strategies which were developed regarding the saving of registers (e.g. the SPARC register window mechanism [10, pp. 2-3

[41] Qualifiers, their lists and their bracket routines were introduced in chapter 13 and are discussed in detail in chapter 24.

– 2-6]) cannot economically be applied directly to the storing of registers for the security-sensitive SPEEDOS calls. The strategy which can be used, at least in part, is that the system should not save registers on calls, since the compiler/application has a better knowledge of the registers which it needs after a return and therefore the number to be stored can be minimised if this is the compiler's responsibility. The following rules apply *mutatis mutandis* to all kinds of kernel calls.

7.1 Segment Registers

The most costly registers to store in SPEEDOS are the segment registers, which because of their protection function are considerably larger than the general purpose registers used for addressing in normal RISC architectures. It is therefore appropriate to adopt an efficient strategy, but one which does not endanger the security of the system and its applications.

The simplest strategy would be to require the calling module to store its own segment registers before initiating the call; in this case at least one segment register would have to be loaded by the kernel on the return to enable it to address the location at which they were stored. However this strategy creates a problem, because it would require the application thread to be able to store parameter segment registers and those loaded from free capabilities, which the application thread is normally not allowed to store, since they address data in containers other than that of the current data container.

The strategy suggested by Espenlaub [4, p. 168 paragraph 1] is that the kernel stores all the segment registers (except one) in the linkage. This works correctly, but is inefficient if at the point of the call (or after the return) the application thread does not use all the registers.

The following convention is therefore used.

a) The kernel, which is the only software that can load and invalidate segment registers, maintains a bit list indicating which segment registers are currently valid.

b) When the kernel executes a call it saves in the linkage the currently valid registers, and it also saves the valid bit list.

c) When the thread executes a return back to the caller, it restores those segment registers, invalidating the rest.

Notice that when the kernel stores segment registers, it stores their entire contents, i.e. it does not use its `store_pointer` code, because this in effect stores a register as a single word pointer and uses the red tape at the pointer destination to recover the remaining information.

When the kernel reloads the segment registers after a return, it must ensure that each referenced segment still exists; otherwise malicious hackers could use this as a security loophole. The reason why a segment might no longer exist is that another thread may be (or may have been) active in the calling module and have deleted the segment (either deliberately or as a result of erroneous synchronisation operations). To prevent this potential security loophole the Segment Manager provides a unique identifier for each segment which it creates in a container in a protected area not accessible to applications. When the kernel stores a segment register it notes this identifier in the linkage, and on reloading it checks that the segment register actually addresses a segment correctly and that this has the correct identifier.

7.2 General Purpose Registers

The compiler knows better than the kernel which general purpose register values should be saved on kernel supported calls, and is responsible for saving and reloading them.

The question arises whether the kernel should also invalidate these registers, which would be an unusual step. However, if security is taken very seriously, this is necessary, not least because they could otherwise be used for secretly passing information to the called module, i.e. they offer an easy to use covert channel for passing a substantial amount of information to the module which they are calling and conversely back to their caller on a return (see chapter 3). The only way to avoid this is by also invalidating the general purpose registers.

But thereby lies a further problem: usually general purpose registers do not have a valid bit. There are two possible solutions for this.

i) A bit list could be implemented in hardware, whereby each bit represents a general purpose register by position. It would be a fast operation for the kernel simply to unset all the bits as part of a call. (A bit list held only by the kernel is not feasible, since the kernel is not aware of individual loads and stores on general purpose registers.)

   However, each time a general purpose register is then loaded, the hardware would have to set the corresponding bit, and each time it is read, the hardware would have to check whether the bit is set (and if not raise an interrupt).

ii) Alternatively, a perhaps less efficient solution, but one which is simpler to implement, is for the kernel to write a standard value to each such register on inter-module calls and returns (e.g. by zeroing each register).

The decision between these depends on the hardware design. Notice that this is not necessary for library calls since a library module can only get information out of a module via an inter-module call or a co-module call.

7.3 Floating Point Registers

If the system has additional register for floating point arithmetic, these can be managed like the general purpose registers. In general we have ignored the existence of floating point registers, since these can be viewed simply as optional extra hardware for improving speed, which may affect performance but not the security of a system.

7.4 Code Registers

The code segment register (which addresses a code segment) and the program counter (which is an offset in the code segment) must in any case be stored by the kernel to allow the code of the calling module to be resumed on the return.

8 Kernel Call Instructions

Having established the general conventions and rules associated with inter-module and related calls, we can now describe the actual calls and returns in more detail.

8.1 Inter-Module Calls

Before a module calls another module, it will typically prepare for the call by calling the kernel's `create_imc_params` instruction (see section 6.2.1) and then using SR2 to prepare the parameters for the call. It might also invalidate those currently valid segment registers which it does not need after the call, to make the return from the IMC faster. The instruction also provides space for return parameters, which the kernel makes addressable by SR3, which is then invalidated. (It is made valid in read-only mode by the inter-module return instruction exiting back to the module.)

It then passes three operands to the kernel's IMC instruction:

a) the module capability for the module to be called;

b) an integer indicating the entry point number of the routine to be called;

c) a boolean parameter indicating whether the caller is requesting read-only or read-write access to the module's file data (used for synchronising with free capability use of the file).

After storing the linkage segment on the stack, these parameters are stored on the stack in an IMC stack record, which allows the further progress of the thread to be recorded. For a simple call this is useful for debugging and for recording whether the page 0s for the data and code containers have already been locked down (see chapter 23 section 4.4), but in more complicated cases (e.g. in IMCs involving brackets or remote calls[42]) it plays a more significant role.

When the IMC has been executed, the called module initially needs access to the following segments:

i) the root data segment of its own persistent data (in its own co-module data container). The kernel finds and locates this from the Co-Module Table and makes it accessible to the called module via SR5.

ii) the incoming parameter segment, as discussed above. The kernel makes this accessible to the called module in SR0 and sets the content to read only access (see above).

iii) the segment via which it can return parameters to its caller, which is made addressable in SR1 in read-write mode.

All the general purpose registers and the segment registers 2 and higher (except 5) are invalidated by the kernel, thus ensuring that access to the data segments of the caller are not possible.

If a caller requests read-only access the access rights field in segment register 5 is set to read only. Hence all further segment registers loaded to access a file segment are also set to read-only.

A module capability needed for gaining access to free capability parameters (see chapter 17) can be passed as an input parameter in the module capability partition of the caller's input parameters. However, the kernel does not automatically load a segment register for this. Instead, it provides an instruction `load_file_root`, which takes as parameters the file capability passed and the number of the segment register which the n-ary routine chooses to use to address the file's segments. The segment register is always set to read-only access. If the module capability does not have free capability access then the instruction generates a security error.

8.2 New Thread Calls

Starting a new (first level) thread is a little tricky, because in the design proposed a new thread should start executing in a new module, but there is no module which can make a normal inter-module call to the first module! To avoid complicated solutions at the operating system level, we simply introduce a `new_thread` kernel instruction. This is a privileged instruction which can only be called via a protected kernel capability. Its only additional parameter is a thread capability for the new thread, thus allowing the kernel to locate the new thread's pre-prepared stack[43]. It carries out the minimal necessary to activate a

[42] For bracket routine implementations see chapter 24, for remote IMCs see chapter 27.

[43] Setting up a new thread is an operating system activity, see chapter 31 section 2.3.

new thread. It presupposes (and checks) that a thread stack already exists which has been pre-prepared to make an inter-module call (including parameters for the call). It checks that a 'thread' backstop marker has been placed at the base of the stack (to stop it attempting to return back below the beginning of the stack). When all the checks have been satisfactorily completed it activates the thread in its first module, after invalidating the segment registers except the input parameter register.

A similar problem arises with the creation of subthreads, but since these are activated dynamically in the module which needs them, the mechanism is different. A subthread is activated in a routine of the active module via the kernel call `new_subthread` in a routine of the module's Subthread Entry Point List (see chapter 19 section 9.5). The parameters for the kernel instruction are an entry point number in the module's Subthread EPL, the unique module number of the module in which the subthread is to be activated and a thread capability for the new subthread. As in the `new_thread` case the stack has been pre-prepared and all segment registers except the input parameter segment register and in this case the host module's value for segment register 5 are invalidated. The kernel checks that a 'subthread' backstop marker has been placed at the base of the stack (to stop it attempting to return back below the beginning of the stack).

8.3 Library Calls

To call a library module, the client module executes the kernel's `create_pc_params` instruction (see section 6.3) and then uses SR2 to prepare the parameters for the call. It might also invalidate those currently valid segment registers which it does not need after the call, to make the LC faster. The instruction also provides space for return parameters, which the kernel sets to be addressable by SR3, which is then invalidated. (It is made valid in read-only mode by the inter-module return instruction exiting back to the module.)

It then passes the following operands to the kernel's LC instruction:

a) the code capability for the module to be called;

b) an integer indicating the entry point number of the routine to be called;

c) the number of the caller's segment register currently addressing what is to become the library routine's root segment, addressable via Segment Register 5. A value of zero indicates that no root segment is being passed.

NOTE: if the caller wishes to restrict the library routine to read only access to its data, it sets the segment register which it passes (see (c) above) to read only.

Library routines are not separately bracketed (since they can only call other library routines and the segment manager) and they cannot be invoked as a remote call. Nevertheless a record of their operands (an LC record) is stored on the stack since this can be useful for debugging.

When an LC instruction is executed, the called module initially needs access to the following segments:

i) the incoming parameter segment. The kernel makes this accessible to the called module in SR0 and sets the content to read only access (see above).

ii) the segment via which it can return parameters to its caller, which is made addressable in SR1 in read-write mode.

The general purpose registers are left untouched. (The caller can zero these if he chooses, and store the values which he will need on the return in a segment not reachable by the library routine.) The segment registers 2 and higher (except 5) are invalidated by the kernel, thus ensuring that access to the data segments of the caller are not possible, except via the pointers passed in the input parameter.

Free capability parameters cannot be passed as input parameters for a library call. If for example the intention is to carry out n-ary operations on the internal data of the caller, references to the n-ary data can be passed as pointers in the input parameters.

8.4 Co-Module Calls

Before a module calls another co-module in the same container, it will typically prepare for the call by calling the kernel's `create_pc_params` instruction and then using SR2 to prepare the parameters for the call. It might also invalidate those currently valid segment registers which it does not need after the call, to make the CMC faster. The instruction also provides space for return parameters, which the kernel sets to be addressable by SR3, which is then invalidated. (It is made valid in read-only mode by the inter-module return instruction exiting back to the module.)

It then passes three operands to the kernel's CMC instruction:

a) a module capability for the module to be called;

b) an integer indicating the entry point number of the routine to be called:

c) a boolean parameter indicating whether the caller is requesting read-only or read-write access to the module's file data (used for synchronising with free capability use of the file).

These parameters are stored on the thread stack in a CMC record, which allows the further progress of the thread to be recorded. For a simple call this is useful for debugging, but in more complicated cases, e.g. in CMCs involving brackets (see chapter 24) or remote calls (see chapter 27), it plays a more significant role.

When a CMC is made, the called module initially needs access to the following segments:

i) the root data segment of its own persistent data as indicated in the module's Co-Module Table entry, i.e. it does not automatically share access to the caller's root persistent data but has its own persistent root in the same container. The kernel makes the called co-module's root segment accessible via SR5.

ii) the incoming parameter segment. The kernel makes this accessible to the called module in SR0 and sets the content to read only access. This can include short pointer parameters (i.e. offsets within the container) which can be set to read only if appropriate. Notice that the persistent root segment of the caller can (but need not) be passed as a pointer parameter in appropriate cases.

iii) the segment via which it can return parameters to its caller, which is made addressable in SR1 in read-write mode.

All the general purpose registers and the segment registers 2 and higher (except 4) are invalidated by the kernel, thus ensuring that access to the data segments of the caller are not possible except via the pointers which it receives as input.

Free capability parameters cannot be passed as input parameters to a CMC. If for example it carries out n-ary operations on the internal data of the caller, these can be passed as pointers in the input parameters.

There is no defined hierarchy between cooperating co-modules. Each can call the other in so far as it has a module capability which allows this.

8.5 Inter-Module Call-Back Calls

These are special calls (CBC) which allow a module to call back the module which invoked it. Since the rules for passing parameters in a call-back are the same as those for an inter-module call the caller will typically prepare for the call by calling the kernel's `create_imc_params` instruction (see section 6.2.1) and then using SR2 to prepare the parameters for the call. It might also invalidate those currently valid segment registers which it does not need after the call, to make the return from the CBC faster. The instruction also provides space for return parameters, which the kernel makes addressable by SR3, which is then invalidated. (It is made valid in read-only mode by the inter-module return instruction exiting back to the module.)

Call back calls use a separate "call back entrypoint list" to locate the destination routine in the call back module. Two operands are passed to the kernel's CBC instruction:

a) an integer indicating the entry point number in the call back entrypoint list of the call back routine to be called;

b) a boolean parameter indicating whether the caller is requesting read-only or read-write access to the module's file data.

The kernel establishes the destination module of a CBC by examining the linkage stack of the currently active thread to determine which module called it. The relationship between a call-back module and the application module which initially called it (via an IMC) is illustrated in Figure 20.4.

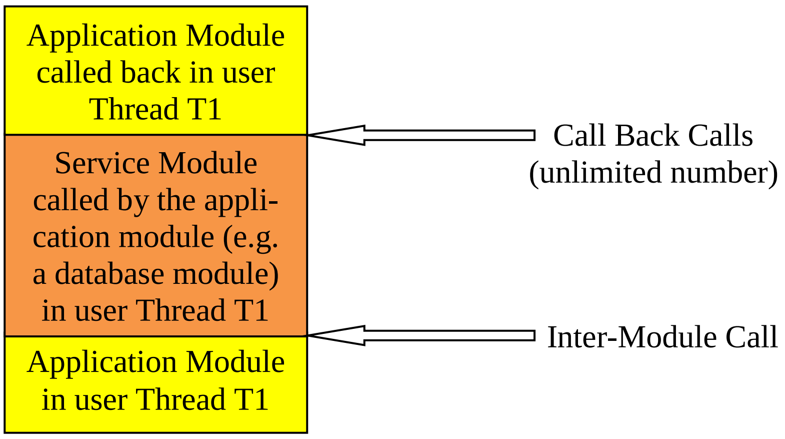

Application Module called back in user Thread T1
Service Module called by the application module (e.g. a database module) in user Thread T1
Application Module in user Thread T1
Call Back Calls (unlimited number)
Inter-Module Call

Figure 20.4: Call Back Modules

A CBC can only be executed if the 'permit callbacks' right is set in the Thread Control Register (see chapter 26 section 4.1). If this permission is not set, the kernel raises a synchronous interrupt.

After storing an appropriate linkage segment on the stack, the parameters are stored on the stack in an IMC stack record, which allows the further progress of the thread to be recorded. For a simple call this is useful for debugging and for recording whether the page 0s for the data and code containers have already been locked down (see chapter 23 section 4.4), but in more complicated cases (e.g. in CBCs involving brackets or remote calls[44]) it plays a more significant role.

When the CBC has been executed, the call back routine initially needs access to the following segments:

i) the root data segment of its own persistent data (in its own co-module data container). The kernel finds and locates this from the Co-Module Table and makes it accessible to the called module via SR5.

ii) the incoming parameter segment, accessible in SR0 which is set to read only access.

iii) the segment via which it can return parameters to the caller of the CBC, addressable in SR1 in read-write mode.

All the general purpose registers and the segment registers 2 and higher (except

[44] For bracket routine implementations see chapter 24, for remote IMCs see chapter 28.

5) are invalidated by the kernel, thus ensuring that access to the data segments of the caller are not possible.

If a caller requests read-only access the access rights field in segment register 5 is set to read only. Hence all further segment registers loaded to access a file segment are also set to read-only.

8.6 Inter-Module and Other Returns

To prepare for a return from IMCs and other calls, the return parameters are passed back via SR1. There is only one `return` instruction (which has no operands); this returns from the highest call currently on the kernel's thread stack. However there is a separate return instruction for bracket routines (see chapter 24.)

On a return a calling module has access to all its previously valid segment registers, except Segment Register 2, which is invalidated. The kernel sets SR3 (the segment register addressing the return parameters from the caller) to read-only access, and deletes the previously called module's input segment, which is no longer needed).

9 Linkage Information Stored on an IMC

The first item in the Inter-Module Call Linkage Segment (see Figure 20.5) is information about the linkage itself, e.g. what kind of call was made.

Linkage Type
Kernel Pseudo-Register Values
Code Segment Register and Program Counter
Parameter Segment Registers 0 – 3
Saved Segment Registers
Bit List of valid Segment Registers
Caller's Local Base on Kernel's Linkage Stack
Previous Top of Stack

Figure 20.5: An IMC Linkage Segment

The kernel's pseudo-registers which are stored in the linkage segment include the Container Registers[45] (except the Container Register for SCID 000 – which identifies the current process container), and further pseudo registers to be described in chapter 26.

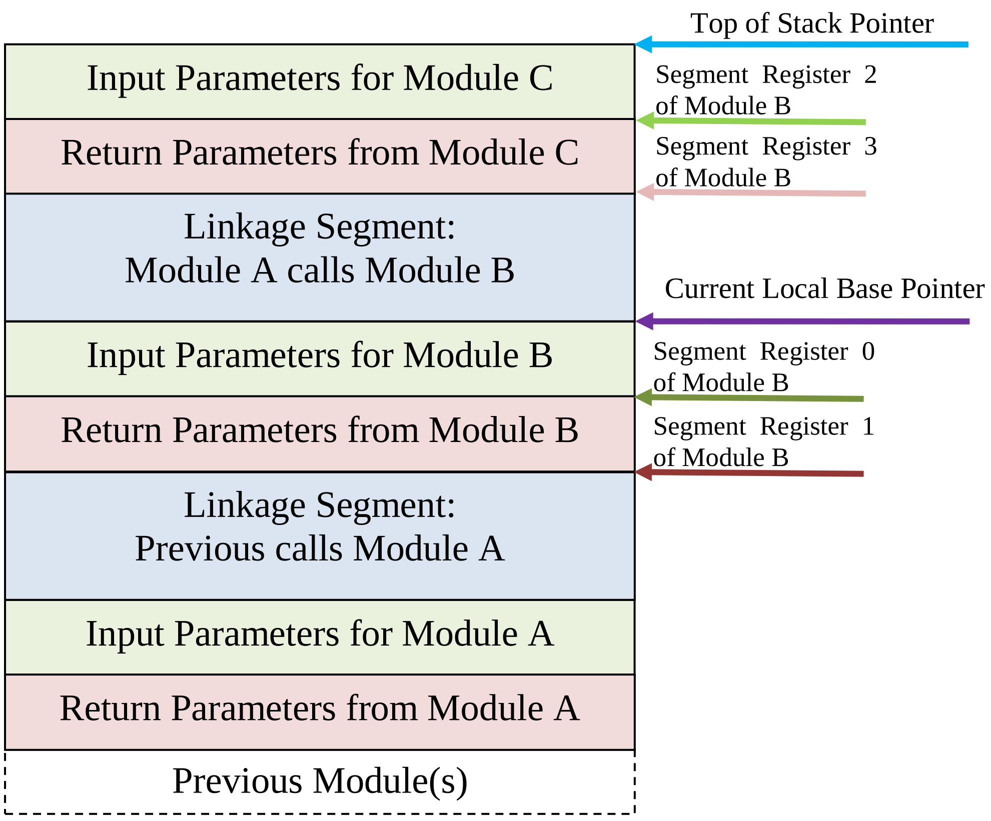
Likewise the current Code Segment Register and the Program Counter are stored, to enable thread execution to return to the next instruction after the IMC

[45] Container Registers were briefly described in volume 1 chapter 16, section 3.3 and are more fully discussed in chapter 23.

when the called module returns. The parameter segment registers and the valid segment registers are stored in the linkage segment because these cannot be stored by the application.

Figure 20.6 shows a kernel thread stack with two modules, where Module A has called Module B and the latter is preparing to call a further module C.

When Module C is called, a further linkage segment is created above its input parameters. When it returns, this linkage and the input parameters for module C are deleted, leaving its return parameters at the stack top. The kernel's own Top of Stack Pointer and Current Local Base Pointer are modified accordingly.



Top of Stack Pointer
Input Parameters for Module C
Segment Register 2 of Module B
Return Parameters from Module C
Segment Register 3 of Module B
Linkage Segment: Module A calls Module B
Current Local Base Pointer
Input Parameters for Module B
Segment Register 0 of Module B
Return Parameters from Module B
Segment Register 1 of Module B
Linkage Segment: Previous calls Module A
Input Parameters for Module A
Return Parameters from Module A
Previous Module(s)

Figure 20.6: An Example of a Thread Call Stack

9.1 Calling Programs

As described in chapter 18, the only difference between a program module and a file module is that the former does not have persistent data, but nevertheless has an associated container in which temporary segments can be created. Hence the only difference in the IMC is that no persistent root segment exists. The kernel therefore invalidates Segment Register 5 on a call. However the module can use this for other purposes.

9.2 Library Calls and Co-Module Calls

Because library calls and co-module calls (which together are called pointer calls) can pass pointers as parameters, the caller must first prepare for the call by invoking the kernel instruction `create_pc_params` to create its parameter segments (see section 6.3).

As with an IMC, the parameters of a pointer call are made addressable to the called module via Segment Register 0. The pointers passed as parameters can be set by the caller to read-only or read-write, and when they are loaded the appropriate access right is set in the corresponding segment register. On execution of a LC or CMC the parameter segment registers are placed on the kernel's call-stack as in the case of an IMC.

Pointer calls are similar to IMCs in the sense that the called module may receive a single root data segment which is set up in Segment Register 5 for the called module. In the case of a CMC the called module its normal root segment is loaded. In the case of an LC the root segment may be provided by its caller and is loaded by the kernel into Segment Register 5, allowing compilers to treat library modules exactly like other modules. (To allow the kernel to set this up the LC includes an integer parameter which indicates the caller's segment register currently addressing what is to become the library routine's root segment. The kernel copies its content to Segment Register 5 for the caller, if it is valid. If not it sets up Segment Register 5 as invalid.)

As in the case of the inter-module call the kernel saves the valid segment registers in the linkage and invalidates the remaining segment registers and the general purpose registers.

Normally the kernel need not store the container register values in a Pointer Call linkage segment. There are however two cases where this may be necessary:

a) If the code container of the called module (e.g. a library module) is not currently addressable via one of the currently valid SCIDs (i.e. SCIDs 001 to 011) then the current value of the container register corresponding to SCID 011 is stored in the linkage segment and is then reloaded by the kernel to allow the new code container needed for the called library module to be addressed. As for an IMC, the segment register value for the current code segment and the current program counter register are stored.

b) If SCIDs higher than 100 are currently in use (i.e. for free capability parameters in the calling module), the corresponding container numbers are stored in the linkage and they are invalidated for the called module.

To return from a PC the application thread executes a kernel `return` instruction, so that here also the compiler sees no difference between a library module and a normal module.

9.3 Inter-Module Call Message Blocks

For each inter-module call the kernel creates an *IMC message block* which serves as a record of the progress of an inter-module call. All the active IMC message blocks for an active user thread are linked together. Each contains a summary of the call parameters (see section 8.1 above) together with information acquired from the Co-Module and Call Table entries for the call and further information generated as a result of the call. The call block for an IMC is deleted when a return from the call is made. These blocks allow kernel processes to communicate with each other, as will become clearer when we describe the mechanism for executing bracket routines in chapter 26 and the idea of remote IMCs in chapter 27. The general idea of message blocks for inter-process communication within in the kernel is discussed in chapter 22.

Similar blocks are chained into the same lists when CMCs and library calls are executed, thus providing the kernel with a convenient overview of the progress of user threads.

9.4 Library Module Call Backs

As described in chapter 18, during the execution of a library call, the library routine can call its caller back using the LCB instruction. This call has only a single parameter, an integer indicating the entry point number in the main module's call-back entry point list.

To make an LCB call the library routine needs to pass parameters to its caller and expects to receive a result from the caller. To do this it needs to call `create_pc_params` before making the call.

The LCB uses the same parameter registers as other calls but it invalidates Segment Register 5. A return from an LCB call uses a normal `return` instruction.

10 Internal Calls

These calls, introduced in chapter 19, are very simple, allowing the compiler to structure the code of a module into multiple segments (thus allowing separate bounds checking on each). The kernel instruction `internal_call` has two operands. The first addresses the code segment in question. The second provides a new offset value for the program counter, indicating where execution should begin. Other registers are not stored or modified.

The instruction stores the current value of the code segment register and the program counter and reloads these with the values signified in the operands.

Chapter 21
Synchronisation

This chapter tackles the issue of how application threads (of the same process and of differing processes) can cooperate with each other, especially in the sharing of data structures within a module (which is the natural form of sharing in an in-process system). It describes how SPEEDOS applies standard techniques and introduces some further techniques which are less widely known and used.

1 Implementing Mutual Exclusion

In volume 1 chapter 8 the general concept of synchronising processes/threads was introduced. In particular that chapter described basic standard techniques, including the use of semaphores, to solve synchronisation problems such as mutual exclusion, producers and consumers using a shared bounded buffer, readers and writers, and private semaphores (used to control the sequencing of process/thread execution).

It is important, on the grounds of efficiency, to have two uninterruptable basic semaphore instructions, such as the DECT (decrement and test) and the TINC (test and increment) instructions described in chapter 8, because these allow a module to handle its own synchronisation situations in those cases where the suspending and reactivating of threads is not necessary. Such situations arise frequently, e.g. when no other thread is contending with the current thread for the use of a critical region; in combination with DECT/TINT instructions the overhead associated with unnecessary calls to a central User Thread Scheduler (UTS) can then be avoided.

DECT and TINC are needed as hardware instructions or uninterruptable kernel instructions (which must be coordinated to function correctly in a multiple CPU system). These must nevertheless be complemented by commutative *suspend* and *activate* routines similar to those which in conventional systems are provided by a central process/thread scheduler, to handle the cases in which clashes for the use of resources arise. It is assumed that the reader is familiar with these instructions, or refreshes his knowledge of them by re-reading volume 1 chapter 8.

1.1 Suspending and Activating Threads

In a secure system such as SPEEDOS it is important that the code of arbitrary modules cannot simply suspend a thread at will, since this may be a deliberate attempt to disrupt a user or the entire system. As was mentioned in chapter 20, there is a Thread Control Manager module (known as a TCM) in each process container, which controls the run-time activities of a thread. Since this is partially user-programmable and can vary from node to node, and even from process to process at the same node, it is impossible here to define all its potential functions. But the facilities which it offers in conjunction with the suspension and activation of threads must be standardised to the extent that threads of different users and processes working in a common environment can cooperate correctly.

It is not the function of a particular TCM to schedule all the threads in a system (i.e. to determine which may use the CPU(s) at a particular point in time); that is the function of the central UTS module. But it should be involved in decisions regarding the suspension and activation of its own threads. There are two good reasons for this.

First, if the code of any arbitrary module called by a thread could suspend that thread, a devious user could take advantage of this to bring threads to a halt without good reason, and thus cause chaos.

The second is that if all operations involving the suspension and activation of the threads of a process are channelled through its TCM, it can keep an overview of what is happening to them in scheduling matters. This can provide the TCM with information which is helpful in recovering from exceptional circumstances, such as the failure of a printer for which threads are waiting in a suspended state, or a failure of other threads which might otherwise leave them suspended "for ever". How such control is exercised in detail need not concern us here. The important issue in the present context is how the system can be organised such that TCMs are in a position to exercise an appropriate measure of control over their own threads.

Each TCM must provide the appropriate semantic routines to achieve this. It is initially assumed (a) that each TCM provides its own `suspend` and `activate` routines, although this will actually be discussed further below, and (b) that these are commutative[46]. This does not preclude a TCM from providing other routines for suspending and activating threads, nor from waking up a commutatively suspended thread in order to recover from errors, etc.

[46] See the discussion in volume 1 chapter 8 about commutative scheduling operations.

1.2 Organising Thread Capabilities

The next step is to limit calls on these routines to modules which are legitimately entitled to do so. The SPEEDOS solution in all such cases is to require that the caller has a capability. But where do these capabilities come from, and how are they distributed to the right recipients?

The capability needed to access the routines of a TCM for a particular thread is a *thread capability* for that thread, see Figure 21.1 (repeated from Figure 19.7).

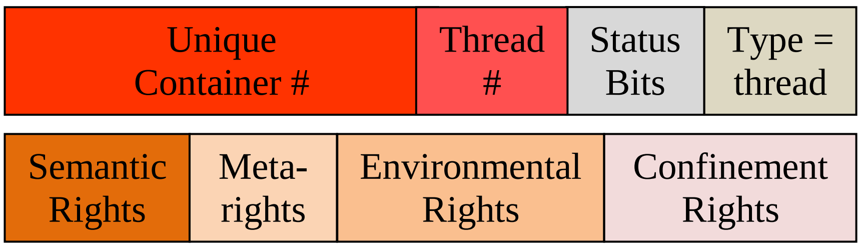


Figure 21.1: The Basic Structure of a Thread Capability

A thread capability is returned (with full access rights set) by the Thread Manager when the corresponding thread is created. A copy of this capability (preferably with the access rights limited to those needed to allow the thread to synchronise with other threads) must be available to any module entitled to carry out synchronising operations on the thread. Whether it will be provided with a permanent copy or a reduced use copy will depend on the circumstances.

TCMs have a capability for calling the central UTS, and are normally the only modules with this privilege (but see section 1.4). In this way, other modules can only call the UTS indirectly via the executing thread's own TCM, while other modules, which should not be calling the UTS at all, cannot do so, thus preventing arbitrary modules from creating chaos.

When a Thread Manager creates a new thread it places a thread capability for it at the base of the new thread's stack at a location known to the kernel. The latter makes such capabilities available on request to executing threads (see chapter 19 section 5), subject to the corresponding permission being set (see chapter 26 section 2).

A thread capability can be used in an inter-module call to activate the methods of the corresponding thread's TCM routines. This is special in that a thread capability is not the same as a module capability for the TCM. The reason for this is that some semantic routines of the TCM can be used to control any thread of the process, but in the synchronisation situation under discussion it would be unsafe to allow a synchronising caller to activate or suspend any thread of the process. Unlike a normal capability for a TCM, a thread capability limits the synchronising process to a single thread defined in the capability. This is achieved in that the unique container number in the capability identifies the container of the process, the thread number field indicates which thread can be synchronised (by the TCM) and the access rights correspond to the semantic access rights provided by the TCM. Hence when the kernel receives an inter-module call which presents a thread capability (recognisable from its type field) it in fact calls the TCM[47] responsible for the corresponding thread, passing to it an integer parameter (a copy of the thread number in the thread capability).

1.3 Implementing Semaphores with Thread Capabilities

An important consequence of this arrangement is that a thread capability is needed in semaphore operations in order that the code which carries out these operations can arrange for the activation and suspension of threads which are required to wait for a resource.

At first glance it may appear to be a simple task to modify conventional semaphore operations to take the thread capability into account. In SPEEDOS basic operations DECT and TINC are provided as kernel instructions. In conventional systems, these are combined with two central UTS routines (**suspend_me** and **activate**) to create P and V operations, which can be (provisionally) defined as follows.

```
P (sem) =>
     DECT (sem.counter, local);
     if local < 0 then scheduler.suspend_me(sem.queue);

V (sem) =>
     TINC (sem.counter, local);
     if local < 0 then scheduler.activate(sem.queue);
```

Notes on *conventional* P and V operations:

1. The semaphore variables and the related code sequences are located in user modules, which invoke the central scheduler's **activate** and **suspend_me** routines only when necessary.

2. The parameter **sem.counter** is the shared integer value of the semaphore (a positive value indicates the number of resources still available, a negative value the number of suspended threads waiting for a resource, and a value of 0 that the resource or resources are currently in use but no thread is waiting). (If the resource is a critical region requiring mutual exclusion, the value is initialised to 1.)

3. The value of the returned parameter **local** is a thread-local copy of the semaphore variable's integer value as defined in DECT and TINC (see volume 1 chapter 8). The executing thread can be interrupted at any point in

[47] The TCM has a fixed position in the Co-Module Table in process containers

the code, though the scheduler routines themselves must be indivisible and commutative.

4. The routines **`suspend_me`** and **`activate`** are calls to a central UTS and the parameter which they pass to the scheduler (**`sem.queue`**) identifies the queue associated with the semaphore. The queue itself is held in the scheduler's persistent data.

5. To create such a queue the module which contains the semaphore must invoke a scheduler routine; after setting up the queue, this returns an identifier for it. The queue must be created before the **`suspend`** and **`activate`** routines can be used. Later the scheduler manages this queue in the code of these routines.

The following code represents an intuitive (but incorrect) first attempt to incorporate the use of thread capabilities into the normal semaphore scheme, in order to include the TCM in the picture.

```
P (sem, thread_cap) =>
     DECT (sem.counter, local);
     if local < 0 then thread_cap.suspend(sem.queue);

V (sem, thread_cap) =>
     TINC (sem.counter, local);
     if local < 0 then TCM[thread_cap].activate(sem.queue);
```

This code contains deliberate problems and errors, a discussion of which helps to illustrate the real nature of the task at hand.

1.4 Creating Queues

The appropriate time to create a queue variable for a semaphore is when the semaphore itself is created. This typically occurs in a module which contains critical sections that need to be synchronised, which is neither normally a TCM nor should it have direct access to the central UTS.[48]

One possibility would be to make a UTS capability freely available to all modules with access rights limited to the creation and deletion of queues. While this would allow queues to be created, it would open up the possibility that a malicious user might hinder the work of the UTS by creating many unnecessary queues.

The alternative here proposed is to introduce the idea of *privileged library modules*. These are basically normal library modules (see chapter 18 section 6.4) that are delivered as part of the SPEEDOS system which have certain privileges. In this case the synchronisation library module is trusted to have a UTS capabil-

[48] This is an issue not discussed in Espenlaub's thesis, since he viewed the design of the Thread Scheduler and of TCMs as a matter for the operating system design, not the kernel design.

ity which allows it to call the UTS to create thread queues. It obtains the (non-copyable[49]) capability from one of its own constant segments. This is placed there as part of the installation of a SPEEDOS system.

Any SPEEDOS module can access this library module if it has access to the appropriate code capability, which provides a high level interface for creating and using semaphores. Its semantic routines include a constructor (i.e. an initialisation routine) to create a semaphore variable, a `claim` method and a `release` method.

At the point in the code where a semaphore is needed, the initialisation routine of the main module executes a kernel library call instruction (LC), passing to the kernel the following parameters: a code module capability for the selected library routine, the entry point number 0 (i.e. for the constructor) and a pointer address (via which the root segment of the library module can later be used to make further library calls to the `claim` and `release` routines).

The constructor creates an instance of a semaphore, which includes not only the semaphore itself but also the UTS's queue identifier for the semaphore. This routine carries out some checks. For example it ensures that the host module is on the current node; it might also have a list of modules which are permitted to create semaphores together with the number of semaphores permitted, etc. (This saves the UTS from carrying out such checks.) Provided that the request is valid, it calls the UTS to create a queue, notes the identifier of the queue in its own root segment. It then creates and initialises the semaphore variable.

The `claim` method carries out the entire P operation, including the DECT operation and if necessary calls the queuing operation of the UTS). For this purpose it must have access to the thread capability for the currently executing thread; this is obtained via a kernel instruction (see chapter 19 section 5). Since thread capabilities can be dangerous if they fall into the wrong hands, the kernel can test whether the request is issued from the code of the synchronisation library module.

The `release` method carries out the entire V operation, including the TINC operation and if necessary the de-queuing operation).

1.5 Informing the TCM of the Activation of its Thread?

The deliberately erroneous program snippet for the V operation at the end of section 1.3 suggests that a thread capability for a thread being activated in the V operation is needed, in order to invoke an activate routine of its TCM. To obtain such a thread capability would in fact be a rather difficult task, since the thread

[49] for access rights in capabilities see chapter 26.

executing the V operation is *not* the thread to be activated.

Fortunately this is not necessary, for the following reason. The thread to be activated is currently suspended in the UTS, which was called by the thread's own TCM, and so after the UTS reactivates the thread it will exit from the UTS back to its TCM. When this point is reached, the TCM code can recognise anyway that the activation has occurred and can, if appropriate, record this in its own data. Hence what could have been a troublesome problem simply disappears!

If this were the final solution, the library module would have **claim** and **release** methods with the following skeleton code (which is not correct Timor but merely illustrates the principle):

```
claim(sem){// sem identifies the semaphore (integer) variable
            // sem.counter is held in the library routine
    // the P operation
    theCapability thread_cap;
    DECT(sem.counter, local);
    if local < 0
      {thread_cap = kernel.get_current_thread_cap(kernelCap);
                 thread_cap.suspend_me(sem.queue);
                // inform TCM of delay
                // TCM calls UTS to queue the thread
                }
    }
release(sem){/* sem identifies the semaphore (integer) varia
              ble */
            // sem.counter is held in the library routine
    // the V operation
    TINC (sem.counter, local);
    if local < 0 {Thread_Scheduler.activate(sem.queue)};
    // after the UTS selects and activates
    // a queued thread. This will return to its TCM
    // and then continue to use the resource.
```

1.6 Delegating Queuing to the Library Module?

At this point we note that the semaphore counter and the semaphore queue have been separated; the counter is in the library module and the queue is in the UTS. This raises an interesting question. If the main semaphore activity is placed in a trusted synchronisation library module, then why not do the same with the queuing operations? In this case one could envisage that at the appropriate time the library module carries out the queuing operations in its own space[50] and simply calls the central UTS to suspend or activate a thread. At first sight this idea seems to have a number of advantages:

[50] In reality a library routine shares the file (or program) space of its host module, but the host is not explicitly aware of this.

a) This would allow different queuing strategies to be used for different semaphores (e.g. using priority or FIFO techniques).

b) It would very considerably reduce the work of the UTS, which is the most frequently activated module in a system, and the most crucial from an efficiency viewpoint. All the scheduler would need to do in this context is to provide a "ready list" of threads, from which it selects a thread to run on each CPU according to its scheduling algorithm and the scheduling parameters supplied to it by the TCM. Of course the UTS must also be called (by the TCM) to remove the current thread from the ready list after it has been placed on a waiting queue by the library module.

c) The scheduling queues are distributed through the system rather than all being stored in the UTS. Consequently an attempt by hackers to manipulate or destroy the system by interfering with these queues becomes much more difficult.

These advantages sound very tempting, but one must also consider the implications of this solution. If the organisation of scheduling queues is delegated to the library modules, this does not eliminate the need to synchronise them, e.g. to avoid problems when several threads want to place entries onto or remove them from the queue in parallel, i.e. a scheduling queue is a critical section. But unlike other critical sections it cannot be synchronised using semaphores, because this queue is part of the technique to implement semaphores!

This does not rule out the solution entirely. The alternative is to use a more primitive synchronisation mechanism to synchronise this queue. In volume 1 chapter 8 a number of such mechanisms were mentioned, including some which use busy waiting (e.g. turning off interrupts, busy waiting instructions such as *test-and-set* or *compare-and-swap*). As a general mechanism for providing mutual exclusion these are all less efficient than semaphores, as was explained in chapter 8, but nevertheless the semaphore queuing operations must be implemented using one of these techniques in the UTS, so it is not entirely out of the question that these be used for precisely the same purpose in library modules, provided that the use of these modules can be well protected.

The low-level synchronisation mechanism preferred for the SPEEDOS UTS by Espenlaub is turning off interrupts, but he does not provide a kernel instruction to achieve this, instead treating the scheduler routines as a special case, without indicating his reason for this [4, pp. 169-170]. It is therefore tempting to consider whether a general mechanism for turning interrupts off (and back on)[51] could be made available by the kernel for use by both the UTS and the synchro-

[51] An instruction pair for turning on/off interrupts at the user level was provided in the MONADS systems, see [5, p. 125].

nising library routines. The standard SPEEDOS technique to do this would be kernel instructions for turning interrupts on and off, protected by means of a kernel capability, as will be described in the next chapter.

If such a mechanism were *really* to turn off interrupts it would not solve the problem at hand, because (a) only the kernel itself is privileged to turn off interrupts, and (b) if it were to do this for non-privileged code the latter could then neither turn interrupts back on itself nor could it activate the kernel to do so, since kernel instructions are recognised as a result of handling an interrupt! Now we see why Espenlaub treated the UTS as a special case.

However, this objection is not a serious as it may appear, since all that is required is that any interrupts which actually occur should not be visible *at the user system level*. What happens at the kernel level is, or should be, invisible to the user system (including the UTS and the synchronisation library module(s)). This was not possible in Espenlaub's SPEEDOS design, because threads which handle real interrupts are scheduled by the UTS. However, as we shall see later, there are good reasons (apart from this) to prefer the MONADS thread scheduling solution, in which turning off and on interrupts at the user level does not affect the kernel's ability to react to real interrupts (including a kernel instruction requesting interrupts to be turned on again).

The kernel instructions for turning interrupts off and on are very simple and are based on Rosenberg's proposal for MONADS:

```
disable_interrupts(modcap kernel_cap)
enable_interrupts(modcap kernel_cap)
```

In SPEEDOS the instructions require a kernel capability to ensure that they are not misused. This capability can also be provided as a constant in the code of the library module (and of the UTS).

1.7 The Final Solution

It is easy to think that the problem has been solved, but to make sure we look again at the code of the library module. Remember now that both parts of the semaphore variable – **`sem.counter`** and **`sem.queue`** – are implemented together in the privileged library routine.

The entries in scheduling queues consist primarily of thread capabilities for suspended threads. It may be sensible to add other information (e.g. the time at which the thread was placed on the queue, which may turn out to be helpful in detecting and correcting errors (such as the death of the thread which is queued on a semaphore), but such additional features are ignored here.

However, by using thread capabilities as entries in the queue a further advantage is gained. When these were hidden within the user UTS the thread capabilities also remained hidden, but now they are used as queue entries outside the user UTS more flexibility (and efficiency) has been gained. It now becomes possible to simplify the **`release`** routine by bypassing the TCM and calling the user UTS's **`activate`** routine directly. We have already seen earlier that the TCM can be informed of an activation of one of a suspended thread when it exits from the user scheduler after the thread has been re-activated. So nothing is gained (and efficiency is lost) by calling the TCM of a suspended thread when it needs to be activated. In other words, it is sufficient to suspend the thread in a **`claim`** operation and rely on its return from the user UTS, in the second half of the **`suspend_me`** routine, to inform its TCM. Hence the library module can now be defined as follows.

```
Library Synchronisation {
MutualExclusion {
 semaphore sem;   // sem identifies the semaphore variable.
                  // sem.counter and sem.queue are both held
                  // in the library module
claim(sem){
     // the P operation
    DECT(sem.counter, local);
    if local < 0 // i.e. if the current thread must wait
      {// obtain capability for current thread from kernel */
       threadCapability currentThreadCap =
                     kernel.get_current_threadCap(kernelCap);
      // disable (pseudo-)interrupts
         kernel.disable_interrupts(kernelCap);
      // add current thread to queue
         sem.queue.enqueue(currentThreadCap);
      // the claiming thread is still executing
      // turn interrupts back on
         kernel.enable_interrupts();
      // inform TCM of delay;
         currentThreadCap.suspend_me();
      // TCM notes the suspension and calls the
      // UTS to remove the thread from its ready list
      }
    // NOTE: when the current thread is later activated
    // by the UTS it will return to the TCM.
    // It will then return to this point
    // and can access the resource.
    // The thread then returns from the
    // library module to continue its normal code.
    }
release(sem){
     // the V operation
    TINC (sem.counter, local);
    if local < 0 // i.e. if a thread must be activated
      {// disable (pseudo-)interrupts
        kernel.disable_interrupts(kernelCap);
      // select and remove a thread capability from sem.queue
```

```
          threadCapability newThreadCap =
              sem.queue.dequeue();
          kernel.enable_interrupts();
          /* informs TCM to activate thread by passing
             it to UTS's ready queue */
          user_thread_scheduler.activate(newThreadCap)
          }
      }
      // The selected thread will then be scheduled
      // and continue to use the resource (e.g.critical region)
      // The current thread continues by exiting
      // from the release routine
  } // end MutualExclusion
  ... // other synchronisation mechanisms
  }   // end of library module
```

There remains one final issue with respect to this solution. What has in effect been done is to outsource some decisions about when threads can be activated and suspended from the UTS to the Library module and the Thread Control Managers. So far one important aspect of this activity has been left unmentioned, viz. the saving and restoring of the state of the thread states which are delayed and restarted. Put simply, the register states of the threads involved must be stored and reloaded to enable other threads to use them. However, the way this has been organised as an in-process mechanism[52] does not affect the important point that the UTS alone makes the final decision about when a thread is actually delayed or activated. For example if a thread has to wait as a result of a semaphore P operation it actually continues to execute until it arrives at the UTS, which suspends it. Similarly if as a result of a V operation a new thread has to be activated, the thread releasing the resource continues to execute until it reaches the UTS and advises the latter that the new thread can be activated. Hence no special action is needed in the library module nor in the TCM with regard to thread switching. How the UTS actually switches threads will be described in the next chapter.

1.8 Summarising the Queuing Operations

Since the queuing operations will turn out to be useful in the implementation of other more specialised semaphores discussed later in the chapter, it will be useful to summarise these here, without comments.

1.8.1 The Suspend Operation

Here is the suspend operation in essence.

```
  if (current thread must suspend)
      {threadCapability currentThreadCap =
```

[52] For programmers familiar with out-of-process systems this may at first be a little difficult to understand, which is why I have attempted to spell it out in detail here.

```
           kernel.get_current_threadCap(kernelCap);
     kernel.disable_interrupts(kernelCap);
     sem.queue.enqueue(currentThreadCap);
     kernel.enable_interrupts();
     currentThreadCap.suspend_me();
    }
```

1.8.2 The Activate Operation

Here is the activate operation in essence.

```
if (a thread must be activated)
    {kernel.disable_interrupts(kernelCap);
     threadCapability newThreadCap = sem.queue.dequeue();
     kernel.enable_interrupts();
     user_thread_scheduler.activate(newThreadCap);
    }
```

This initially concludes the discussion of how mutual exclusion can be implemented in SPEEDOS. I have described this in a number of steps in order to give readers with little experience of synchronisation some idea of how complex this can be and therefore how easy it is to make mistakes, since without explanation the implications of the above steps would probably not be clear, possibly not even to experienced programmers.

In the following sections we describe some extensions of the semaphore idea. These too can, where appropriate, use the same or similar synchronisation modules.

2 Applying DECT/TINC to Other Problems

In this and the following sections we describe some extensions of the semaphore idea. These too can, where appropriate, use the same synchronisation modules.

One of my former PhD students, Prof. Bernd Freisleben, illustrated in his PhD thesis [11, 12] how the basic DECT and TINC instructions can be combined with commutative scheduler routines not only to achieve mutual exclusion, but also for many other synchronisation purposes, including useful operations for user scheduling of threads and Conradi's P* operation [13], synchronisation involving thread counting operations, Campbell and Habermann's path expressions [14], critical block exit in block structured programming situations such as is found in the B6700, and simultaneous P operations (a modified P operation which allows multiple resources to be claimed together, with the aim of avoiding deadlocks). Freisleben goes on to show that with an extension to DECT/TINC operations all synchronisation problems which can be solved by eventcounts and sequencers [15] can be solved efficiently with an extension.

These operations could be implemented in an analogous way to mutual exclusion, with the help of library modules.

Unfortunately for English readers, Freisleben's thesis is in German, but the essential aspects of these solutions have been described in English [16].

3 Semaphores for Different Classes of User Threads

This section begins by describing a special technique developed specifically for handling the reader-writer synchronisation problem. It then goes on to describe a generalisation of this solution for a wider class of problems.

3.1 Reader-Writer Semaphores

Since reader/writer situations occur very frequently in the design of operating systems and database systems, and the protocols to achieve this form of synchronisation using normal semaphores are neither trivial nor particularly efficient (see volume 1 chapter 8), SPEEDOS supports specialised reader-writer semaphores. These were first developed in the context of the MONADS project [17]. Whereas the structure of a normal semaphore consists of an integer and a related queue, a reader-writer semaphore consists of three integers and a boolean variable (see Figure 21.2) together with separate queues for waiting readers and waiting writers.

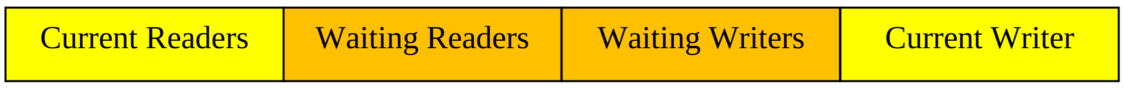


Figure 21.2: The Structure of a Reader-Writer Semaphore

The first three fields respectively hold counts indicating the numbers of current readers, waiting readers and waiting writers while the fourth (boolean) field indicates whether a writer is currently active. This structure will easily fit into a 64 bit word. The initial values of these fields are zero or false.

There are four primary instructions which operate on this structure, as it was developed for use in the MONADS systems:

READ-P is used by a thread to attempt to claim reader access to the critical region. This returns a thread-local boolean result indicating whether the thread should suspend itself on the reader queue.

READ-V signals that a current reader thread is now relinquishing access to the critical region. This returns a thread-local boolean result indicating whether a writer must be activated.

WRITE-P is used by a thread attempting to claim writer access to the critical region. This returns a thread-local boolean result indicating whether the writer should suspend itself on the writer queue.

WRITE-V signals that the current writer thread is now relinquishing access to the critical region. This returns a thread-local boolean result indicating whether a thread should be activated, and a thread-local integer result which indicates whether a further writer should be activated (if the integer = 0) or how many readers should be activated (if the integer > 0).

The published description of this technique provides further details, including the simple algorithms for the four instructions, which can be formulated with a small difference in terms of reader priority or writer priority. It is also shown that this approach is considerably more efficient than implementing reader-writer algorithms using only normal semaphores.

Like DECT and TINC these instructions were designed for use in conjunction with commutative UTS operations. The *suspend* interface remains unchanged from that described in association with DECT, but the *activate* interface requires not only the naming of a queue but also an additional integer parameter defining exactly how many threads need to be activated. This modification can be used in cases involving the TINC instruction simply by setting this parameter to 1.

To adapt this to the SPEEDOS environment requires that

a) the SPEEDOS kernel supports the four instructions described above as indivisible kernel instructions, which are then used as appropriate in the module's code in a similar manner to DECT and TINC.

b) the library modules described in connection with mutual exclusion above be used to organise the queuing operations, but appropriately modified by adding an additional integer parameter to the WRITE-V `activate` routine in order to allow it to activate several reader threads if necessary.

c) thread capability parameters are added to suspend operations as appropriate.

We illustrate here how these semaphores would appear in a library routine using a similar pattern to that used for mutual exclusion in section 1.8 above.

```
ReaderWriter {
rwSemaphore rwSem;
 readClaim(rwSem)
    {READ-P (rwSem, local);
     if local // i.e. if the reader must wait
        {threadCapability currentThreadCap =
                kernel.get_current_threadCap(kernelCap);
         kernel.disable_interrupts(kernelCap);
         rwSem.readerQueue.enqueue(currentThreadCap);
         kernel.enable_interrupts();
         currentThreadCap.suspend_me();
        }
    }
 readRelease(rwsem){
    READ-V (rwSem, local);
    if local // i.e. if a writer must be activated
```

```
              {kernel.disable_interrupts(kernelCap);
               threadCapability newThreadCap =
                             rwSem.writerQueue.dequeue();
               kernel.enable_interrupts();
               user_thread_scheduler.activate(newThreadCap);
              }
      }
   writeClaim(rwSem){
      WRITE-P (rwSem, local);
      if local // i.e. if the writer must wait
          {threadCapability currentThreadCap =
                   kernel.get_current_threadCap(kernelCap);
           kernel.disable_interrupts(kernelCap);
           rwSem.writerQueue.enqueue(currentThreadCap);
           kernel.enable_interrupts();
           currentThreadCap.suspend_me();
          }
      }

  writeRelease(rwSem){
      WRITE-V (rwSem, local, readerCount);
      if (local & readerCount == 0) // a writer to be activated
          {kernel.disable_interrupts(kernelCap);
           threadCapability newThreadCap =
              rwSem.writerQueue.dequeue();
           kernel.enable_interrupts();
           user_thread_scheduler.activate(newThreadCap);
      else
      if (local & readerCount > 0) // readers to be activated
          {kernel.disable_interrupts(kernelCap);
           for i in {1 .. readerCount}
            // activate a reader each time through for loop
              {threadCapability newThreadCap =
                       rwSem.readerQueue.dequeue();
               user_thread_scheduler.activate(newThreadCap);
              }
           kernel.enable_interrupts();
      }
  } // end ReaderWriter
```

3.2 Priority semaphores

This semaphore variant [18] was proposed by Freisleben and myself as a generalisation of the semaphore concept to allow for claims on a resource (e.g. a critical section) being made by different classes of threads with different priorities, whereby the use of the resource by different classes can be determined to be either mutually exclusive or shared within a class. From the application viewpoint there are only two simple instructions at the machine level, PRIORITY-P and PRIORITY-V for requesting and releasing resources respectively. PRIORITY-P nominates a priority semaphore (implemented in the library routine) together with an integer indicating its priority class and a thread-local boolean variable.

PRIORITY-V releases the semaphore and has four operands: the priority semaphore, the priority class, a count of threads and a thread-local boolean variable. Like the earlier cases, the instructions are used in association with library module queues, whereby the activate routine, as in the case of reader-writer semaphores, has an additional parameter indicating how many threads need to be activated.

In principle priority semaphores eliminate the need for reader-writer semaphores, since the latter are a special case of the former more general solution. But since the reader-writer problem is such a common problem and the implementation of reader-writer semaphores is more efficient than that of priority semaphores, there is a strong case for implementing reader-writer semaphores in SPEEDOS as described above. On the other hand the relative complexity of priority semaphores and the fewer synchronising problems for which they are relevant suggests that they should not necessarily be implemented as kernel instructions in SPEEDOS[53].

4 Set Semaphores

In Chapter 8 it was explained that the value of the integer associated with a general semaphore can be understood as follows:

> 0: the number of resources currently available

= 0: no resources free and no waiting threads

< 0: the number of threads waiting for a resource.

What the integer fails to indicate is which resources, if any, are currently available, or which threads, if any, are currently waiting. For this reason my former students and I proposed an extension, called set semaphores, which supplements the integer with a set (implemented as a bit list), whereby each bit in the list represents one of the set of available resources (if the integer value is positive) or one of the set of waiting threads (if the integer value is negative) [19]. Assuming that the bits are numbered from left to right (starting at zero) and the bit list has the value 0010110... then depending whether the integer part is positive or negative, this means that resources numbered 2, 4 and 5... are available, or that threads numbered 2, 4 and 5... are waiting to acquire a resource. Typically the set semaphore will be initialised by setting the number of available resources in the integer part and the set part will have bits set indicating which resources are free.

The two meanings of the bit list need not be used together, i.e. it is possible

53 If a decision were made to implement priority semaphores as kernel instructions, the pattern for doing this would follow a similar pattern to that used for reader-writer semaphores.

to work only with resource sets or only with thread sets. First we describe resource sets in more detail.

4.1 Resource Sets

For resource sets the following two basic instructions (similar to DECT and TINC) are needed.

RSETP claims a resource. This has three operands. The first addresses the set semaphore (which is implemented as an integer and a bit list). In the second, an integer, the instruction indicates which resource, if any, has been allocated. The third, a thread-local boolean value (e.g. a condition code), indicates whether a resource has been allocated.

The instruction decrements the semaphore's integer value by one. If the result is greater than or equal to zero, the boolean result indicates that a resource has been allocated. If so the application discovers from the integer result which resource has been allocated (and the instruction removes the corresponding resource from the bit list).

If a resource was not allocated the thread suspends itself on the associated scheduling queue. When it is activated (see below) the resource which has been allocated is indicated in a result from the suspend routine.

RSETV releases a resource. It also has three operands. The first addresses the set semaphore; the second, an integer, indicates which resource, if any, is being released; the third, a thread-local boolean value (e.g. a condition code), indicates whether a thread should be activated. The instruction increments the semaphore integer by one and if the result is greater than zero (i.e. a resource is now available) it sets the thread-local boolean variable to indicate that a thread should be activated from the corresponding scheduling queue. The releasing thread then causes a thread to be activated from the corresponding scheduling queue, indicating which resource it has released in an integer operand.

4.2 Waiting Thread Sets

Waiting thread sets are organised in a similar fashion to resource sets, although they are less relevant to synchronisation and more relevant to thread scheduling. We now describe how waiting thread sets function and then we examine their advantages and disadvantages. Although it will become clear that they cannot sensibly applied to normal user thread scheduling, it will be shown in the next chapter how they can create an excellent basis for scheduling kernel threads designed to manage interrupt handling.

4.2.1 How Waiting Thread Sets Work

Just as resource sets can be implemented without waiting thread sets, so also waiting thread sets can be implemented without resource sets. We now describe how they can function in association with normal scheduling queues for implementing the wait operation. In this case, two instructions (WSETP and WSETV) are required.

WSETP has three operands. The first addresses the set semaphore. The second, an integer, identifies the currently active thread. The third, a thread-local boolean value (such as a condition code value), indicates whether the thread must suspend itself on a scheduling queue until a resource becomes available.

The instruction decrements the semaphore integer and if the result is greater than or equal to zero, a resource is available and the thread-local variable indicates that the thread can proceed without invoking a scheduler routine. (Unless it is combined with a resource set it must by some other means establish which resource it has now acquired, unless only a single resource, e.g. a critical section, is involved.)

If a resource could not be allocated, the thread-local variable indicates that the thread should call a scheduler to suspend itself, and the thread number corresponding to the calling thread is set to one in the waiting thread set.

WSETV releases a resource; it has three operands. The first addresses the set semaphore. The second, an integer, returns the identity of a thread to be activated from the scheduling queue. The third, a thread-local boolean value, indicates whether a waiting thread should be activated.

The instruction increments the integer part of the semaphore by one and tests whether the result is greater than zero. If so the thread-local variable is set to indicate that the resource has been successfully deallocated and that no further action is required. If the result is less than or equal to zero (thus indicating that the thread must activate a waiting thread) the thread-local boolean operand is set to indicate that another thread must be activated. It selects a new thread from the thread set and clears the corresponding bit. The thread is advised in the second operand which thread must now be activated by calling the scheduler.

4.3 Applying Set Semaphores in SPEEDOS

At the level of synchronising user threads, only the resource sets are relevant. Waiting process sets would have the following problems at the level of user synchronisation.

a) At his level in the system there is an unspecified number of user threads which cannot be uniquely identified simply by integers.

b) The selection criterion for selecting a thread to execute is in effect a priority mechanism, which could lead to unfair starvation of some threads[54].

c) Waiting process sets cannot easily be adapted to the principle that TCMs should be kept up to date.

As we shall see shortly, this does not imply that waiting process sets are entirely irrelevant for the design of SPEEDOS.

However, such restrictions do not apply to resource sets. These can be implemented for use by normal SPEEDOS user threads in a manner similar to the above description of mutual exclusion. The differences are as follows.

a) The SPEEDOS kernel supports the two instructions RSETP and RSETV as indivisible kernel instructions (for use by user-level threads), which are then used as appropriate in the module's code in a similar manner to DECT and TINC.

b) An additional integer parameter is required in the library module **`activate`** routine, allowing this to return the number of the free resource;

c) A thread capability parameter is added to suspend operations as appropriate. This does not affect the kernel's RSETP instruction.

5 Summary

In the first section of this chapter it was shown how the normal queuing operations for semaphores can be extended to allow for the SPEEDOS use of thread capabilities which allow Thread Control modules to be kept informed of the status of their threads. It was further shown how the work of the central UTS can be reduced by placing the responsibility for the synchronisation queues associated with semaphores with a privileged synchronisation library module, thus reducing the work of the central UTS and increasing its efficiency (which is important because this is normally the most activated routine in an operating system).

This technique can also be applied to the extended semaphore types discussed above. These should form the basis for a number of synchronisation library modules which made are available to all SPEEDOS users. Such library modules should at least include:

a) normal mutual exclusion based on TINC and DECT instructions, which can also be used, for example, to implement scheduling control via private semaphores[55];

b) reader-writer synchronisation based on READ-P, READ-V, WRITE-P and WRITE-V instructions;

[54] A round robin algorithm could be implemented with a little more overhead.

[55] see chapter 8 section 12.3.

c) the allocation of resources based on RSETP and RSETV instructions.

The implementation should be based on non-interruptible kernel instructions and non-interruptible commutative queuing routines, managed in synchronisation library modules based on the pattern described in section 1.

In the next chapter it will become clear how waiting thread sets can make an extremely useful contribution in the scheduling of kernel processes.

Chapter 22
Thread and Process Scheduling

This chapter discusses thread scheduling in SPEEDOS. The key issues here are what role the kernel plays in this activity, which thread is active on a CPU at a particular time, how CPU interrupts are handled and finally, how a persistent process and its threads are logged out and back in. The important theme of synchronisation, i.e. how threads sharing the same data can synchronise their activities with each other, was described in Chapter 21.

1 The Kernel's Role in User Thread Scheduling

Since thread scheduling algorithms can be quite complex, and different algorithms can be appropriate depending on the kinds of applications which are executed at a node (see volume 1 chapter 8), it is appropriate that the user level scheduling algorithm itself should not be built into the core kernel, but should be provided in a security sensitive co-module, viz. the (central) User Thread Scheduler (UTS), thus allowing different SPEEDOS nodes to have different scheduling policies. However, this is not an activity which belongs in each container (in contrast with many other security sensitive co-modules), but is best placed in a separate container.[56]

Here it is only important that the scheduling activity exists outside the core kernel, and that it can cooperate and communicate with the core kernel, which

[56] The previous chapter discussed the activities of the UTS in the context of synchronising user-level critical regions via semaphores, and the conclusion was reached that much of the code for handling this difficult issue can be outsourced to a privileged library routine. This both reduces the work to be carried out by the UTS (which is important since this is usually the most invoked module outside the kernel) and increases the security of the system. What remains for the UTS to do is to maintain lists of user level threads waiting for input-output operations to terminate and a list of user level threads which are ready to run. From its ready list it selects the thread(s) to which the CPU(s) should be allocated at any given point in time, i.e. to schedule the use of the CPU(s) at the user level.

must be responsible for the crucial security aspects of thread scheduling. When the UTS communicates with the core kernel (by executing kernel instructions), it always presents a kernel capability to identify itself.

While the UTS decides which user threads should run on a CPU it does not carry out the actual (highly sensitive) thread switching activity itself. Instead it signals to the kernel that a thread switch is necessary by executing the kernel instruction `thread_switch`. This has three parameters. The first, as indicated above, is an appropriate kernel capability demonstrating its right to call this kernel instruction. The second is the unique thread identifier of the thread to be activated. This consists of a process container identifier in combination with the thread index number. The third parameter 'sleep' indicates to the kernel that the currently active thread is being suspended (without activating a further thread). When the `thread_switch` instruction is invoked, the process container must be a local container, since the thread switch must be immediately effective in order to guarantee that the CPU(s) at the node are used to best advantage. Thus a UTS only schedules threads residing at its own node.

When it is called to make a thread switch, the first task of the core kernel is of course to check the authorisation of the caller (by examining the kernel capability passed to it) and then, assuming that this is in order, it must store the state of the current thread (i.e. the thread which is about to lose the CPU). It does this by storing its current register values in a save area at the base of its [the thread's] thread stack. Having stored the state of the current thread, the core kernel must then restore into the CPU registers the previously stored state of the thread now selected to run. The full thread identifier of the new thread allows the kernel to locate its container and, using the thread number as an index into the container's Thread Table, to locate the base of its thread stack, where the current state of the selected thread was previously stored. Having reloaded this state, the new thread can continue.

In Espenlaub's version of SPEEDOS the kernel supports a further thread switching instruction, `return_thread_switch`, which, in addition to the parameters supplied with the normal `thread_switch` instruction, expects as an additional parameter the number of a segment register containing return parameters (as in an inter-module return instruction). Espenlaub described this instruction as:

> "... a combination of the `inter_module_return` instruction and the `thread_switch` instruction. Its purpose is to avoid blocking threads in the context of the UTS. Since each node has a different UTS and it is generally not possible to invoke methods of a particular UTS on a node other than that for which it is responsible, this would prevent the straightforward migration of threads to another node..." (Espenlaub, p.240)

In the new SPEEDOS approach to parameter passing (described in chapter 20 section 6), this additional parameter would be unnecessary, because by definition segment register 1 holds the parameters returned from an inter-module call. In SPEEDOS threads are not migrated as Espenlaub envisaged, but the kernel instruction may nevertheless still be useful to combine an inter-module return with a thread switch to the chosen thread.

When a new thread is first activated, the thread's TCM passes it to the UTS interface routine **`start_new_thread(kernel_cap, new_thread_cap)`** and the UTS uses the kernel instruction `new_thread` (see chapter 20 section 8.2) to activate the new thread.

In order to allow a thread to delete itself the UTS provides an interface routine **`kill_me()`** which can obtain a thread capability for the thread in the usual way. As the final instruction executed (by the UTS) in this thread the kernel instruction `switch_delete` is executed, which nominates a new thread to be executed and deletes the current thread. This is normally called on the advice of the thread's TCM, but may also be called by a surrogate thread (cf. section 11.2 below).

A further kernel instruction which can be called by the UTS is the `idle` instruction, which indicates to the kernel that there are currently no user-level threads to be scheduled.

Finally there is a kernel instruction `shutdown`, which has as its only parameter a kernel capability authorising the caller to execute the instruction. The kernel then immediately closes down the system. (It is the responsibility of the associated security co-modules to write the content of all active pages to disc before the instruction is executed.)

2 The User Thread Scheduler

It is important that the various nodes in a SPEEDOS network can have different scheduling algorithms tailored to their specific needs. This is one reason why the UTS is implemented as a security sensitive co-module rather than as an integral part of the core kernel. This approach supports the principle of separating mechanisms from policies. It is also important in that it keeps the fully privileged activities of the core kernel to a minimum.

The UTS has no special privileges except that it has access to a kernel capability which allows it to call the kernel's `thread_switch` and related instructions, including the disabling and enabling of (pseudo) interrupts (`enable_ interrupts`, `disable_interrupts`).

2.1 Interrupt Handling at the UTS Level

User threads often need to wait for the occurrence of some event, e.g. that a particular time has been reached, that a number of milliseconds has expired, that an input-output operation which the thread has initiated has now completed, etc. Such occurrences are initially known to the kernel as a result of an interrupt, and the kernel must then advise the UTS when such an event occurs. This is achieved by means of a pseudo-interrupt mechanism, whereby the kernel can place a message for the UTS into a buffer which is accessible to both the kernel and the UTS. The kernel can then activate the UTS to handle the pseudo-interrupt. However this activity must be co-ordinated, because the UTS might otherwise be interrupted in the middle of a critical section.

To avoid such a situation the UTS can use the kernel instructions `enable_interrupts` and `disable_interrupts` to indicate when it can/cannot be safely interrupted by the kernel. If the kernel is ready to activate a pseudo-interrupt but the UTS has turned off interrupts, the kernel buffers the interrupt until interrupts are turned back on. If it meanwhile wants to activate further interrupts, these are also added to the buffer, and when interrupts are turned on by the UTS, the kernel immediately turns off interrupts again for the UTS and passes to it the first pseudo-interrupt, etc.

This mechanism has a further advantage. The normal semantic routines of the UTS are called by normal user threads (e.g. to request that they be suspended), i.e. the Scheduler itself is a critical section and it is therefore necessary to prevent multiple users from being active concurrently in the UTS. By turning off interrupts the UTS also prevents other threads from being active[57], and hence provides a mutual exclusion mechanism for the UTS (which cannot use the normal mechanism, since its function is partly to implement the mechanism which others use, and attempting to use the same mechanism would become a recursive problem.)

As we saw in chapter 21 section 1.6, the same mechanism can be used by the privileged synchronisation library module, since it also ensures that a reschedule cannot take place in the UTS, and therefore can be guaranteed (in a single CPU node) not to lose the CPU at the user level. The fact that real interrupts can still be serviced by the kernel does not affect this, since relevant interrupts will simply be buffered by the kernel until interrupts are turned back on by the library module.

One final point: the description above envisages a simple interrupt mecha-

[57] This is the case only in a single CPU system. To keep the issue simple we consider only single-CPU systems in this chapter. The measures to be taken in a multiple CPU system depend on the design of the co-ordination facilities available in such a system.

nism, but in fact it would be possible to simulate an interrupt system for the UTS in which interrupts could be selectively turned off, in which different classes of interrupts have different priorities, etc.

3 Scheduling Parameters

The example thread scheduling algorithm described in volume 1 chapter 8 illustrates the need for threads to have scheduling parameters (e.g. a priority, a time slice). This aspect of a thread is not considered to be part of the thread's state (in the sense of register values, etc.), but it is a significant factor in optimising the throughput of a system and in guaranteeing the performance of particular threads (e.g. for real-time process control purposes).

SPEEDOS provides for the management of such scheduling parameters (which might not merely involve the setting of static values but also their automatic re-calculation based on past performance, for example) by including in each process container a Thread Control Manager (TCM) as a further co-module. What this actually does depends on the nature of the process.

As was described in Chapter 19 section 10, when a Thread Manager creates a new thread, a thread capability is returned to the caller. This can be used in inter-module calls to invoke semantic routines of the TCM. In the simple case, for example, the latter might provide operations for suspending and resuming the thread in question. The TCM is privileged in that it can directly invoke methods of the UTS. But since the UTS can vary from node to node and since TCMs can be programmed differently for different processes, it is impossible to define all the possibilities in more detail. The important point, however, is that this design leaves open considerable flexibility for managing processes and their threads. Furthermore, it is important that at the interface level a standard set of routines (e.g. for use by the synchronisation library and TCMs) is provided. The relationship between the TCM and synchronisation mechanisms was discussed in the previous chapter.

4 Managing Real Interrupts in the Kernel

Various CPU (hardware) designs can include a variety of support mechanisms for assisting in the management of interrupts from the hardware into the kernel, with various levels of complexity. Here we take the simplest approach, viz. that all local interrupts cause the CPU to start executing at the same location in the main memory and that they provide the details necessary to process the interrupt. The UTS does not see these (real) interrupts, as they are initially handled by the kernel.

Following Rosenberg's approach in MONADS I (see [5, pp. 154-155])[58], interrupt handling in SPEEDOS is performed by a short section of code (in SPEEDOS called the *interrupt analysis routine*). The kernel immediately turns off (real) interrupts (if the hardware does not already do this) then stores into a memory area in kernel space the values currently held in the registers (i.e. the register values of the interrupted user thread or kernel process)[59]. Kernel processes[60] have access to the stored registers and may further copy them to an appropriate place (e.g. into the stack of the interrupted thread).

Normally an interrupt leads to the activation of a kernel process (see section 6) and an immediate reschedule of these processes to ensure that their priorities are respected. They synchronise with each other using a variant of set semaphores which leads to the efficient "automatic" scheduling technique described in section 7 below.

Immediately an interrupt occurs the working registers are stored, and can then be used by kernel processes to carry out their work.

At the kernel level there are two interrupt categories, known as *synchronous* and *asynchronous* interrupts.

4.1 Synchronous Interrupts

Synchronous interrupts occur as a result of some event or action caused by the executing user thread or kernel process. Along with asynchronous interrupts they are initially analysed by the kernel's interrupt analysis routine, which classifies synchronous interrupts into the following four groups:

a) kernel instructions (e.g. inter-module calls). The kernel's interrupt analysis routine passes these interrupts to a kernel process called the User Request Process, which has a relatively low priority (compared to other kernel processes).

b) interrupts which require direct action by the kernel and have one or more associated kernel processes. The kernel's interrupt analysis routine passes such interrupts to an appropriate kernel process. (Virtual Memory page

[58] Rosenberg's thesis describes the kernel for the initial MONADS system, based on a modified HP2100A system, later known as MONADS I. This thesis has unfortunately not been widely published. Rosenberg and other members of the MONADS team later developed the MONADS-PC system. F. A. Henskens has documented parts of the MONADS-PC kernel in his own thesis [20] (see esp. Chapter 4 and 8), which is available at www.speedos-security.org.

[59] In a multiple CPU system there is such an area for each CPU if both can take interrupts.

[60] Since kernel processes are organised quite differently from user processes and their threads, we use the term process to distinguish the former from user level threads. (From the context it should be clear that these should not be confused with user level processes, see chapter 19 section 10 and chapter 20 section 2.)

fault interrupts fall into this category. In this special case the actions taken by the page fault kernel process are described in chapter 23.)

c) security sensitive error interrupts, regardless whether they are caused by programming errors or deliberate attempts to break the system. Attempts to violate the restrictions imposed by segment registers (e.g. attempts to access memory outside the range of a segment's data partition or to violate the access mode defined by the segment register) fall into this category.

d) non-sensitive program errors (e.g. divide by zero, arithmetic overflow, etc.)

Cases c) and d) are handled as *forced inter-module calls* on the top of the faulting thread, as described in section 9.

4.2 Asynchronous Interrupts

Asynchronous interrupts (e.g. an I/O or clock interrupt) occur independently of the currently running user thread. Usually the handling of such interrupts leads initially to the unblocking of a kernel process (e.g. an I/O interrupt process), which may then cause a pseudo-interrupt into the UTS to advise it that a user thread waiting for the interrupt can be unblocked. The latter is then moved to the ready state (see volume 1 chapter 8). Whether it is immediately selected to run is determined by the UTS's algorithm. Some asynchronous interrupts are handled internally in the kernel (e.g. by a disc process when a disc interrupts).

5 Kernel Instructions

The kernel *never* executes its instructions as a genuine in-process call, because the kernel is neither a normal module nor a library module. Instead it supports a number of kernel processes which are activated as described below.

The stacks for these processes, like other purely internal temporary kernel data structures, are held in a *non-persistent memory* which is logically separate from the persistent memory which the kernel and its co-modules create and manage for the user level. The use of non-persistent memory by the kernel is important in order to allow a kernel to be fully or partially replaced (e.g. to correct errors, provide more facilities or improve performance) when the system has been shut down, without concern for the state of internal data structures.

Protection of those kernel instructions which are not intended for general use is achieved in that a *kernel capability* must be passed as an operand of the instruction. This contains an identification of the node on which it can be used (because a kernel instruction cannot be executed on a node other than the node on which the active thread is currently executing) and a set of access rights corresponding to the kernel instructions which the holder of the capability may validly call.

In order to avoid many problems which have arisen in earlier kernels[61], kernel instructions in SPEEDOS are implemented *atomically* on a kernel stack (per CPU) in the non-persistent main memory. In this context *atomically* means that a kernel instruction always either completes without blocking or "fails" (because it cannot immediately complete the requested action). If it fails, for example if during the execution of an inter-module call a page fault occurs, the kernel resets the state of the thread to a position immediately *before* the execution of the kernel instruction. Consequently when the thread is later reactivated (e.g. after the missing page has been brought to the node and can be accessed by the thread) it will repeat the instruction.

In this environment the kernel needs only one kernel instruction stack per CPU in order to process kernel instructions. This can be re-used for each kernel instruction executed on that CPU without losing parallelism. Espenlaub points out the considerable advantages of this approach in comparison with the mechanisms used in other kernels [4, p. 170].

Throughout the book the simplifying assumption is made that the node under discussion has only a single CPU, since the handling of multiple CPUs depends heavily on the nature of the hardware itself.

6 Kernel Processes

The approach adopted for the management of kernel processes differs in Espenlaub's suggestions for SPEEDOS from Rosenberg's original MONADS design. It is instructive to compare both approaches.

6.1 Rosenberg's MONADS Approach

Although an ardent supporter of the in-process approach to process structuring at the user level, Rosenberg adopted an out-of-process structure for MONADS kernel processes. In his solution there are a fixed number of threads which have different (defined) tasks to perform. These threads are organised as a priority hierarchy. Higher priority threads execute before lower priority threads. Interrupts are transformed into messages. Each thread has its own buffer into which other kernel processes place messages. Reschedules take place when a new interrupt arrives and when a thread completes the processing of a message. Kernel processes can pass messages to other kernel processes and activate them. Typically the messages are passed between related processes in a standard *message block*. These can not only serve as a vehicle for new messages but can contain a record of progress so far in dealing with a request. One such message block type is the virtual memory message block, which will be introduced in chapter 23 to

[61] For a comprehensive description of kernels in other research systems see [4], chapter 3.

record the progress of page faults[62].

The highest priority kernel code is the interrupt analysis routine. In contrast with the remaining kernel processes this has no standard input buffer, as its inputs are interrupts. It is activated as a result of a (real) interrupt, and immediately turns off (real) interrupts until it completes, when it turns (real) interrupts on again. It handles trivial interrupts directly (e.g. buffering of characters for an unbuffered device, requests for the date and time). Where appropriate it places messages in the buffers of the lower priority threads. These are used, for example, to handle disc interrupts and page faults.

The lowest level priority processes are for handling the requirements of user level processing. User level threads execute within a single kernel process[63], the User System Process, which is the second lowest priority process (see Figure 22.1). The MONADS user thread scheduler (UTS)[64] executes in this slot; it manages all user level threads, including decisions to switch threads (which are actually carried out by a kernel instruction, cf. `thread_switch` described in section 2 above).

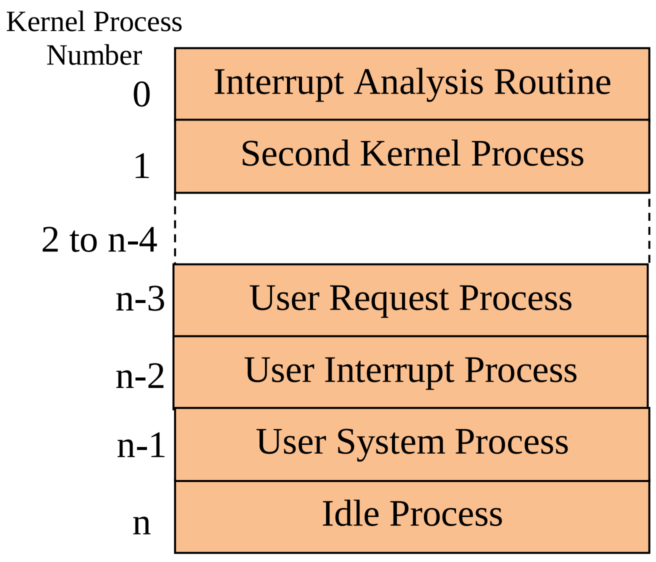


Figure 22.1: The MONADS Kernel Process Table

The third lowest priority thread is the User Interrupt Process, which receives relevant requests from other kernel processes (in the form of interrupt message blocks) and transforms these into pseudo-interrupts for the UTS in a

[62] An example which will play an important role in SPEEDOS is the IMC message block, which will be introduced in chapter 24 to record the progress of inter-module calls.

[63] All the MONADS systems were single processor systems. In a tightly coupled multiprocessor system there would be one such thread per CPU. (Other modifications would also be necessary to the system as described here in order to synchronise multiple CPUs. The details would depend on the nature of the hardware in question.)

[64] This is called the process scheduler in MONADS literature.

shared buffer. (From the viewpoint of the user level scheduler these appear to be real interrupts but in fact they are simulated pseudo-interrupts. To achieve this, the User Interrupt Process saves the current state of the User System Process at a fixed location available to the UTS and replaces this with a state which imitates an interrupt handler.)

The fourth lowest priority process is the User Request Process, which is used to execute kernel instructions for user threads. The very lowest priority process is an 'idle' process, which in the absence of an IDLE instruction simply loops doing nothing useful. This can run when the UTS issues a kernel `idle` instruction once the kernel also is inactive.

As will be explained in section 7 below, the actual kernel process scheduling mechanism in MONADS was based on an efficient microcoded implementation of set semaphores, where resource sets were used to access the message buffers and an extension of waiting process sets were used "automatically" to schedule the kernel processes [5, p. 155ff].

6.2 Espenlaub's SPEEDOS Approach

In his proposal for the SPEEDOS kernel, Espenlaub abandoned some aspects of the MONADS scheme, but retained the idea of supporting a separate thread or group of threads for each potential interrupt source [4, p. 171] (which might be a single device or a group of devices, depending on the hardware design). The fundamental difference between Espenlaub's and Rosenberg's schemes is that in the former the threads for handling asynchronous interrupts are persistent and are managed as normal threads scheduled by the user UTS, whereas in the MONADS approach interrupt processes are scheduled directly by the kernel and automatically have higher priority than application threads.

Espenlaub's design introduces more flexibility (e.g. by allowing new interrupt threads to be introduced at a later point into a system as a result of users introducing new devices into the system) but the cost of this flexibility is a time penalty in the thread scheduling activity (e.g. because the core kernel must interact with the UTS to activate the interrupt threads and because of the need to interact with Thread Control Managers).

6.3 The New SPEEDOS Solution

The differences between the two approaches introduce a dilemma into the final design of SPEEDOS, because in modern general purpose systems (including desktop and laptop computers) it is important to be able to introduce new devices (e.g. new external discs or printers), which could be supported in Espenlaub's concept by creating new persistent threads outside the kernel for handling additional asynchronous interrupts. However, in section 10 we introduce an alternative design which allows new device drivers to be introduced into a running system while following the MONADS approach. Furthermore, efficiency must also be a key criterion when making decisions about thread scheduling, since the UTS may be activated many thousand times per second. Not only is MONADS more efficient, as described above, but an extremely efficient implementation is possible using the concept of *waiting process sets* [19] (see chapter 21 section 4.2.) This implementation is described in more detail below.

A compromise between the two can be achieved by implementing asynchronous interrupt processes as a MONADS-style priority list (known in SPEEDOS as the Kernel Process Table (KPT) (see Figure 22.2), but also providing a security-sensitive co-module outside the core kernel, the Kernel Process Manager (KPM), which manages information relating to the kernel processes. The KPT has a similar structure to that of the MONADS kernel process table (cf. Figure 22.1).

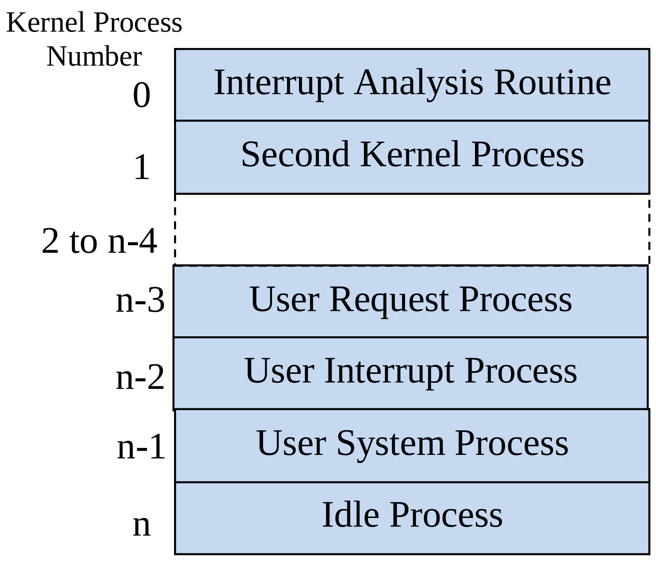


Figure 22.2: The SPEEDOS Kernel Process Table

Each entry in the KPT holds the information needed to activate and schedule its process. In particular it contains a pointer to the code which the process executes, and storage space for its registers, as well as a pointer to its input buffer and the semaphore which regulates its access to the buffer.

The Kernel Process Manager (a privileged co-module held in a container for kernel modules) is responsible for creating processes to handle interrupts (for the kernel) and for entering these in the KPT. But the core kernel schedules these and they are invisible to all other software above the kernel. This solution retains the flexibility of Espenlaub's proposal but also the run-time efficiency of Rosenberg's MONADS design.

The Kernel Process Manager also maintains further information needed by kernel processes, known as the Kernel Process Information (KPI), as will be discussed later in the chapter.

7 Scheduling Kernel Processes Automatically

Waiting process sets provide a technique for scheduling a set of processes which use enhanced semaphore operations to claim a resource or to schedule a set of resources. The basic principles behind both resource sets and waiting process sets were already described in chapter 21, section 4.

7.1 The Automatic Scheduling Mechanism

Waiting process sets can be taken a step further, as is described in Rosenberg's PhD thesis [5, pp. 155-162, 19, pp. 146-150] and was successfully implemented in the MONADS systems. By combining waiting process sets and resource sets and using them in conjunction with a system-wide "ready process set" (RPS, a bit list identifying all kernel processes which are waiting for a CPU) and an integer "current process" (CP), it becomes possible to schedule threads "automatically". In this context "automatically" means controlling the synchronisation and scheduling of threads entirely by semaphore instructions, which were implemented in MONADS in microcode[65].

This is achieved via a modified implementation of `WSetP`/`RSetP` and `WSetV`/`RSetV` instructions (called `ASetP` and `ASetV`, where A indicates "automatic"). The `ASetP` and `ASetV` instructions combine modified `WSetP` and `WSetV` operations with resource set instructions. Each instruction has two operands. The first is the address of a semaphore (here called SEM) which controls one of the kernel's process buffers. The second is an integer (here called R) which contains information about the appropriate resource. We refer to the integer part of the semaphores as SEMINT and the set part as SEMSET. Two global variables are required:

i) RPS (Ready process set) is a set containing a bit for each kernel process. If the bit is set it indicates that the corresponding thread is ready to execute, if unset that it is not ready.

ii) CP (Current Process) is an integer identifying the currently executing process.

A flowchart of these operations appears as Figure 14 in the published paper [19, p. 149]. However, this does not include the use of resource sets to enable the thread to discover which resource has been allocated or deallocated. We add this information in the following program snippets, where R is an integer operand

[65] It would be feasible in more modern systems to implement the semaphore operations efficiently in a combination of hardware and kernel software.

which is used to indicate the number of a message in a bounded buffer.

```
(ASetP)
SEMINT = SEMINT - 1;  // claim a message to be processed
if SEMINT ≥ 0     // if message(s) available in buffer
         {R = findbit(SEMSET);// find position of message and
                             // return it in the operand
          SEMSET = SEMSET – {R};}
                        // make message unavailable
                        // and continue executing

else {SEMSET = SEMSET + {CP}; // add waiting process
                           // to the set of waiting processes
      RPS = RPS - {CP};  // remove current process
                       //  from ready set i.e. wait
      reschedule;} // reschedule selects another thread
                  // from RPS to run and activates thread

(ASetV)
SEMINT = SEMINT + 1; // release a message position in buffer
if SEMINT > 0 {SEMSET = SEMSET + {R};}
   // if no processes waiting, free released message position
   // in buffer and continue executing
else     {chosen = findbit(SEMSET); // find a waiting process
          RPS = RPS + {chosen}; // add it to ready threads
          copy R into register of chosen;
          reschedule;}
```

The *findbit* operation searches a SEMSET bit list and returns the integer position of the selected bit. In other words in this context it either selects a message to be processed (ASetP) or it selects a process to be activated (ASetV).

The *reschedule* operation selects a process from the RPS and activates it. It is defined as follows.

```
(reschedule)
integer selected = findbit(RPS);
if selected ≠ CP {switchRegisters(selected); CP = selected;}
```

Each kernel process uses a set of registers to carry out its defined activities. However the switching of processes would be an expensive activity if *all* the segment registers and general purpose registers in SPEEDOS systems were to be available to these processes. Consequently, with the exception of the User System Process, which is the thread in which the user threads are executed, each kernel process is restricted to the use of a smaller number of general purpose registers and a few segment registers. In this way the switching of kernel process registers can be made more efficient.[66]

One further feature needs to be added to the automatic scheduler mechanism to allow it to become fully functional in the SPEEDOS concept, viz. a

[66] A final decision about which registers are available to kernel processes is left open here.

mechanism to allow a kernel process to suspend itself during the execution of its algorithm. This was already included in the MONADS-PC system, in particular to allow the disc process to start a disc access (read or write) and suspend until the access has completed [20, pp. 159-160]. We here call the suspend instruction `AsusP` and the corresponding activate instruction `AactP`. `AsusP` has no explicit operand; it always suspends the kernel process executing the instruction and causes a reschedule. `AactP` has a single operand defining the number of the thread to be activated.

These are implemented via an additional word SPS (suspended process set), in which, like RPS, each bit represents a kernel process number. When a process issues an `AsusP` instruction, the bit position corresponding to CP is set and a reschedule is then started. When some other process (e.g. the Interrupt Analysis Routine) wishes to activate a suspended process (e.g. a disc process) it uses `AactP` (providing the number of the process to be activated). This unsets the bit in SPS corresponding to the process number and starts a reschedule. The reschedule instruction itself must be modified to take account of suspended bits in the SPS. This is most easily achieved by modifying the operand of `findbit` in the reschedule operation into an exclusive or of RPS and SPS, i.e.

```
(reschedule)
integer selected = findbit(RPS xor SPS);
if selected ≠ CP {switchRegisters(selected); CP = selected;}
```

Notice that a process number in RPS must always be set if the corresponding bit in SPS is set.

This automatic scheduling mechanism is extremely efficient, eliminating the need for a kernel process scheduler in software[67] and at the same time it very efficiently solves synchronisation problems. Why then is it not widely used? One reason is the limitations described in section 4.3 of chapter 21, which provided the reasons for not using waiting process sets *at the user synchronisation level*. Furthermore, the operations which it supports are quite primitive (e.g. it is not possible for a process to wait on the union of several conditions such as a timer interrupt or an input from a terminal).

Automatic scheduling in this form requires that the number of kernel processes should be quite small (e.g. not more than 64 in a system with 64 bit words) if the set operations are to be kept efficient. Furthermore processes should be fairly static (i.e. adding and removing processes during a running system should be avoided if possible), so that bit positions corresponding to process

[67] The automatic scheduler was implemented in microcode in MONADS, and it might be feasible to implement some parts of it in hardware as the basis for a priority interrupt system.

numbers do not become ambiguous.

7.2 The Scheduling Algorithm

The most significant limitation of the automatic scheduling technique is that only two scheduling policies are easy to implement. If the search for a new process for execution (i.e. the `findbit` function) always begins at bit zero in RPS and in `ASetV`, choosing the first position where a bit is set, the result is a priority scheduling algorithm, with the thread corresponding to bit 0 having the highest priority and the thread corresponding to the last bit having the lowest priority. Since the synchronisation of the kernel processes relies on a priority system, this is ideal for the SPEEDOS kernel. But it is far from ideal for user level threads, which is one reason why these are scheduled by a separate scheduler, i.e. the UTS.

The only easily implementable alternative would be to have a cyclic pointer indicating the bit position at which the last search ended, thus providing a FIFO buffering mechanism. This would be a feasible alternative way of implementing the `findbit` algorithm in `ASetP`, but in view of the added complexity and the nature of the SPEEDOS kernel the priority version is to be preferred.

In the final SPEEDOS system, as in MONADS, priority scheduling is preferred not only because of its simplicity but also because this simplifies the design of the kernel processes with respect to their synchronisation with each other.

7.3 Managing the Buffers

There are central pools of message blocks, which are passed via pointers between processes. These both serve as parameters for the kernel processes and can hold further relevant information which can in effect be stored as long as it is needed and used by several related threads. The `ASetV` operation is used by the sending process when a message is passed from one process to another. A pointer to the relevant message block is placed in the input buffer of the receiving process, which in effect is a producer-consumer buffer[68]. The process will eventually be scheduled and will use the `ASetP` operation find the first available message in its buffer. When it has completed the processing of the message it loops back to the `ASetP` instruction to process the next message. If there are no messages in its buffer the `ASetP` operation causes it to wait until a message arrives. A thread may be temporarily halted as a result of a reschedule (and of course its register values are preserved until a further reschedule activates it). It will resume (and its registers reloaded) when a further reschedule selects it as

[68] see Chapter 8 section 12.1 (Bounded Buffers).

the currently highest priority thread.

There are two exceptions to this scheme, as follows.

– The interrupt analysis routine has no buffer but is activated by an actual hardware interrupt, then after analysis it places a message resulting from its analysis into the buffer of the appropriate thread.

– Disc processes do not simply consume messages passed to them in the order of their arrival but scan the buffer to achieve a more efficient use of the disc which they are controlling (e.g. to minimise the disc head movements[69]). This is disguised from the rest of the kernel by having a special form of the `ASetV` instruction for each disc process.

7.4 Passing Interrupts to the User Thread Scheduler

Kernel processes frequently pass on interrupts to the main operating system, which in practice means that they advise the User Interrupt Process by passing a message to it. As this process has a higher priority than the User System Process, it cannot be interrupted by the latter, which has the lowest priority except for the idle process. Several user pseudo-interrupts might gather in its buffer as a result of interrupts being passed on to it from higher priority kernel processes. These are serviced one at a time by the User Interrupt Process.

In order that the UTS can manage these in an orderly fashion, a protected mechanism for enabling and disabling (pseudo) interrupts at the user level, implemented via a binary semaphore, is provided (as in MONADS). This does not affect real interrupts, but only the interrupts from the kernel to the UTS. The protection is provided in that the caller must also present an appropriate kernel capability.

Before passing an interrupt to the UTS the current state of the currently active user level thread (i.e. the current state of the User System Process) is saved, in order that it can be resumed by the UTS later.

In MONADS the User Interrupt Process had three "registers" (which might be implemented in memory: an Interrupt Target Subsystem[70] (ITS) register, an Interrupt Target Address (ITA) and an Interrupt Parameter Pointer (IPP). The first two defined the module to be interrupted and the address at which the interrupt should be made. IPP points to an area in which the parameters (message) should be placed.

In SPEEDOS equivalent information, i.e. a module capability (for the UTS) and an entry point number in the module for the interrupt routine, can be placed in the KPI, and the current user-level Thread Stack can be used to pass the pa-

[69] e.g. the elevator algorithm, see https://en.wikipedia.org/wiki/Elevator_algorithm
[70] In MONADS modules (in the SPEEDOS sense) were called subsystems.

rameters (message) to the UTS. In this way it becomes a kernel design decision whether the same interrupt routine is used for all interrupts or whether several UTS interrupt routines are provided to handle different kinds of interrupts.

When an interrupt routine completes, a reschedule will usually be needed at the UTS level (e.g. if the pseudo interrupt signalled the completion of an input-output operation). To achieve this, the interrupt routine ends by executing a reschedule operation, which selects the next thread to be run. It then executes a `return_thread_switch` kernel call (see section 1 above), which causes the kernel to combine an inter-module return with a thread switch to the chosen thread. Of course the kernel had already stored the state of the thread which it interrupted at the base of its thread stack.

8 Kernel Interactions with Co-modules

The synchronisation mechanisms described in chapter 21 can be used by user-level threads to synchronise their activities with each other and, as we have seen above, to co-ordinate the interactions of kernel processes with each other. But there remains one issue to be answered. How can the kernel synchronise with those co-modules with which it shares data? And how can it interact with user level modules? We examine these two issues in this section.

8.1 Sharing Co-module Data

As was described in chapter 17, the kernel makes direct use of data of its co-modules (e.g. from the Co-module Tables, Code Tables and Thread Tables). It only reads the information in them[71]. Is it possible that while it is reading information from a data structure shared with a co-module that this could interfere with a user-level thread which also wishes to access the same data?

Recalling that this chapter is only concerned with synchronisation at a single-CPU node[72], we realise that the same CPU cannot at the same time be executing both at the user level and at the kernel level. Suppose now that the kernel attempts to coordinate with the user threads not by actively using shared reader-writer semaphores but merely by looking at their state, assuming of course that the kernel knows where these semaphores are (e.g. at the beginning of each shared co-module data structure). One of the following possibilities arises:

a) The semaphore shows that readers are active. In this case the kernel can also safely read the data. Since the kernel process involved (i.e. the User Request Process) is executing at a higher priority than the User System

[71] It writes information (e.g. parameter and linkage segments) on a thread stack but this does not conflict with actions of the co-modules.

[72] because there are several possibilities for designing synchronisation between multiple tightly coupled CPUs, which cannot all be discussed here.

Process (in which user-level threads run) it can also safely read the data, without adding itself to the set of current readers. Although it can be interrupted by higher priority kernel processes, it will always resume before a user-level thread can run; hence in this situation there is no conflict.

b) The semaphore shows no readers and no writers. Here too in a single CPU system the kernel process, executing safely at a higher priority without registering itself as a reader.

c) The semaphore shows that a writer is active. In this (seldom occurring) case, the kernel simply sends a request to the user level UTS, telling it to reschedule. Since the program counter of the user-level thread has not been updated the user-level thread would simply re-try the kernel instruction when it is later selected to run by the user level UTS.

Since the kernel *never writes* shared co-module data, this possibility simply cannot arise, and hence we have a trivial solution for the problem in a single CPU system, without all the complexities created by Espenlaub's solution [4, pp. 172-173].

8.2 Surrogate Threads

Sometimes the kernel needs to interact with the user system level not simply by reading data, but by executing an algorithm involving non-kernel code. To achieve this it can activate a *surrogate thread.* Such threads are designed to carry out special tasks on behalf of the kernel. In some cases they carry out tasks on behalf of the kernel, e.g. helping with the login activity (see section 11 below), but sometimes they assist with the execution of user activities in special situations such as executing bracket routines (see chapter 24) or assisting with the execution of remote inter-module calls (see chapter 28).

Surrogate threads are prepared at system initialisation and can be activated when needed by the kernel. They can be implemented as follows.

a) For each kernel service a pool of threads is set up in its own process container as part of the system initialisation. The kernel is then advised how many such threads (i.e. thread stacks) have been created and in which container they are held.

b) The kernel maintains a bit list for each service pool in which each bit corresponds to a thread. It can then treat this as a resource set and so quickly select a currently unused surrogate thread and determine which thread has been allocated.

c) The threads are identified in the system via thread capabilities which as usual contain the node number, the process container number and the thread number. However the type field in a thread capability for a surrogate thread is not "thread" but "surrogate thread".

d) As part of the system initialisation the Thread Manager for each surrogate thread pool creates the corresponding number of threads and in these builds up an appropriate addressing environment. Typically this involves setting up values such as the following:

   – images of the kernel pseudo-register values and segment registers (e.g. segment register 5 and if the routine needs parameters, segment register 0 as well as other relevant registers such as the Thread Security Register, as appropriate[73];

   – appropriate values in the capability accessibility area.

   – parameter values in the input parameter segment.

e) Not all values can be set up at system initialisation time and some may need to be set up immediately before the kernel activates a thread (e.g. the code segment register and program counter to address the code[74]). What can actually be set up at what time depends on the individual cases. Similarly some registers may need to be invalidated when a surrogate thread completes a task.

e) When the addressing environment is fully prepared the kernel can activate the thread by causing an interrupt into the UTS, passing to it a surrogate thread capability, as an identifier. The UTS does not keep a complete list of surrogate threads, but adds them when advised by the kernel and deletes them when advised by the thread.

f) The UTS schedules them using the normal kernel instruction `switch_thread`, providing the thread capability (and the usual kernel capability) as operands.

g) When a surrogate thread has completed its task it calls the normal UTS routine (**`killMe()`**). This then uses the kernel instruction `switch_delete` which advises the kernel that the current (in this case surrogate) thread should be deleted. The kernel can determine from the type field in the thread capability that the thread to be deleted is a surrogate thread, in which case the kernel returns the thread to its deallocated status.

This may sound like a complicated mechanism but it is in fact simpler, more efficient and more flexible than carrying out a normal inter-module call, for example, or trying to implement the forced module calls suggested by Espenlaub A first example of how surrogate threads are used in practice is provided in the section 11.

[73] see chapter 26.

[74] These cannot be part of the system initialisation, since they change dynamically as each thread executes.

9 Handling Synchronous Interrupts

In section 4 four kinds of synchronous interrupts were listed. The first of these (kernel instructions) is straightforward to handle, as we have seen above, in that the interrupt analysis routine transforms these into messages placed in the buffer of the User Request Process. Now follows a brief discussion of the other three cases.

9.1 Page Faults and Related Interrupts

In chapter 17 section 3 we referred to Espenlaub's suggestion which would allow the kernel to make so-called 'forced method calls'[75] to interface routines of its co-modules and to examine their returned results in order to obtain information which it might need. This suggestion is by no means as simple to implement as it may sound and has therefore been rejected as a general technique for the final SPEEDOS system, although it would theoretically be useful, for example, to call an interface routine of a container's Virtual Page Table Manager in the course of resolving a page fault. However, we will see in chapter 23 that there is a much more efficient way of achieving this particular requirement.

9.2 User Errors and Security Violations

An error in the execution of a user thread (see cases (c) and (d) in section 4.1 above) implies that normal execution of the thread cannot immediately continue. Hence there is no problem in handling the error via an inter-module call to the appropriate error handling module on the faulting thread stack.

In this case the kernel's interrupt analysis routine places details of the interrupt into the input buffer of the kernel's User Request Process. This saves the thread's status (from the kernel analysis routine's register save area) as it would if the UTS had issued a kernel `thread_switch` or an inter-module call instruction. It then creates a new stack frame (including input parameters based on the interrupt details) at the top of the thread stack. The kernel process then "fakes" an inter-module call to the required entry point of the co-module to be activated, which may differ for different errors, but should always be a co-module with a fixed index number in the current container.

For non-sensitive program errors this could, for example, be a white box debugger co-module (at a fixed position in the co-module table, thus allowing the kernel to generate an appropriate module capability to fake the co-module call).

For protection violations the co-module called is always a system co-

[75] This should not be confused with the mechanism for passing interrupts to the UTS, discussed in the previous section, nor with the forced inter-module call.

module, which first carries out security related tasks such as logging the error and might then optionally be programmed to activate a user defined module on the user thread stack, e.g. to carry out debugging (as described in (c) below).

When the stack has been fully prepared, the kernel then starts its execution (as for a normal inter-module call). The user level UTS does not need to be informed of this, because it was not informed of the interrupt and so assumes in any case that the thread is executing.

On completion of the error handling module it might be necessary to advise the Thread Control Manager to delete the current thread, or if the problem is recoverable, the kernel can arrange an exit back to the thread state as it previously existed.

One potential danger with this approach was mentioned by Espenlaub, recalling that the issue of stack overflow might occur as in systems such as the Burroughs B6700 (see chapter 8 section 8). However, with the availability of vastly more main memory and much larger virtual addresses, together with the paging design presented in chapter 23, this issue is no longer relevant, provided of course that precautions are taken to avoid it.

10 Handling Asynchronous Interrupts

The kernel receives interrupts which are not directly related to the kernel process or user thread) which is currently executing. The most important of these interrupts fall into two groups: general device interrupts (e.g. from printers, from keyboards) and disc interrupts.

Of the remaining asynchronous interrupts only normal asynchronous interrupts from hardware devices are now discussed, since in chapter 23 we will discuss disc interrupts (which are related to the virtual memory).

10.1 Espenlaub's Proposal for Handling Asynchronous I/O Interrupts

For handling asynchronous interrupts arising from device completions and similar cases Espenlaub suggests handling these almost entirely outside the kernel, with the cooperation of the UTS in an *out-of-process* style. This implies that such user level threads contain device drivers which must often send messages to other user threads that are using the device, which therefore requires a facility to do this. My own intuition is that device drivers are best handled in as *in-process* modules in the user thread which wants to use them. This means for example that module capabilities can be used in the normal way to determine whether a user thread has the right to activate a device. We therefore now explore this possibility for handling asynchronous interrupts.

10.2 Handling Input-Output Operations In-Process

The basic idea is based on the fact that asynchronous interrupts usually arise as a result of a user thread requesting an input/output (I/O) operation. The user thread may then wait for the I/O request to be completed. This occurs in the form of an asynchronous interrupt, and the user thread must then be reactivated. (The only significant exception is the management of memory device (e.g. disc) interrupts, which are normally part of the SPEEDOS virtual memory and are therefore a special case, discussed in chapter 23.)

10.2.1 In-Process Device Drivers

The main issue which concerns us at this point is what part the kernel plays in the above pattern. The most important question is how device drivers fit into the pattern. These are special hardware modules (usually supplied by device manufacturers), which contain the detailed knowledge needed to "drive" the device, i.e. to make it function correctly.

In conventional computers there are two ways of achieving this, depending on the hardware design. One is a special hardware instruction (here called 'start device'), which can only be executed when the computer is currently in a privileged mode (corresponding to SPEEDOS kernel mode). Alternatively, the hardware might use memory mapped I/O. (For more detail see volume 1 chapter 6.) In SPEEDOS memory mapped I/O is strongly preferred, since in this case the only special privilege required is that a segment register is loaded to address the device memory. To achieve this, the device driver is "simply" a module which is invoked by a normal inter-module call and thus executes in the thread of a user wishing to use the device. The device driver, when invoked, uses the kernel instruction `load_devSR` in order to gain access to the device in question, specifying as operands the number of the segment register to be loaded, a device capability which defines the device via a channel number and device number and a normal kernel capability authorising the caller to call this kernel instruction. The device capability provides evidence that the current thread/module is authorised to use the device, and at the same time identifies the device for the kernel.

The main question remaining (in this context[76]) is how the deactivation and reactivation of the user level thread are organised. A kernel device process first becomes aware of a user thread's need to use a device when a device driver in a user thread requests the loading of a segment register via the kernel `load_devSR` instruction. This can then be the used to access the device via memory mapped I/O. When the thread reaches the point at which it needs to wait for the result of

[76] Issues such as organising the use of devices and spooling of printer output are questions for the operating system design, which will be discussed in chapter 33.

an I/O operation, it calls the Thread Control Manager for the thread in which it is active, requesting this to call the user level UTS to suspend it pending an interrupt from the corresponding channel and device number. To ensure that the interrupt will eventually be passed to the correct thread, the UTS executes the kernel instruction `wait_interrupt`, which passes a copy of the device capability to the kernel. (This enables the kernel's device process to maintain a temporary list of thread/device combinations waiting to be activated, called the Waiting Thread List.) The UTS then suspends the thread, waiting for the kernel to interrupt it (using the usual pseudo-interrupt system, see section 7.4).

10.2.2 Handling the Interrupt

Eventually the kernel will receive an asynchronous interrupt from the device, which it passes to its device process. This must simply check the result from its Waiting Thread List, delete the matching entry and create an appropriate buffer entry for the kernel's User Interrupt Process, which will eventually cause the user level UTS to be interrupted. This in turn can reactivate the thread waiting for the device. The latter can then continue to re-use the device following the same pattern, until it either invalidates its device segment register or it exits from the device driver.

10.2.3 Activating Related Threads

There are some asynchronous interrupts which require a small extension to this scheme, i.e. when a thread that receives an asynchronous interrupt needs to activate other threads which also have an interest in the interrupt and need further information about it. In this case there is no problem in activating another thread which is waiting, but the mechanism for activating threads (see chapter 21 section 1) provides no facility for passing on new information to the activated thread. For this purpose the kernel provides an instruction `put_message` which allows the thread receiving the interrupt to pass a short message of one word (8 bytes) to the kernel, and a further instruction `get_message`, allowing the recipient to receive the message. To ensure that this cannot be used as a covert channel, the following precautions are taken:

- both instructions require a kernel capability;
- the `put_message` instruction requires that the user of this instruction provides a thread capability for the destination thread;
- the `get_message` instruction requires that the user of this instruction provides a thread capability for the sending thread;
- the `put_message` instruction requires that the user of this instruction provides a module capability for the destination module;

- the `get_message` instruction requires the user of this instruction to provide a module capability for the sending module;
- When the `get_message` instruction is executed, the kernel not only returns the message to the destination module, but also clears it to zero.

An example of the need for this mechanism appears in chapter 32 section 4.

10.2.4 Adding New Devices to a Running System

One of Espenlaub's aims was to make it possible to add new devices to a running system. This is achieved in the above alternative in that device drivers are still located outside the kernel, but instead of having special threads for this purpose they are simply treated as (in-process) modules in the thread using the device. Their only special feature from the user's viewpoint is that the user may need to provide them with a device capability as a parameter.

In this design a kernel process which receives a device interrupt can have a standard design which simply reacts according to a channel/device number.

When a device driver is installed, this follows the normal rules for installing a module. It is of course possible to install multiple device drivers in a single container and it may even be appropriate for them to communicate with each other or share data. In this case they can be installed as cooperating co-modules (see chapter 18 section 7). Device capabilities which give them the privilege to access I/O devices can in principle be stored in the constant segments of their code. However, since these serve as evidence that a thread has the right to use the device, it may be better to insist that device capabilities are passed as parameters to the device drivers, as is discussed in the next subsection.

10.3 Device Management

In order to use an external device a thread needs a device capability. Users must obtain such device capabilities from a source which has such a capability. This might for example be from the creating user when a new user is created, or it might be from an operating system module, or a system manager, or a central directory of device capabilities, etc. Which user processes and threads obtain the privilege to use external devices is a policy decision, not a mechanism and is therefore not part of the kernel.

10.4 Handling Interactive Interrupts

Interactions with modern keyboard-monitor devices are in principle similar to interactions with other devices, involving a device driver mechanism. In principle these can be handled like other input-output devices (see previous section), but there are several unusual issues. First, several (but not all) modules which are active in the same thread must have easy access to a device capability. Second, different threads may need access to the same device. Third, the question arises how the threads with a need to access the same device acquire the required capability, since there is no guarantee that a user will always log in at the same device.

If we make the assumption that each user logs into a separate device, and that this only displays information to which the user should have access, then the sharing issues are no problem. A mechanism has already been provided in chapter 19 section 5 which allows authorised modules in a thread to gain access to standard input and output modules for the thread, via a capability accessibility area[77]. Once a device driver capability is accessible to one thread it can share this with other threads, like any other capability. The more interesting question, from our present point of view, is how this device driver capability gets allocated and placed in the capability accessibility area.[78] The answer can be found in the next section.

11 User Commands to the Kernel

Sometimes a user may need to instruct or make requests to the kernel, which are comparable with operator commands in early systems. For example he may need to advise the kernel that he wishes to remove an external disc from the node. Such requests may involve several interactions, which are best organised initially outside the kernel. These are handled in SPEEDOS via normal threads owned by the system administrator, or in single-user systems by the owner of the computer. This thread communicates with the kernel by executing privileged kernel instructions and examining the responses (see chapter 17, section 6). To execute these instructions the thread needs appropriate kernel capabilities, which are provided with a new system in segments of the code which the appropriate thread executes.

12 Long Suspending Processes

Volume 1 chapter 15 section 3 explained how interactive persistent processes (i.e. threads in SPEEDOS terminology), are not destroyed when they log out (in contrast with conventional systems) but continue to exist in a "long suspended" state. Then when the user logs in again they return to the active state, continuing from the first instruction after the long-suspend call. This will typically be in a

[77] How some modules in a thread can be restricted from obtaining capabilities via the capability access area of a thread will be described in chapter 26.

[78] For those programmers who are only familiar with the out-of-process model and who are concerned about how the appearance of a screen can be coordinated for different threads/processes, the answer is simple. Multiple threads can share a module (with access coordinated by semaphores) that holds a representation of what is held on the screen!

routine of a user module which can then check the authenticity of the person attempting to log in.

The impression was given in volume 1 chapter 15 that the kernel is responsible for managing long suspends. However, that preceded the discussion on the structure of the SPEEDOS kernel and in particular the possibility of using security-sensitive co-modules to carry out kernel-related tasks. In fact, the core kernel plays only a relatively small direct role in managing SPEEDOS long suspends.

This is important because the kernel itself stores no persistent information. If the kernel is shut down and restarted, it must rely on information in its co-modules to enable it to re-start threads logged out at the end of an earlier system session. Hence it cannot rely on maintaining its own list of logged out user threads. Instead a co-module, the Login Service Module, maintains a list of logged out threads. A possible format for entries in this list is shown in Figure 22.3.

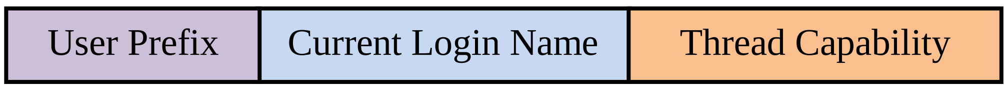


Figure 22.3: Entry in the Login Service Module's Logged Out List

The User Prefix is a unique "username" of up to 16 bytes. However, this is not normally used to login. Instead, the user supplies a current login name, also up to 16 bytes in length, which can be changed whenever he logs out. Hence a different login name can be used not only for each thread or user process but also for each session if desired. Since the current login name is not guaranteed to be unique, in the case of a clash the system can request the user prefix in order to disambiguate the name. This mechanism has the advantage that the user does not normally even have to reveal his username when logging in. The user prefix is supplied to the Login Service Module by a new user when his first process is created.[79] If he proposes a prefix which already exists for another user, he must choose a different one which is not already in use.

12.1 Logging Out

The "logout module" shown in Figure 15.5 and later diagrams of volume 1 chapter 15 can now be equated with the Thread Control Manager responsible for the process container of the logging out thread. It has a semantic routine, called `logout`, which is the logout command for threads in its container. This should only be called via a thread capability for a thread which wishes to long suspend.

[79] Creating new users is described in Chapter 31.

Depending on the parameters of the logout routine the calling thread alone, or a defined set of threads in the process or all the threads in the process can be suspended (and later re-activated).

Here we describe the case of a logout which involves only the currently active thread. Situations involving further threads and subthreads are left to an actual operating system designer, since they may involve the management of errors (e.g. if a thread has claimed a semaphore or is in a semaphore queue when its owner issues a logout command), which would by far exceed the scope of this book.

12.1.1 TCM Actions in Preparation for Long Suspensions

The TCM **`logout`** routine performs the following actions:

a) It carries out any necessary housekeeping duties, such as noting the logout time, etc. Via a parameter to the TCM's logout routine the thread/process owner is also given the opportunity to change the current login name.[80]

b) The TCM's **`logout`** routine executes a kernel instruction `invalidate_IO_cap` to invalidate the device capabilities for the current keyboard and screen devices.

c) The TCM invokes the **`logout`** routine of the node-wide Login Service Module, passing to it the old login name and optionally a new login name. Since it is possible to change the current login name each time the service module's **`logout`** routine is called, a hacker cannot even rely on the "username" part of a logged out process or thread[81].

   The Login Service Module's **`logout`** routine obtains a thread capability for the current thread (from the capability accessibility area) and makes an entry in its Logged Out List of suspended threads, noting the new login name in its list if appropriate, and returns to the TCM.

d) The TCM then invokes the UTS's **`suspend_me`** routine.

Note: all these actions are taken in the thread which requested the logout.

[80] A login name may include characters such as hyphen, full stop, forward slash, backslash, etc. From the viewpoint of the Login Service Module such names are simply viewed as a string to be matched. Only the character used by the kernel to recognise the end of a character string cannot appear in a login name.

[81] In case a user forgets the name, he can invoke a different semantic routine of his Thread Control Manager in a different thread. This can then recover the forgotten name for him without having to reveal this to a system manager or superuser, see Chapter 5 section 7.1. (In a mandatory protection environment (see Chapter 3) it is possible to provide a semantic routine for the Login Service Module which gives a 'superuser' access to its list of logged out users.)

12.1.2 User Thread Scheduler Actions for Long Suspending Threads

The UTS selects a new thread to be scheduled and calls the `thread_switch` instruction of the core kernel[82]. This instruction causes the current state of the thread to be stored at the base of its stack (as usual). At this point the logout is effective. From this we see that the core kernel has no direct role in logging out activities.

12.2 Logging In in a Multi-User System

The most interesting issue from the viewpoint of the kernel is how a logged out thread is reactivated. Espenlaub's description [4, pp. 200-201] is not detailed. It simply refers to a "node-wide login service" which is *out-of-process* and which, after establishing the login name from the user, calls the corresponding Thread Control Manager's **`login`** routine. This then calls the UTS to reactivate the appropriate thread(s). This sounds very plausible, but it leaves unexplained (a) how the login service thread(s) can be activated when a user wants to log in and how it discovers which terminal is active. It also leaves open the efficiency issue (for the case that in a large system multiple users attempt to log in at the same time). Here is a more detailed and somewhat modified, largely *in-process*, alternative.

12.2.1 Handling an Unexpected (Asynchronous) Keyboard Interrupt

A user normally signals his intention to log into a system by activating a free interactive device, which creates an unexpected asynchronous interrupt. The kernel's interrupt analysis routine will pass the interrupt to its device process. This detects from its Waiting Thread List that there is no process designated for handling this case, so the kernel puts a message (consisting of the device source) into the buffer of the kernel's internal "login" process.

12.2.2 The Kernel's Login Process

The primary purpose of the login process is to activate a surrogate thread in the privileged Login Service Module. This thread is responsible for obtaining a user name for the logging in user.

12.2.3 The Login Surrogate Threads

The kernel's login process uses a resource set semaphore to claim a free surrogate thread (see section 8.2). After it has prepared for the activation of the surrogate thread (including placing a device capability for the interrupting device into the thread's capability accessibility area), the login process creates an interrupt

[82] A 'sleep' parameter can be used by the kernel as an indication that the stack can be paged out. It otherwise takes no special actions with respect to logging out.

into the UTS, indicating (by passing its surrogate thread capability) that the thread is ready to be scheduled.

The UTS then schedules the surrogate thread (using the `thread_switch` instruction). The surrogate thread, executing in the Login Service Module, sends a login request to the device and suspends itself in the UTS awaiting a reply (see section 10.2). It will then need to synchronise its access to the Logged Out List via the usual reader-writer semaphore. If a search shows that the name provided is on the list it will remove the entry, release the semaphore, and execute the kernel's `transfer_terminal` instruction. This has three operands: a kernel capability allowing it to use the instruction, the device capability for its current thread, and the thread capability for the logged out thread which is to be activated. The kernel then stores the device capability in the capability accessibility area for the thread to be logged in.

The surrogate thread then calls the UTS **`killMe`** routine, passing the thread capability from the matching entry in the list as a parameter and an indication that this thread should be activated. When the UTS has done this it calls the kernel's `switch_delete` instruction and the surrogate thread is returned by the kernel to the pool of surrogate threads.

Since the kernel surrogate thread requires only few instructions its job is done very efficiently.

12.3 Logging In in a Single User System

In a single user system the start-up phase of the devices will cause the system to be initialised and then can request his log-in details without the above complications, but carrying out the same security checks.

12.4 Logging In (the User Thread Level)

After the UTS has activated the logging in thread, this returns back to the TCM's `logout` routine (which had been called as part of the logging out procedure). This continues to execute the second part of the `logout` routine, which can then carry out the following.

a) It performs housekeeping duties (e.g. note the login time).

b) It next calls the application's authentication module[83], which then challenges the logging in user to provide some evidence that he is the valid user. (This is possible because the kernel has already set up a capability for the interactive input-output device in the thread's capability accessibility area, see section 11.2.3 above.) There is no restriction on the form this check can

[83] A capability for the authentication module has previously been provided to the Thread Control Manager by the owner of the thread.

take. If the authentication is successful the Thread Control Manager then exits back to the calling module, which then continues as normal. It may allow repeated login attempts, but if the authentication finally fails it returns this information to the TCM, which again logs the thread out. The TCM can also provide a semantic routine which allows the user to change the authentication module by providing a new capability.

13 Scheduling Real Time Systems

Espenlaub argued that his kernel design better supports real time systems "which cannot tolerate deviations from the schedule", in that the UTS has complete control of scheduling both normal and interrupt threads [4, p. 171]. This could certainly be the case in some systems, but in many real time environments handling interrupts by priority is a satisfactory approach.

In the environment proposed above for SPEEDOS, real time systems which rely on priority scheduling can be supported effectively by adding the real-time threads to the list of kernel processes in the KPT, using the Kernel Process Manager to achieve this. Such threads will not be persistent, but like the kernel's own processes they will usually be "lightweight" processes and, like the kernel's own processes, they can be initialised by a user level co-module when the system is loaded.

SPEEDOS was not initially intended for use in real time systems, but there appears to be no reason why Espenlaub's concept could not be implemented in a variation of the standard kernel for systems which are schedule-based. Alternatively, and probably more efficiently, a variant of the kernel process scheduling mechanism described above in sections 6 and 7 could be implemented.

Chapter 23
Virtual Memory Management at a Single Node

This chapter describes one of the SPEEDOS kernel's most significant functions, the management of the virtual memory. In many respects this differs substantially from virtual memory management in conventional systems. The main reason for this is that SPEEDOS adopts a significantly different approach in terms of both the persistence of the virtual memory and its distribution over the internet.

Several aspects of the SPEEDOS design presented in this chapter are based on ideas which were successfully implemented in the MONADS kernel [21], the very substantial difference being that SPEEDOS "outsources" more complex kernel functions into security-sensitive co-modules.

In this chapter we make the simplifying assumption that all discs comprising the virtual memory at a node are always on-line at that node. In later chapters this restriction will be relaxed.

A small amount of information in this chapter is repeated from chapter 11, to save the reader from the need to frequently refer back to, or to re-read, that information, which is very considerably expanded in this chapter.

Appendix 1 of this volume provides a detailed overview and diagrams of the formats of relevant data structures from this and later chapters.

1 Hardware Translation of Virtual Addresses

So far we have presented a SPEEDOS virtual address as consisting logically of four 64 bit words, as is illustrated in Figure 23.1.

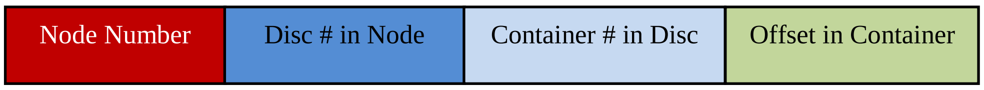


Figure 23.1: A SPEEDOS Virtual Address

The node number is a unique SPEEDOS identifier of a node. Each SPEEDOS computer has a distinct node number which is built into its hardware and can be read (but not modified) by the kernel or other software. In order to simplify the allocation of node numbers the first 8 bits of the node number represent a manufacturer number. However, this does not affect the calculations in this chapter.

The disc[84] number within a node is allocated by the kernel when it initialises a disc[85] and is unique within that node. In reality a physical disc can be subdivided into partitions. For this reason the last four bits of the disc number are used to indicate a partition number on disc (see chapter 27), hence reducing the number of physical discs which can be created on a node to 2^{60}.

The container number within a disc is allocated when a new container is created on a disc (see chapter 19); this is unique within the disc on which the container resides. The first 8 bits of a container number are an index field, indicating a module number (within data and code containers) or a thread number (within process containers)[86]. The next eight bits are also reserved; they hold a 3 bit type field, a one bit "valid" field and four status bits, all of which are only relevant in the context of capabilities and will be explained in later chapters. The effect of this is that only 48 bits are used to identify a physical container. This is significant in the present chapter.

The "offset in container" part of a virtual address itself decomposes into the pair «page # in container, offset in page», see Figure 23.2. The maximum size of a container is limited to the maximum size of a disc, since a container must fit in a single disc. With current technology this suggests a maximum container size of 2^{42} bytes. Since the page size is assumed to be 8 KB (i.e. 2^{13} bytes[87]), the page number part requires 29 bits. This allows a container to have a maximum size of 4 TB. Of course most containers will be very much smaller than this.

In view of the above remarks the maximum size of a SPEEDOS address from the viewpoint of virtual memory management is as shown in Figure 23.2.

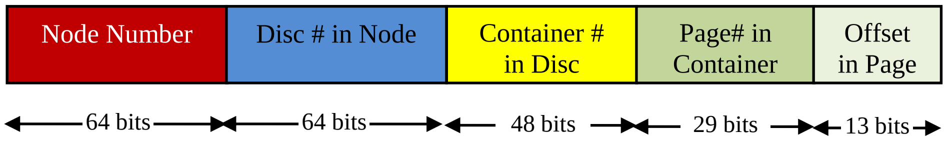


Figure 23.2: A SPEEDOS Address for Virtual Memory Management

[84] The word "disc" is here used generically to describe all devices which can persistently store information, including read-only media.
[85] It is discussed in a later chapter how a disc can be used on several nodes.
[86] Recall that multiple modules/processes can be held in a single container
[87] It would be possible to implement byte addressing via the instruction set, as in the Alpha computers. This would affect some of the calculations in this chapter.

A *virtual page number* has a unique network-wide structure, shown in Figure 23.3, where a *container identifier* consists of <node #, disc #, container #>. In principle the task of an address translation unit (ATU) at a specific SPEEDOS node is to map such virtual page numbers onto main memory page frames.

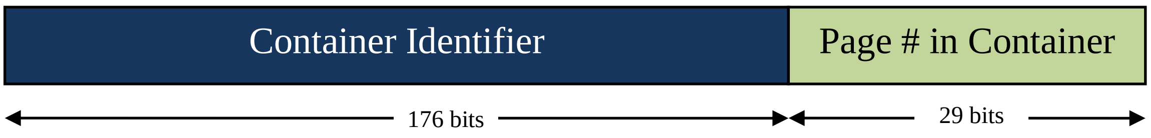


Figure 23.3: A SPEEDOS Virtual Page Number

In the MONADS-PC system, which had 60 bit virtual addresses, the ATU was actually able to handle the equivalent of this. Each virtual address in the MONADS local area network was unique. David Abramson designed and built an ATU (with inverted page table functionality), based on a hash table implemented in hardware, which could translate any virtual page number in the network to a page frame number (or cause a page fault interrupt) [22]. That was in the late 1970s/early 1980s.

Since then the size of main memories has increased enormously, making such an implementation economically infeasible. But not only that; MONADS had only 60 bit virtual addresses (including two bits to indicate on which of the four nodes in the MONADS local area network the container resides), whereas in SPEEDOS we are discussing worldwide unique 218 bit virtual addresses.

These parameters would create two sets of problems for a SPEEDOS implementation based on the MONADS-PC ATU design. First, the increased size of main memories means that the number of entries in an ATU based on the MONADS approach would increase very considerably. Second, because the width of SPEEDOS virtual addresses is vastly greater than that of MONADS virtual addresses, the width of entries in a MONADS-style ATU would also be significantly larger.

The first problem alone makes a MONADS style implementation infeasible, but the second problem creates substantially greater problems. Hence a different approach is needed in order to translate SPEEDOS virtual page numbers into main memory page frame numbers. In the next two subsections we consider these two problems in turn. The aim is to achieve the translation of SPEEDOS virtual addresses in about the same time as the simpler addresses of current systems are translated and if possible to resolve page faults more quickly than in current systems.

1.1 Managing the Number of Main Memory Page Table Entries

In the early 1980s, at the time the RISC idea was becoming popular, the problem of increasing main memory sizes had already begun to emerge. In Chapter 11 it was illustrated how RISC designers began to cope with the problem by designing systems in which the entire address translation hardware consists simply of a translation lookaside buffer (TLB), which did not have enough entries to translate all virtual page numbers in the main memory. Figure 11.7, which for convenience is repeated here as Figure 23.4, indicates the task of the software in this RISC scenario.

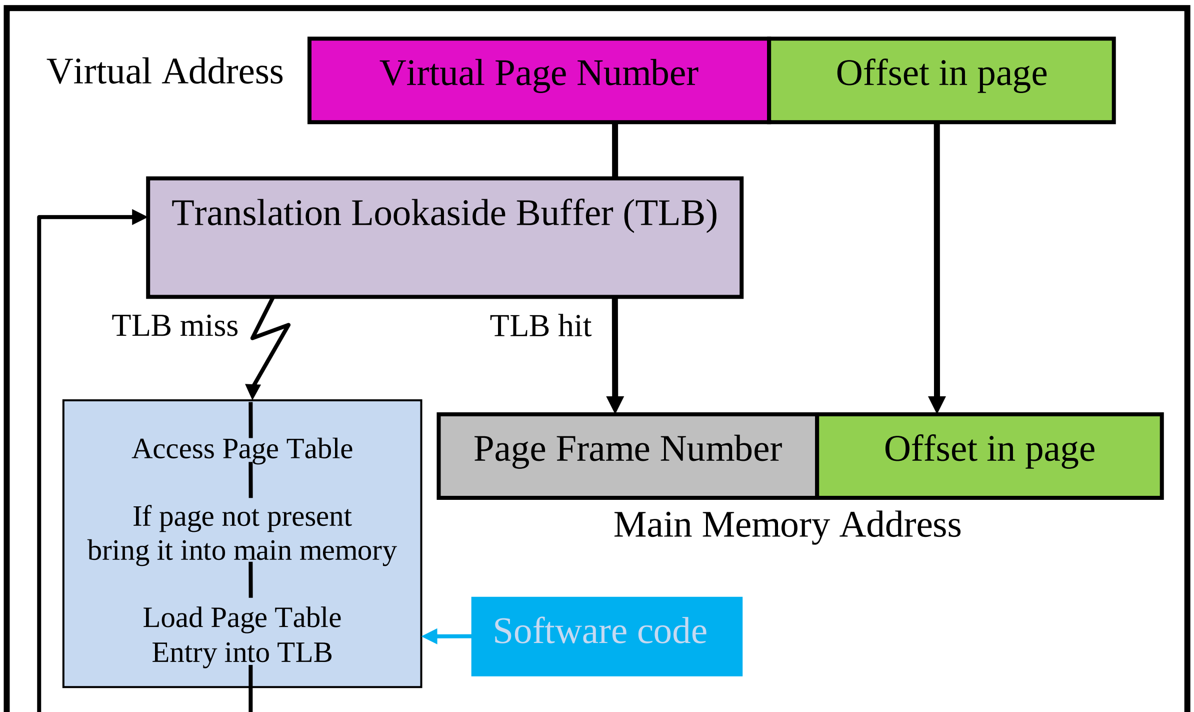


Figure 23.4: The TLB as the entire ATU

Translated into SPEEDOS terms, the core kernel software is responsible for the mechanism aspects of the software code functionality shown in blue in the diagram. Because the TLB is too small to provide a mapping for each page frame in the entire main memory, a complete mapping from page frames to virtual pages (i.e. an inverted page table[88], in SPEEDOS terminology the Main Memory Page Table, MMPT) must be maintained in software. The MMPT is permanently locked into main memory.

When a TLB miss occurs the hardware interrupts into the core kernel code. This first examines the MMPT to establish whether the miss occurred simply because the TLB is not large enough to hold an entry for each page. If that is the case, it updates the TLB using the information in the MMPT and loads the ap-

[88] In this context the use of the name *inverted page table* is not intended to imply a specific implementation, merely the principle that the actual data structure implemented can rapidly translate a virtual page number into a main memory page frame number, without holding information about virtual pages not currently in the main memory. This might for example be a software implemented hash table which has similar functionality to that of the MONADS ATU mentioned above.

propriate information into the TLB, allowing the process to continue execution without being suspended.

On the other hand if the TLB miss arises because a genuine page fault has occurred, the kernel must resolve the fault. This activity cannot take place synchronously, because the effect would be that all other processes would be held inactive until the page fault is resolved. Espenlaub suggested that the Container Manager be responsible for resolving page faults [4, p. 159], but this is an expensive solution, which can be more efficiently handled by kernel processes directly.

1.2 Managing the Width of TLB Entries

The second ATU problem for SPEEDOS systems is the width of entries, which arises primarily because a unique logical SPEEDOS address would require very wide TLB entries. This follows from the decision to support unique worldwide container numbers. Providing an implementation of this in the TLB would be especially costly because for each TLB entry a separate comparator is needed in hardware *for each bit in the virtual page number*. Hence an alternative solution is needed.

In practice TLBs can be implemented in different ways. In some conventional systems an *address space identifier* (ASID) can be associated with virtual page numbers in each TLB entry, thus making addresses belonging to different programs unique (within the TLB), with each currently active thread using a different address space identifier. On other systems the TLB restricts access to a single address space, so that the TLB has to be flushed on each context/thread switch.

We now describe how SPEEDOS might effectively use a TLB which was designed to support only a single address space, i.e. without ASIDs.

1.2.1 TLBs Supporting Only a Single Address Space

If the TLB hardware assumes that only one address space is mapped into the TLB at a time and that on a context switch the TLB is flushed, this raises a special problem for SPEEDOS, because a SPEEDOS container is never active alone. Typically there are at least three active containers: a process container, a code container and a persistent data container. Under some circumstances, there may be more concurrently active containers.

– A module may need access to one or more library code containers (whereby several library code modules can be held in a single container, as was described in chapter 19).

– A need for more data containers can arise if a module provides *n-ary* functionality (e.g. to allow two sets of file data to be merged into a third, or to compare two sets of file data).

It therefore makes sense to support up to, say, eight containers concurrently in a TLB which is flushed on each context switch. To achieve this, the three top bits of a virtual address (as viewed by the TLB) act as a *short container identifier* (SCID)[89]. With three bits used in this way up to eight containers can be concurrently active. Figure 23.5 shows how a virtual address can be used to address eight containers simultaneously in what the TLB views as a single address space.

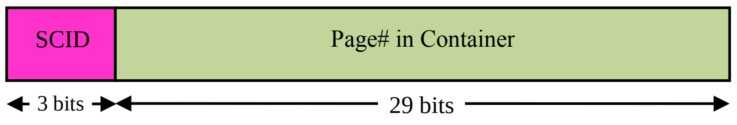


Figure 23.5 The Page Number Presented to the ATU

The actual mapping of the 3 bits might by kernel convention be defined as is shown in Figure 23.6.

000 identifies the process container of the currently active thread.
001 to 011 identify the currently active code containers, i.e. for the main code container and up to two active code library containers.
100 to 111 identify up to four data containers.

Figure 23.6 A Possible Allocation of Short Container Identifiers

It should be noted that

(a) the mapping of containers to actual container numbers in SPEEDOS is a trivial activity which the kernel can organise as part of inter-module calls and returns, and in association with the loading of segment registers;

(b) on an inter-module call and also on a thread switch between two threads of the same process, entries in the TLB for the stack container do not need to be flushed;

(c) it would be a straightforward matter to implement separate TLBs (and main memory caches) in hardware for stack, data and code addressing in an optimised processor design.

1.3 The Main Memory Page Table

Figure 23.7 illustrates the basic structure of the SPEEDOS MMPT. In contrast with the TLB (which in the above proposal contains entries only for the currently executing module), the MMPT contains a complete view of all the pages in

89 It is also assumed that "stealing" three bits of an address is acceptable, but this is unlikely to be a problem in 64 bit computers, even if the within container address is restricted to less than 64 bits.

the main memory, and must therefore hold the *full virtual container number* and of course the page number in container of each page currently held in the main memory.

Virtual Page Number	SCID	Lock Count	Disc Address	Use Bit	Change Bit	RO Bit	EX Bit

Figure 23.7 Structure of the MMPT

The *SCID* is the short container identifier of the page. It is necessary in order that the kernel can work efficiently with the TLB.

The *lock count* indicates whether the page is locked into main memory. It is a count rather than a single bit, indicating how many times (if any) the page is considered to be locked into the main memory, thus allowing different parts of the system to lock in the page independently of each other. Only when it has a zero value can the page be considered for discard from the main memory.

The *disc address* field holds the current disc address of the page, thus allowing a page to be written back to its location in the virtual memory without having to access the relevant page table.

The idea behind a *use bit* is to indicate whether the page corresponding to this entry in the table has recently been used. This bit is set by hardware in the TLB each time the page is accessed. It is used by the kernel's discard algorithm to help determine which page(s) to discard from the main memory when a page frame is needed for a different page. Such an algorithm usually follows the *least recently used* (LRU) strategy [23, pp. 323-333], which is based on the high probability that a page used very recently is likely to be needed again in the very near future. (Like people, computers cannot see into the future, but this algorithm provides a good approximation.)

The discard algorithm can also receive advice from other modules. For example, when a file module becomes inactive (e.g. as a result of all relevant threads closing it) this information can be passed from the appropriate Segment Manager to the MMPT module (possibly via the container's Virtual Page Table module). Similarly when a persistent thread logs out, its pages can be immediately discarded.

A *change bit* is set in the TLB to indicate that the hardware has detected that the page corresponding to the entry has been modified. The advantage of this bit is that if a decision is made to discard a page which has not been changed, then the page need not be written back to disc.

The use and the change bits are set by the hardware in the TLB entry for a particular page. The kernel ensures that they are copied into the MMPT, in order that the kernel can use them when making decisions about discarding pages.

The *read only* bit is not really required (at the page level) in SPEEDOS. However, it is included in the design for compatibility with other systems[90].

The execute bit is not really necessary, since the execute bit in the code segment register allows the page to be executed as code. It is added for the case that the Address Translation Unit can also be used in a more conventional way.

Two kernel processes, the Page Fault Interrupt Process and the Virtual Memory Process are the main users of the MMPT. Their primary functions are to maintain this table and to determine which pages should be held in/removed from the MMPT.

1.4 Mapping SPEEDOS Container Numbers onto SCIDs

The use of SCIDs is sufficient (from the hardware viewpoint) to allow SPEED-OS addresses to be effectively translated by a TLB which was originally designed to translate addresses within a single address space, provided that the TLB is flushed on thread switches and on other context switches (e.g. inter-module calls).

However, that is only part of the story. It is also necessary, from the software viewpoint, to maintain some form of mapping between the SCIDs and the corresponding full 176-bit virtual container identifiers, information which is needed, as we shall see shortly, by the core kernel. To achieve this, the core kernel maintains this mapping in pseudo (or real) processor registers, called *container registers*. As defined above, 8 such container registers are needed, their functionality corresponding to that defined in Figure 23.6.

With this scheme the segment registers (which are the vehicle via which virtual addresses reach the TLB) must be loaded by kernel instructions to contain a 3-bit SCID index followed by a within container address[91]. While the TLB regards the entire structure as an address, the core kernel can use the SCID pre-

[90] In the MONADS systems a *read only* bit was essential to implement remote paging, but in SPEEDOS the use of remote paging was rejected in favour of remote inter-module calls (see chapter 28).

[91] This also has the advantage that segment registers can be much shorter (and therefore much cheaper) than they would otherwise be.

fix as an index into the bank of 8 container registers which it maintains for the currently active thread, thus allowing it rapidly to discover the full virtual address associated with a short address, e.g. when a genuine page fault occurs.

The container registers form part of a thread's current state, and they must be stored/reloaded along with the values in the thread's real registers (e.g. its segment registers) on thread switches (as already described in chapter 20). The kernel sets them up initially when a thread is first activated. Which container registers need to be stored and reloaded on various kinds of call instructions is described in chapter 20.

2 The Local Virtual Memory

The SPEEDOS virtual memory (from the viewpoint of a specific node) consists of the information on its currently mounted discs (which we refer to as its *local virtual memory*) together with the information on other discs at other nodes which it can reach via remote inter-module calls. The latter aspect is considered in chapter 28. In the present chapter the description concentrates exclusively on the local virtual memory of a node, as if it were a stand-alone computer.

The most significant difference between the SPEEDOS virtual memory and the virtual memories of conventional systems is that the SPEEDOS virtual memory is persistent, i.e. there is no separate file system in the conventional sense. The files of a SPEEDOS system are not held as entities outside the virtual memory but are held in containers which comprise the virtual memory. In such an environment the main (RAM) memory can be viewed simply as a cache for the all-encompassing virtual memory[92].

2.1 Virtual Memory Message Blocks

As in the MONADS design the core kernel maintains a heap of virtual memory message blocks, which are of fixed size and therefore can be rapidly allocated and deallocated via a resource set semaphore. They enable those kernel processes that manage the virtual memory to coordinate their activities with respect to the state of the virtual memory. The basic idea is that when the need arises (e.g. when a page fault occurs) a message block is allocated and set up with the initial details (e.g. of the page fault). It can then be passed between the kernel processes, whereby each process updates the message block as a result of its own activity then places the message block's address in the input buffer of the next relevant process. When the reason for creating the message block has been fully handled (e.g. a page fault has been finally resolved) the message block is deallocated and becomes available again in the pool of message blocks.

[92] see Chapter 12.

2.2 The Layout of Information on Disc

A SPEEDOS full virtual address contains both the node number on which a new disc was first initialised and the disc number allocated by the node to the new disc. At this stage we assume that a disc is always mounted on the node which initialised it, but in chapter 27 we consider what happens if the disc is mounted at a different node.

Container number 0 on a disc is a directory of the remaining containers on the disc; this is located at a well-known address on the disc. It is otherwise organised like all other containers from the viewpoint of the virtual memory, with its own page tables (see sections 2.3 and 3.6). It contains the number of the next unused container on the disc, information about the free space on the disc, the length of the directory and the directory itself. The latter provides rapid access to the disc address of page 0 of each container stored on the disc (see Figure 23.8).

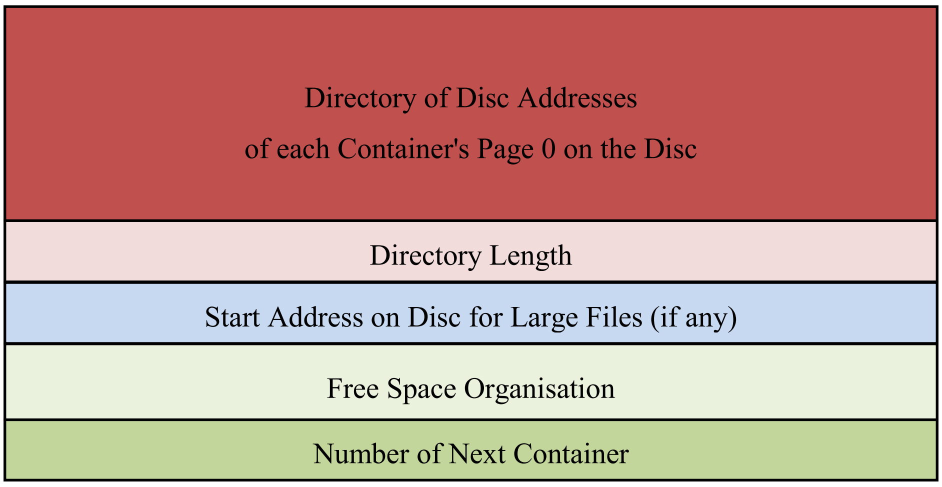

Directory of Disc Addresses
of each Container's Page 0 on the Disc
Directory Length
Start Address on Disc for Large Files (if any)
Free Space Organisation
Number of Next Container

Figure 23.8 Structure of a Disc Directory

When a disc is mounted at a node the kernel reads page 0 of its container 0 into main memory and locks it down until the disc is dismounted (or the system shuts down). Thus when a page fault has to be serviced from a container held on the disc, locating the page table for the directory does not itself cause a page fault (see section 3). But access to the directory entry and to page 0 of the faulting container may cause page faults.

2.3 Organising the Page Tables

A key aim in organising page tables is to minimise the number of additional page faults which can arise in the course of resolving a page fault. In the SPEEDOS environment this in principle involves both finding on disc the container holding the faulting page and finding the page within the container.

The organisation of the page tables in the MONADS-PC system provides a good starting point[93]. SPEEDOS follows the same pattern as far as possible, but must take into account both a number of developments in the SPEEDOS design (especially the use of security sensitive co-modules) and technological advances which have taken place since the late 1970s (e.g. the greater capacity of discs, the use of 64 bit addressing, the prevalence of large *removable* mass storage devices such as USB discs[94] and of the Internet).

2.4 Security Sensitive Co-modules and the Virtual Page Tables

The SPEEDOS design has introduced the idea of security sensitive co-modules to assist the kernel. Some of these modules are located in the containers which they serve, e.g. the Co-Module Manager, the Code Manager, the Segment Manager, the Thread Manager and the Thread Control Manager and, perhaps most significantly in this context, the Virtual Page Table Manager. An important question which this raises is the extent to which this development might conflict with another significant aspect of MONADS, viz. the idea of implementing internet (and other network) activity via the transfer of pages between nodes (i.e. remote paging). This issue is further discussed in chapter 28, where the decision is taken not to use the remote paging technique, but instead to support the idea of remote inter-module calls (RIMCs) via a technique akin to remote procedure calls.

3 Structuring the Page Tables of a Container

Each container has a Virtual Page Table (VPT) Manager. In contrast with Espenlaub's approach, the VPT Manager is not invoked via an inter-module call by the kernel to translate virtual memory addresses into disc addresses. However it is responsible for organising the page table in its container and for preserving this during system shutdowns.

Assuming that a page (corresponding to a block or contiguous group of blocks on disc) is 8 KB (2^{13} bytes) in length and that a page table addresses a disc block in 32 bits, this allows for a maximum disc size of 2^{13} x 2^{32} bytes = 2^{45} bytes = 32 TB[95]. This is more than enough to support the proposed maximum container size of 2^{42} bytes (see section 1). In principle we might envisage an indexable page table with a separate entry for each 8 KB block on a disc, in the worst case that a single container consumes a whole large disc. However, such a page table (with 2 entries per 64 bit word) *for a single container* (restricted to 4 TB in length) would require 2 K addresses per page, and for a complete 4 TB

93 For a detailed description see [21].

94 Solid state devices (SSDs) are organised as discs and are here simply treated as USB discs.

95 This assumes byte addressing. If word addressing is used, 256 TB could be addressed.

disc it would require about 2 billion (short scale[96]) pages, an obviously ridiculous approach.

3.1 Small Files

Most of the files (i.e. co-modules) in most systems are relatively small (e.g. text files, code files, emails, presentations, spreadsheets, letters). In fact, every container is likely to hold a number of small files, e.g. the kernel co-modules associated with a container, such as the Co-Module Manager, the Segment Manager, etc. Consequently a *Small Page Manager* (within the Virtual Page Table Manager) must be automatically installed in every container as part of the initialisation of that container.

This might, for example, provide a total space of up to 4 MB (2^{22} bytes) on disc for all the small files (user files and kernel co-modules) in a container. With a page size of 8 KB (2^{13} bytes), and assuming on-demand dynamic allocation of individual pages on disc, the maximum number of entries required for the Small Page Manager's page table (i.e. a mapping from virtual page number to disc block number) is 2^9. Assuming once again that the blocks of a disk can be addressed in 32 bits (2^2 bytes), the maximum page table length for the small files would be 2^9 x 2^2 bytes = 2^{11} bytes, i.e. one quarter of a page for each container. This could be stored in the first page (page 0) of the container, leaving ¾ of page 0 free for other information. This is a more realistic approach.

3.2 Large Files

Large files need not have massive page tables. Because SPEEDOS decouples the mapping of virtual page numbers to main memory page frames from the mapping of virtual page numbers to disc addresses, the ATU hardware need not be aware of the structure of such page tables (for small or large files). This creates the opportunity to devise alternative schemes, especially for large files.

For example, if all the pages of a large file are placed contiguously on disc, then a page table can consist simply of a start address on disc and a length. If space on a large disc is (partially) allocated in units of say 128 MB (or larger) then a very large file could consist of several "multi-pages" of 128 MB, where the addresses within a 128 MB unit, each normal page address within a multi-page unit only needs a start address and length to find the appropriate page. Using such methods the location on disc of any of the contiguous pages can be calculated rapidly. Such strategies might be used by a *Large Page Manager* (hidden within the Virtual Page Table Manager) for video files and other contiguously stored files, and pre-paging could in some cases be useful. Another possi-

[96] see https://en.wikipedia.org/wiki/Billion.

bility is that large files which start life as small files could be organised initially as small files which after a certain limit is reached could be extended to become (or converted into) large files. Alternatively they could use small files to index into large files. Conventional database techniques and/or "big data" techniques might be used in the Virtual Page Table Manager's Large Page Manager. Different containers can use different techniques.

3.3 Page Tables for a Process Container

Like other containers a process container holds security sensitive co-modules, including in this case a Thread Manager and a Thread Control Manager. Their needs can be handled using the same small file organisation proposed for the other security sensitive co-modules.

Beyond that, the kernel needs a thread stack for each user thread in the container. This almost exclusively contains inter-module linkage and parameters[97], which we refer to jointly as a stack frame. A rough calculation suggests that a page frame of 8 KB will typically hold at least 8 such stack frames, and most threads will probably never require more than 8 nested stack frames, so that, being *very* generous in most situations (the content of) an entire thread stack will easily fit into say four pages, and in most cases into one or two pages.

This situation is not comparable with stacks in other in-process systems such as Multics, the Burroughs 6700 or the ICL2900, because those systems included not only linkage data and parameters but all the process-oriented data of the user programs themselves, and therefore were more prone to the risk of stack overflow.

Even allowing for say 256 threads in a process container (the maximum possible using an 8-bit index number), each using four pages, the total space for all the thread stacks in a process would be 2^8 threads x 2^2 pages x 2^{13} bytes = 2^{23} bytes, double the proposed size of 2^{22} bytes for a small data file. This suggests that a different page table organisation for process containers might be used, in which (a) a small file organisation could be used for the co-modules and (b) the second page (page 1) of the process container could be used for thread stacks. If the entire second page were used as a page table this would allow for 256 threads each with a stack frame of up to 8 pages. The actual allocation of space could be left to a further standard algorithm in the kernel.[98]

97 see chapter 20.

98 This makes sense because the kernel manages space directly on the thread stacks, whereas the need for new pages in other circumstances is organised by the Segment Manager (see section 5 below).

3.4 Which Page Tables are Needed in a Particular Container?

From the above discussion it is clear that different page table constellations are needed (and can be organised differently) in different containers. A simple way to do this is to let a user creating a new container indicate that the container will be used by him for small files and/or for large files (and if the latter approximately how large his file will be) or for a process.

Notice that the form which this choice takes need not be expressed at the user level simply in the form "large file" or "small file", but rather in terms which can also provide other useful information (e.g. a video file, a code file, etc.). In some cases it will not be necessary to demand information at all directly from the user, since user level software often creates such files (e.g. an email program usually creates email files, some video programs create video files, etc.). In such cases the program will usually have much more information about which Virtual Page Table Manager variant is needed. Similarly, in a copy operation it should be possible for the software to discover from the Virtual Page Table Manager of the file to be copied what is the most suitable constellation.

But from the system's viewpoint it will usually be sufficient to provide standard configurations. A page management table for serving the kernel's comodules will *always* be necessary, but a decision will usually be necessary regarding a large file (or files) and a small user file (or files) or a process. A further parameter which will be useful (in all cases) is approximately how large the required page table should be, and in the case of large files whether the table should be organised as a number of units of fixed (relatively large) size. It is also conceivable that special managers imitate database techniques (e.g. using "record keys") or Virtual Page Table Managers which pre-page (e.g. when it is known that the information will be accessed sequentially, as for video files and some commercial files).

3.5 Organising the Page Tables

From the organisational viewpoint a small file page table can be organised at a fixed position in page 0 of each container, indicating the virtual address where the individual page tables can be found, as is illustrated in Figure 23.9. This may differ from the organisation in a process container, since further information about the individual stacks may be necessary.

The address of the actual page table for the small (8 KB-paged) files is fixed in page 0 and is known (see section 2.2.1). The page boundaries in the blue boxes allow the kernel to determine which of the two page tables applies when a page fault occurs. In each case the address points to information describing how the actual page table is organised, its start address, the length of the page table, etc. This information allows the page fault resolution software quickly to establish which of the page tables is relevant when a page fault occurs. A null pointer indicates that the relevant page table does not exist. The orange fields allow the page management software to discover the last valid entry in each page table. The green fields are the complete page tables for a large file. This structure allows all the page table information for a container to be held in a single page, i.e. page 0!

Page boundary for small user co-modules	Page count for small user co-modules
Page boundary for the large user co-modules	Page count for the large user co-modules
Disc address of first 128 MB unit	Disc address of 2nd 128 MB unit
Disc address of 3rd 128 MB unit	Disc address of 4th 128 MB unit, etc.

Figure 23.9 An Index of Page Tables for a Container

3.6 Page Table Code

The above discussion suggests that a number of virtual page table co-module configurations should be available in SPEEDOS. To ensure the integrity and privacy of other users' information on a disc, it would be unsafe to allow normal users to write such modules. Consequently they must be written and tested as part of the SPEEDOS software, with users being offered a choice of module.

3.7 The Disc Directory

So far it has been assumed that the start addresses of the containers on the disc are known, and Figure 23.8 assumes that a directory exists to achieve this purpose.

The main issue to be faced for a SPEEDOS disc directory is that a very large range of numbers (in principle 2^{48}) is used to address the containers on a disc. Such a large number was chosen to ensure that container numbers remain unique over the life of the disc (and so never have to be re-used[99]). However, nobody is suggesting that a disc will ever, over its lifetime contain (or have created) such a large number of containers. In fact there are very, very considerably less bits on even a large disc than there are potential container numbers! Hence such large numbers cannot simply be used as indices into an array which has an

[99] see volume 1 chapter 2.

entry for each possible container number. Consequently an efficient technique has to be used in order to locate the first page of a container on the disc. Let us first get a feel for the nature of the problem.

The initial impulse of some computer scientists is probably to use a hash table[100] to achieve the necessary mapping from container number to disc address. But in the present context this technique has a disadvantage, i.e. the entire hash table must be created in advance, before it can be used. However, this is an overkill solution for many discs (especially large discs which contain large files that are rarely deleted).

In practice it would be more flexible to use a data structure which does not require us to specify in advance the number of containers to be placed on a disc but which can dynamically grow to suit the actual need, and which can ideally locate containers in a single probe. That may sound like a tall order, but in fact it turns out to be quite simple to implement, at least for discs which contain files that are rarely deleted. The reason for this is that the hash keys (i.e. the container numbers) are not allocated randomly (as they typically are in problems to be solved using hashing), but start at zero and increase by incrementing the previous highest value by one. In our case the first container to be placed on the disc is container 0, the second is container 1, the third is container 2, etc. This means that at least in principle an extensible array would be an appropriate data structure. Let us now consider this in practice.

a) At the beginning of a disc's life it is possible to use only a single page for the directory. If the directory entries are 32 bits wide, up to 2^{11} (2048) entries fit into an 8 KB page of the directory.

b) When the next container number allocated reaches 2048, the table can be extended by a page so that the directory can now hold 2^{11} additional entries, etc. We see that at any point in time the array need only be large enough (in pages) to accommodate the largest current container number.

c) If a container is deleted, this involves zeroing the relevant entry (without changing the length of the array).

d) So long as container numbers are relatively small, each page of the directory can be locked down in main memory.

But now the question arises whether the table can, if necessary, be extended sufficiently to cope with the maximum number of containers likely to be placed on the disc. To answer this question we need first to consider how many containers can fit on a disc, and then to allow a reasonable number of deletions and re-use their space (but not their container numbers).

[100] https://en.wikipedia.org/wiki/Hash_table#Hashing

The minimum size of a container (including its data) for a very small file (e.g. a simple short text file) is one page. The maximum number of pages, i.e. small files, which would fit into say a 4 TB (2^{42} byte) disc is $2^{42}/2^{13} = 2^{29}$ pages/small files, and these can be addressed (using 32 bits per address = 2^{11} addresses per page) in 2^{18} directory pages. Suppose now that each file is replaced $2^4 = 16$ times over the life of the disc, the length of the directory would grow to 2^{22} pages = 2^{35} bytes = 32 GB, which is (arguably) relatively insignificant in a 4 TB disc.

The numbers which we have used are more or less worst case (and apply to discs which at the time of writing are larger than most discs currently in use): in reality many files on a large disc will be large files, thus reducing the number of containers used very considerably. Likewise the likelihood that each file will be replaced 2^4 times over the life of a disc is improbable, so we can assume that the extreme case described above is very unlikely to occur. In fact the situation can arise where a large disc is never even once completely filled. This is likely to happen ever more frequently as the size of disc capacities increases and the cost per byte decreases.

On the other hand there will also be cases where over time discs are filled and their space re-used multiple times. As the extendable array suggested earlier grows and its entries become ever sparser, some people will be in favour of a solution which is less wasteful of space, the more so if a good chunk of the main memory is locked down to ensure faster directory lookups.

I therefore suggest a compromise solution for such cases. When a new disc is initialised the extendable array solution is initially used. But if its size reaches a certain limit[101], the system warns the system manager, who at a convenient point requests the system to switch to a hash table solution. (Whereas the size of the extendable array solution is proportional to the number of containers used, the size of a hash table is proportional to the size of the disc.)

Such a solution might be as follows. The length is determined for a hash table; this has entries that include a valid bit, a container number (as key) and the address of the first page on disc of the relevant container, together with an overflow mechanism to deal with collisions.[102] Whether the system continues to work with the indexable array for the container numbers which it already holds, or adds them to the hash table, is a system design decision. Of course the current structure of the directory is held in the first page of the directory. Not only the

[101] This limit will depend on the size of the disc in question, on the availability of main memory to lock down directory pages). The limit can be immediately recognised when a defined new container number has been reached.

[102] see e.g. https://en.wikipedia.org/wiki/Hash_table

first page of the directory, but also as much of the directory as is reasonable, should be locked into main memory.

4 Resolving TLB Faults and Page Faults

This section describes how the SPEEDOS system can handle page faults. This is one of the most significant activities of the kernel, and it serves as an example of how the virtual memory approach works. But it also belies the belief of many computer scientists that protection and privacy can be "bought" only at the cost of efficiency. Whereas some conventional systems are slowed down quite considerably by the fact that they can only resolve a page fault by creating a number of further page faults (to access page tables), in SPEEDOS page faults can in many cases be resolved without creating additional page faults to access page tables (or often in one probe if the relevant directory entry is not in locked down main memory).

Several kernel processes are potentially involved in the handling of page faults. The most important of these is the kernel's Virtual Memory Process (VM Process) which is responsible for controlling the use of the main memory and organising the transfer of virtual memory pages between the discs and main memory. It is activated by all other kernel processes which indirectly need access to these services. The kernel processes communicate with each other via virtual memory message blocks, as was briefly described in section 2.1 above.[103]

We begin by describing how a TLB fault is handled. This may or may not lead to a genuine page fault. We then turn to the handling of the page faults which initially arise during the handling of an inter-module call. Finally we consider normal page faults.

4.1 Handling a TLB Fault

When a TLB miss occurs, this causes an interrupt which the kernel's interrupt analysis thread passes to the Page Fault Interrupt Process, handing to it the SCID and address (page and offset) within container of the faulting page (Figure 23.10).

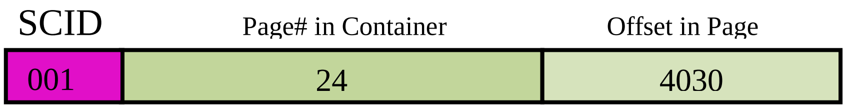


Figure 23.10: The TLB Fault Information

The Page Fault Interrupt Process then establishes the full virtual page number of the faulting page (by establishing which thread is currently executing in order to

[103] Much of what follows is based loosely on the MONADS PC design as outlined in chapter 8 of Frans Henskens' thesis [20, pp. 159-181].

disambiguate the SCID). For efficiency this is placed in a global variable when the kernel executes a `switch_thread` instruction. It then scans the Main Memory Page Table to check whether the missing page is in the main memory. If so it replaces some other TLB entry with that formed from the information in the MMPT and exits, causing a reschedule of the kernel processes. Since the User System Process, in which user threads execute, has not been disturbed by the TLB fault, the next user level instruction executed (which was executing the user thread that caused the TLB fault), is simply repeated[104] as if nothing had happened.

4.2 Handling Page Faults

We concentrate on the resolution of *local* page faults in order to keep the initial description relatively straightforward, leaving a description of the issues associated with the mounting of discs on "foreign" nodes[105] to later. Thus it is assumed in this section that the missing page is on a locally mounted volume.

An important aim in the design of page fault handling is to minimise the number of page faults which arise (as a result of page table accesses) when resolving a page fault. The issue arises in SPEEDOS because more than one page table must be consulted. In the following scheme the number of page faults can often be reduced to one (the minimum possible), i.e. the page fault actually to be resolved can be handled without a further page fault occurring (or with a single page fault when accessing the disc directory). This is subject to the disc directory entry being available in locked down memory.

Since there are only a small number of page table organisations, the code for all of them is part of the kernel code[106].

4.3 Locking Down Page 0 of a Disc Directory

This is relevant to the handling of page faults because page 0 of the first block on a SPEEDOS disc holds the start of its disc directory, see Figure 23.8 and section 3.7. It is therefore essential that this page (for each mounted disc, including fixed discs) is read into the main memory when a disc is mounted (or in the case of a fixed disc at boot time) and is then locked down until the disc is dismounted. (What happens when a foreign disc is mounted is explained in more detail in chapter 27.)

104 Recall that the program counter is only adjusted after an instruction completes.

105 We refer to a disc mounted on a node which is not the creator node as a "foreign" disc.

106 This differs very substantially from Espenlaub's solution, which involved a forced method call to the Container Manager and a further call to the Virtual Page Table Manager (see [4, p. 159]). He argued for flexibility, but here efficiency is more important.

The VM Process maintains a list of mounted discs[107] and establishes which kernel disc process is responsible for the disc (if necessary allocating an extra kernel process for it) and ensures that the disc can be activated (e.g. by organising for it an appropriate disc driver and checking whether the disc is authorised to be mounted on the current node). The VM Process then allocates a message block, records in it a free page frame number and passes the message block to the appropriate disc process to read its page 0. On a successful read the disc process records this in the message block and returns it to the VM Process. This then marks the page as locked down in the Main Memory Page Table and deallocates the message block. Thereafter page 0 of the disc directory can be accessed as part of the resolution of page faults for that disc without causing a page fault.

4.4 Page Faults Arising on an Inter-Module (or Similar) Call

Page 0 of a file (or program) container is normally first used (leaving aside the initial setting up of the containers) when the kernel receives an inter-module call or similar call to a module in the container. One of the operands of the call instruction is a module capability. From this the kernel (executing in the User Request Process) forms the address of page 0 of the module in order to access the "pointer to state data" of the module's root segment, which is held in the container's page 0 (see Figure 19.6). This allows the kernel to load the address of the root segment of the module into segment register 5[108]. From the paging viewpoint this is fortunate, because page 0 of this container also contains its page tables, which will be needed to resolve page faults for *all further references* to the state data in that container.

The User Request Process (i.e. the process handling the inter-module call) claims a new virtual memory message block of the type "request and lock page 0" (containing the virtual memory address of the required page 0) which it passes to the VM Process. As a result of this, a reschedule allows the (higher priority) VM Process to execute before the User Request Process can continue.

When the VM Process receives this message block it checks the MMPT to see whether the page is already in the main memory. If so it locks the page (not necessarily the first lock) and returns the message block to the User Request Process indicating that page 0 is locked down and waits for its next task. If the page is not in the main memory it sends a message block to the User Interrupt Process indicating to the User Thread Scheduler that the current user thread should be suspended and hence should schedule another user level thread. The

[107] This is known as the Local Mount Table (see chapter 27 section 2.2 and Figure 27.4).
[108] See chapter 20 section 8.1

VM Process next updates the message block and passes it to the disc manager with a request to read the page. The latter eventually reads the page and passes the message block back to the VM Process, which locks the page and returns the message block to the User Request Process with an indication of success.

The User Request Process can now access the information required to set up the root node of the file module to be called, but still requires the information needed to set up the code segment and the code start offset for the IMC[109]. It therefore repeats the same procedure by sending a modified "request and lock page 0" using the address of the container holding the code (which it obtains by examining the co-module table in page 0 of the data container). When it has the assurance that page 0 of both containers are locked into main memory it sets up the appropriate addresses in the user thread's register save area (including the values for parameter segment registers 0 and 1 as well as the code segment register and the state data register 5). It then creates a message block for the User Interrupt Process indicating that the user thread can now continue. When the UTS schedules the thread it will as usual invoke the kernel `switch_thread` instruction causing the kernel to change the global variable indicating which thread is active, load the registers for the IMC and proceed in the new module.

Without taking further precautions it would be possible for the above mechanism to result in the page 0 of the data container and/or of the code container to be locked multiple times, because if the thread has to be delayed to service a page fault, the call instruction is repeated. To avoid this the User Request Process places the linkage segment and an IMC stack record (see chapter 20 section 8.1) on the thread stack at the first attempt to execute the call, and always checks whether this already exists (i.e. whether this is a repeat attempt) *before* carrying out an inter-module call. It records in the IMC stack record whether the page 0 for the file module and page 0 for the code module have already been locked, to avoid double locking them.

The page unlock operation must be applied (twice) on inter-module returns. This simply involves the User Request Process, when processing an inter-module return, in sending a virtual message block to the VM Process in which it provides the virtual addresses of the two page 0s to be unlocked. If the lock counts reduce to zero then the pages can, but need not, be discarded.

Finally it is obvious that other kinds of calls (library calls, co-module calls) can be treated in a similar way, except that for a library call no persistent data

[109] In view of the decision (see chapter 28) not to support remote paging, the file module and its code module must be stored at the same node. It might be possible to allow remote paging on code modules, since their pages are never modified, but I have not considered this idea in detail.

container is required, since it shares this with its host module.

4.5 Locking Page 0 for the Process Container

As with other containers the normal page tables of a process container are held in its page 0. However the page table entries for an individual thread stack are held in page 1 (see section 3.3 above).

The appropriate time to access and lock down a process container's pages 0 and 1 is when the User Thread Scheduler instigates a context change by executing the kernel `thread_switch` instruction (see chapter 22), which has two parameters: a kernel capability demonstrating the right to call the instruction and the unique thread identifier of the thread to be activated. The latter contains the unique identifier of the process container. By appending to this the page numbers 0 and 1 it forms the virtual addresses of the pages containing the appropriate page table entries (including those for the thread stack). Using this information it can claim a new virtual memory message block of the type "request and lock page 0" (containing the virtual memory address of the required pages 0 and 1) and pass this to the VM Process. This causes a kernel reschedule such that the VM Process will execute before the User Request Process can continue. Thereafter if follows a similar procedure to that described in the previous section, including requesting that the process container pages 0 and 1 of the previously active user thread be unlocked.

Note: If the user thread scheduler has no threads which can be scheduled, it executes the kernel `thread_switch` instruction in which the "unique thread identifier" of the thread to be activated has all zero fields. In this case the kernel takes appropriate action to allow the kernel's own lowest priority process (the "idle" process) to be executed when it has no work to do.

4.6 The Page Fault Interrupt Process

If a TLB miss turns out to be a normal page fault, the Page Fault Interrupt Process must first establish the full container identifier corresponding to the SCID of the missing page, which is held in a global variable of the kernel (section 4.1).

Next, as in the MONADS system, the Page Fault Interrupt Process checks the currently valid virtual memory message blocks to establish whether there is already an outstanding request for the missing page. If so the new page fault is linked into the existing message block and then causes a kernel reschedule.

If this is the first request for the page it must then be checked whether the missing page is an unmodified page still in the main memory after it has been released to the disc's free list, or whether it is a modified page waiting to be written to disc before being released to the free list. In both cases the page can be removed from the appropriate list and can be treated as if it has just been read from disc.

If neither case holds, a new message block is obtained from the pool; details of the page fault are entered into this block, i.e. request type = <new page fault>, container#, page#, user thread#. The process then sends a message to the User Interrupt Process to suspend the current user thread and finally places the message block into the queue of the Virtual Memory Process.

4.7 The Virtual Memory Process and the Disc Processes

The Virtual Memory Process (VM Process) controls the use of the main memory and organises the transfer of virtual memory pages between the discs and main memory. Using the page table structures defined above, the handling of a page fault can logically[110] be summarised as six basic stages:

1. Request the UTS (via the User Interrupt Process) to suspend the faulting thread.
2. Ensure that enough page frames are free to resolve the page fault (including accesses to page tables). If not, free sufficient pages by calling the discard algorithm. Often only one page frame will normally be required, i.e. for the page which has faulted.
3. Access the appropriate Disc Directory to discover the disc address of page zero of the faulting container. (This stage is only necessary if stage 4 fails! Depending on the structure of the disc directory this may require multiple probes.) Since the faulting page address can be used to formulate the virtual address of page 0 of its container, stage 4 can in practice precede (and perhaps eliminate the need for) stage 3.
4. Access the page tables in page 0 of the faulting container to locate the missing page's disc address.
5. Read the missing page into main memory and adjust the MMPT.
6. Indicate to the User Thread Scheduler that the faulting user level thread can now continue.

We consider these steps in turn.

4.7.1 Request the User Thread Scheduler to Suspend the Faulting Thread

This need not be the first step, but it must be carried out before the disc process is requested to read in the new page. It is achieved by passing a request to the kernel's User Interrupt Process to suspend the current user level thread. This allows other user level threads to be activated and so use the CPU time while the page is being read into the main memory.

[110] In an algorithm, stage 4 is carried out before stage 3, and may make stage 3 unnecessary.

4.7.2 Availability of a Page Frame for the Faulting Page

Assuming that at most one additional page fault (for the directory) for reading page tables will normally arise in order to resolve a page fault, then at most two free page frames are normally required. (This depends on the directory structure and how much of it is locked down.) The VM Process can check if there are free page frames by scanning the (locked down) MMPT (or by some optimisation of this if the MMPT includes a linked list of free page frames).

If none is available the page discard algorithm must be activated. This is executed in the VM Process. Discard algorithms have been discussed extensively in the literature and need no further discussion here, except to point out that SPEEDOS can use information such as the fact that a persistent thread has been logged out as indications that related pages can possibly be discarded.

When sufficient page frames (normally one or two – see above) have been freed by the discard algorithm they are temporarily reserved for use in resolving the page fault.

4.7.3 Accessing the Appropriate Disc Directory

When the system starts up, the associated fixed discs are initialised such that the disc directory of each disc (page 0 of its container 0) is read into the main memory and locked down.

When a removable disc (e.g. a USB hard drive) is mounted at a node the kernel's interrupt analysis routine receives an interrupt which it passes to its disc process. This then reads its disc directory and, assuming that it is a SPEEDOS disc (which we temporarily assume was created at the current node) it similarly reads the first page of its disc directory (page 0 of its container 0) into the main memory and locks it down. Henceforth page faults for pages on that disc can access the directory without causing a page fault on its initial page, though depending on the structure of the disc directory and the extent to which its pages can be locked down, further page faults might occur. (In this respect it will be sensible, as in MONADS, for the MMPT software to provide a "peek" operation, which indicates whether a page is in the main memory without causing a page fault.[111])

4.7.4 Accessing the First Page of the Faulting Container

In sections 4.3 and 4.4 we have seen how this stage can normally be carried out without causing a page fault, since a previous inter-module call or user level thread switch will have resulted in the first page of data, code and stack containers will already be locked down when a page fault occurs for some random page

[111] see [22] chapter 7 section 7.4.2.9.

within a container.

4.7.5 Reading the Missing Page into the Main Memory

When a kernel process (e.g. the VM Process) establishes the need for a page to be read from disc into the main memory (or written from main memory back to disc), it modifies the current virtual memory message block to indicate the disc address of the virtual memory page to be read/written and the main memory page frame into which/from which a page it is to be written/read, signifying that the operation is a read or write. It then places the message block onto the queue of the appropriate disc process and issues an `AsetV` operation to activate the process (see chapter 22).

Each disc process operates in two parts. The first part takes an entry from its input queue, starts a disc transfer then waits for the transfer to complete. The second part follows its activation on completion of the transfer, checks the result of the read[112] (or write), removes the message block from its input queue and places it on the input queue of the requesting process, indicating that the transfer was successfully completed.

Following the MONADS technique, a disc process has an algorithm along the following lines:

```
repeat {
     modified_AsetP // i.e. scan message blocks and get best
          // entry from queue, see below;
     start transfer on associated hardware device;
     AsusT; {wait for interrupt from h/w device}
     check result;
     finish job;
     update and remove the corresponding item from the queue;
     notify the requesting process of completion (using AsetV)
     }
forever;
```

It can easily occur that more than one message block accumulates waiting for the process to continue, i.e. theoretically for its `AsetP` operation. However, if the process simply begins with a standard `AsetP` operation and uses the information waiting for it in the corresponding virtual memory message block, the result will be that the process will sequentially access the requests in the order that they occurred. This may be acceptable for SSDs and for memory sticks, but not for conventional rotating discs. As is well known, the result of such a course of action would be inefficient head movements on traditional discs, and it would be much more satisfactory to use an efficient algorithm (such as the elevator algorithm in its C-Scan form[113]) for this purpose. But to use such an algorithm im-

112 Here we assume that there are no errors.

113 see https://en.wikipedia.org/wiki/Elevator_algorithm.

plies that a disc process has an overview of all the currently outstanding requests for the disc under its control.

This can best be achieved by providing a special modified version of `AsetP` (which remains invisible to the clients of the process) in which the process scans the entire list of waiting input message blocks to select the best request (i.e. that which is most efficient according to the algorithm chosen). The message block numbers for all the currently outstanding requests can be discovered by examining the resource set associated with the disc process's semaphore, in which the bits correspond to the positions of the relevant virtual memory message blocks holding the details of the required transfers. When the choice has been made, the semaphore's integer part and set part must both be modified to reflect that the chosen message block has been used.

The actual transfer of information can then proceed. When the transfer has been initiated the process can suspend by using the `AsusT`[114] instruction[115]. In due course, when the transfers have completed, the appropriate kernel process must then remove the message block from its queue and pass it to the next stage, using an appropriate `AsetV` instruction. This is probably the User Interrupt Process, advising the UTS that the page fault has now been resolved.

Each disc mounted at a node has its own individual kernel disc process which is responsible for managing the disc and for accessing (reading and writing) individual pages between its disc and the main memory.

Processes for fixed discs can be statically allocated at system start-up. When a removable disc is mounted a free process must be allocated and noted in a table of discs. Its disc directory is then locked into the main memory.

When the kernel's interrupt analysis routine receives an interrupt for I/O completions on a disc, it first consults the table of discs to find out which kernel process is responsible and then executes `AactT` to de-suspend the appropriate process.

Finally, the above explanation of page fault resolution illustrates quite clearly that Espenlaub's suggestion of issuing a forced method call from the kernel to an interface routine of the Virtual Page Table Manager is not necessary and would be much less efficient (though more flexible) than the above proposal. Also, his concerns about stack overflow can be forgotten.

5 Allocating Space on Discs and Segment Management

Having established how the discs are laid out and how page faults are handled,

[114] See chapter 22 section 7.1.

[115] It may be necessary to use `AsusT` more than once, e.g. if there is a separate seek operation (see [5, p. 168]), and in the case of other more complicated devices.

we now consider how space is actually allocated to containers and their pages.

5.1 Creating Segments

We first consider the case of small files, where the page tables allow the possibility that pages can be individually allocated on a page/disc block by page/disc block basis.

The first issue to note is that although the Segment Manager co-module is primarily concerned with segments in its container it can easily determine where page boundaries are, assuming that the page allocation for users starts at a new page boundary which is known to it, since (assuming 8 KB page sizes) each new user page begins at a virtual address in which the final 13 bits are zero. Hence given the start addresses and lengths of segments it can, for example easily determine whether they are in a single page, whether they span a number of pages, etc.

Thus the Segment Manager has an overview of the mapping between segments and pages, and is therefore in a position to determine (via requests to allocate segments) when new pages should be allocated for the container which it serves. Hence it can use a protected kernel instruction (`request_pages`), which allocates a virtual memory message block and causes the Virtual Memory Manager to be scheduled. This in turn requests the appropriate Disc Directory Manager to allocate space on the appropriate disc and return the disc address(es) of the page(s). All the additional pages required for the segment (if any) can be allocated together. While new pages are being allocated, the kernel requests the User Thread Scheduler to suspend the current thread until the page has been allocated. When this has been completed the Segment Manager continues.

The disc manager maintains a free list of disc blocks and from this allocates the required number of small pages, removes these from the free lists, advising the segment process of their addresses. The segment process adds these addresses to the appropriate page table.

For large files the procedure is similar, except that it will be quite unusual (but not impossible) to have a segment that spreads over two or more large allocation units.

5.2 Deleting Segments

Deletion of segments follows a similar pattern except that this is more or less the reverse procedure. The Segment Manager can use a `delete_pages` kernel instruction to initiate the deletion.

5.3 Segment Manager Requirements

The actual organisation used by a Segment Manager can vary from container to container, but this must conform to the requirements of the kernel, as follows:

a) It must organise segments into separate groups corresponding to the separate page tables and should not allow a segment to cross the page boundaries defining these groups. (It can use information obtained by calling the Virtual Page Table Manager to learn where the boundaries lie.)

b) When requested to create or delete a segment it also checks whether a new page is required or a page can be deleted, and calls the Virtual Page Table Manager advising it of this.

c) The Segment Manager is also responsible for garbage collection, and can of course use different techniques in different containers.

6 Creating a Container

The Container Manager provides a semantic routine **`newContainer`**, which is responsible for creating new containers. This routine has two main tasks. In the first phase, after checking the validity of its input parameters and possibly providing a default disc capability, it activates the kernel instruction `new_container`. In the second phase it prepares the container for use.

6.1 The Kernel's `new_container` Instruction

This instruction has three operands. The first of these is a capability for the disc on which the container is to be created; the second is a boolean value defining whether the container is planned to contain a large file and the third is a boolean value indicating whether the container is for a new user (or simply a further container for the user identified in the red tape of the process container containing the thread stack which executed the `new_container` instruction).

After carrying out appropriate checks (e.g. that the disc is on-line and is not full) the kernel then prepares a page-sized buffer in its main memory to act as a page 0 for the new container, in particular preparing a small file page table and if appropriate a large file page table.

It then begins to fill out the identification fields of the new container, as defined in Figure 19.2 and is here repeated as Figure 23.11 for convenience.

The container numbers in the identification fields are directly or indirectly available to the kernel in the red tape of the requesting thread's container or in its stored registers (as augmented by extending the SCIDs into full addresses). The only exception is the container number of the new container, which is obtained in the next step. Depending whether the third operand of the instruction is set to 'new' or 'existing' user the current owner field is set to either the new container number or the owner of the creating thread.

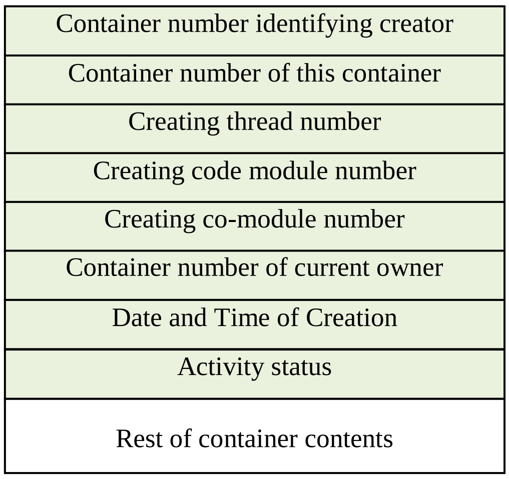

Container number identifying creator
Container number of this container
Creating thread number
Creating code module number
Creating co-module number
Container number of current owner
Date and Time of Creation
Activity status
Rest of container contents

Figure 23.11: Identification Fields of a Container

The creating thread number is the full thread number of the currently active thread. The date and time of creation are obtained from the time and date in the system clock. Finally the activity status field is set to "new container".

After the identification fields have been prepared, the kernel then creates the new container, which requires the following steps.

i) Obtain and update the next container number. This is held in page 0 of container 0 of the chosen disc (see Figure 23.8).

ii) Record the container number in the appropriate identification field in the buffer being prepared to become page 0 of the new container.

iii) Obtain the disc address of the next available page (8 KB block) on the requested disc.

iv) Add an entry to the disc directory for this container number, mapping container number within disc to the disc address.

v) Add the disc address into the (first entry of) the small page table for the new container.

vi) Prepare a virtual memory message block (see section 2.1) which includes the virtual page number of the new container and the disc address, and its own process identifier (allowing the disc process to return to it). Pass this to the appropriate disc process, requesting a write operation (see section 4.7.5).

The disc process completes the write and returns, without locking the new container's page 0 into the main memory.

Finally the instruction prepares an owner capability for the new container and returns this to the Container Manager, after unlocking its page 0.

6.2 Preparing the New Container for Use

The Container Manager then sets up the organisation described in chapter 19 section 7 and following sections, i.e. it creates a co-module table (see Figure 19.6) and a code table (Figure 19.8), etc. This stage can involve considerable dialogue with the user with respect to which Virtual Page Table Manager, which Segment Manager, which Co-Module Manager, which Code Manager, which Error Manager(s) should be installed (if a choice is provided), and is primarily an operating system issue rather than a kernel issue.

When the Container Manager has completed this stage, it returns an owner capability for the container and a capability (not necessarily an owner capability) for each co-module installed in the container. Using these, the user can then initialise the individual co-modules.

7 Copying a Container

Various techniques for copying containers were discussed in chapter 19, but without providing a detailed explanation of how an implementation can be achieved. This chapter has added the necessary additional background knowledge to consider how the copying of a container can actually proceed in the context of the first and certainly most important possibility mentioned in chapter 19 section 13 ("to make a copy which the owner, or some other user, can use independently of the original"), using the page-by-page method.

The Container Manager provides a **copy** command for user threads. This normally requires at least three input parameters:

- a valid capability for the file to be copied,
- a capability for the disc on which the copy is to be located[116], and
- an indication whether the owner of the original should remain the owner of the copy or whether the owner of the requesting thread should become the owner of the new copy.

The copy command first carries out appropriate checks, including establishing that

- the capability for the file to be copied has the appropriate access rights to allow the copy to be made (including a copy with owner change, if this is requested, see chapter 26),
- the disc capability allows the user to create files on the disc, and

[116] If the second parameter is null, this might be interpreted to imply that the boot disc should be used. But that should be a decision take in the container manager, which could then maintain a default (such as the boot disc) or even different defaults for different users.

- the container does not contain troublesome capabilities (which can be established by calling the container's Segment Manager).

If these tests are passed, the Container Manager then passes the parameters to the kernel's `copy` instruction, which actually carries out the copying.

7.1 The Kernel's `copy` Instruction

The kernel first needs to carry out some further checks, e.g. to ensure (from the Local Mount Table) that the source and destination discs for the copy operation are both mounted locally. When this has been clarified, the kernel's copy process proceeds as follows.

i) It claims a new virtual memory message block of the type "request and lock page 0" (see section 4.4) and enters into this the virtual memory address of page 0 of the container to be copied) then passes it to the VM Process. (If the page is not already in the main memory, the VM Process organises for it to be paged in and in this case suspends the user thread executing in the Container Manager's copy routine.)

ii) It examines page 0 of the container to be copied to determine the pages to be copied and creates a page-sized buffer in the main memory. It copies the appropriate entries from the old page 0 into this, making a change of ownership (if this is permitted in the generic rights of the original capability, which should have been checked by the Container Manager). It zeros the page table entries in the new page 0.

iii) It creates a new container for the new copy (see section 6) and writes the new page 0 into it.

iv) It then carries out a page by page copy of page 1 to the end of the original container, by calling the disc process(es) in a loop, modifying the new container's page table on each loop cycle.

7.2 The Page by Page Copy Mechanism

This mechanism does *not* use the Segment Manager to create/copy segments, but works entirely at the page level. This is not problematic, since all the pointers in a container are addresses relative to the beginning of the container, and a page by page copy does not affect the positions of segments. Furthermore, if the Segment Manager has placed co-module data (its own or that from some other co-module, including from a user co-module) in page 0 of the original file, this will be copied automatically into page 0 of the new container. (In this way a trivial file might be completely copied simply by writing page 0 – as described above – without a further loop.)

The actual reading of pages of the copied container and writing them to the new container in a loop is a straightforward procedure which is not here described in detail. For each read and for each write operation the thread requesting the copy operation must be suspended in order to allow other kernel processes/user threads to be executed, and reactivated on completion of the disc operation. The only complication is that when a thread is reactivated and is scheduled by the UTS it must know where the copy operation must be resumed. I suggest that this is noted in page 0 of the *new* container (because in a multiprocessor system the same container may be copied in parallel to multiple containers).

If the copy is successful, the kernel creates an owner capability for the new file and returns it to the Container Manager; this in turn returns the new owner capability (with all access rights set) to the requesting thread.

Part 6
Security Mechanisms

Chapter 24
Qualifiers with Bracket Routines

A *qualifier*, or *qualifying module*, is a module which can be associated with other modules in order to qualify (modify) the effects of their execution. The idea, originally called "attribute types" [3], occurred to me in 1996 while on study leave at the University of Sydney. It was further developed by my research group at the University of Ulm in the context of modular programming design methods, and was later incorporated into our object oriented programming language Timor, a new language developed to provide programming language support for the novel features of SPEEDOS which cannot be programmed in normal programming languages [7]. The most significant features of *qualifying types* (the name used in connection with Timor and SPEEDOS) were described towards the end of volume 1 chapter 13. The reader may at this stage find it useful for understanding the following to re-read the relevant section of Chapter 13 and to study the related diagrams. This chapter describes how qualifiers can be integrated into the SPEEDOS kernel.

1 Basic Principles of SPEEDOS Qualifiers

A qualifier is in most respects a normal module. It can have normal semantic methods, it can have persistent data structures, and its normal semantic routines are explicitly invoked in the usual way via an inter-module call. In addition it has bracket methods, which cannot be invoked explicitly, but can be associated with the normal methods of other modules in order to qualify them. Bracket methods can access the persistent data of their own module and can invoke its internal routines.

1.1 Timor Qualifiers

In Timor, qualifiers can be *statically* embedded into other modules. In this case SPEEDOS is unaware of the bracket methods; it is the responsibility of the compiler to organise the appropriate code sequencing. Alternatively Timor qualifiers can be *dynamically* associated with small objects in Timor programs. Such qualifiers are entirely handled by the Timor run-time system. The third possibility, which is the subject of this section, is the SPEEDOS kernel mechanism which associates qualifiers with the semantic routines of entire SPEEDOS modules.

1.2 Call-in and Call-out Brackets in SPEEDOS

What makes a qualifier special is that a normal module (or indeed a qualifier), here called the *qualified module*, can be qualified by zero or more qualifiers, which may have two different kinds of *bracket routines* (*call-in* and/or *call-out* brackets). Call-in brackets "catch" incoming calls to the qualified module and can execute their own code as a *prelude* to executing a *body statement*, which causes the qualified module's semantic routine (or the next bracket routine in the list) to be invoked. When this finally exits it returns not to the client module which originally called the qualified module's semantic routine but to the call-in bracket routine at the instruction following the `body` statement, known as a *postlude*. After the postlude exits, control is returned to the client module (or the previous bracket routine in the list). If the call-in bracket exits (via a `bracket_return` instruction) without executing a `body` statement the target routine of the qualified module is not called, and the call-in bracket simply exits back to the client module (or an earlier bracket routine in the list).

The second kind of bracket routine, *call-out brackets*, has no influence on calls into the qualified module, but is activated when the qualified module attempts to make calls out to further modules (here called *target* modules). Call-out brackets "catch" such calls and execute a prelude. This can optionally be followed by a *call statement*, which allows the appropriate routine of the destination module to be entered. When this finally returns, it continues at the postlude following the call statement, and finally a return is made to the qualified module. If a call statement is not executed in the call-out routine, the call never reaches the intended module but returns to the postlude of the previous bracket.

The same qualifier module can support both call-in and call-out bracket routines.

These above possibilities are illustrated in Figures 13.2 to 13.7 in volume 1. The SPEEDOS kernel supports `body` and `call` instructions corresponding to the Timor body and call statements. These statements, along with the `bracket_return` instruction, can only be used in bracket routines. Normally such an error can be detected at compile time and should therefore not arise at run-time. However, a run-time error occurs if they are executed in normal modules at run-time.

1.3 Multiple Qualifications

A list of call-in and/or call-out qualifiers can be associated with a qualified module. In this case a `body` or `call` statement causes the next bracket in the list to be executed, or in the case of the last bracket in a list the appropriate semantic routine of the target module is called. The order of accessing is determined by the position of the qualifier in the list, which is held in a *qualifier list module*. The ordering can be significant in its effect, e.g. if the first bracket method exits without invoking a `body` (or `call`) statement, the second bracket routine in the list will never be invoked.

1.4 Sequencing of Bracket Routines with Qualifier List Modules

In the case of qualifiers dynamically associated with SPEEDOS modules, the kernel organises their execution. This is possible because the latter is responsible for all inter-module and similar call invocations. In SPEEDOS a bracket routine is not permitted to execute the `body` instruction or the `call` instruction more than once within a single bracket routine. (A synchronous interrupt occurs if a normal semantic routine issues these instructions, if a call-in bracket issues a `call` instruction or if a call-out bracket issues a `body` instruction.)

The organisation of lists of qualifiers for a qualified module is not a trivial activity and is therefore not directly undertaken by the kernel. Instead it relies on privileged co-modules, known as Qualifier List Modules (QLMs), to provide semantic methods which allow the owners of modules to organise lists (e.g. by adding and removing entries and changing their sequencing).

Whereas at the programming language level (i.e. internal to a module) it only makes sense to associate bracket methods with "objects" (in the sense of object-oriented programming, where an object consists of a fixed combination of data and code), at the operating system level it is sensible to associate separate qualifier lists with the data file and the code file of a module. One reason for this is that the code module may need to be separately protected (e.g. to guarantee the rights of software providers) while the data file, which may have a different owner, needs to guarantee the latter's protection and security rights.

For a user defining a list of bracket routines, the order in his qualifier list module is first the call-in brackets then the call-out brackets. This applies both to QLMs associated with a data file and to QLMs associated with a code file.

Capabilities for data file QLMs are located in the Co-Module Table and those for code files in the Code Table, where they are accessible to the kernel[117]. In this way, on receiving an inter-module call the kernel can gain access to the qualifier list modules.

[117] see volume 1 chapter 19 sections 7 and 9

The system applies the brackets in the following order: (for each inter-module call made *to* the qualified module) first the call-in brackets for the code file, then the call-in brackets for the data file[118], and (for each inter-module call made *from* the qualified module) first the call-out brackets for the code file, then the call-out brackets for the data file. The effect of this is that if a module A calls a module B in the course of its execution, then the code call-out brackets for A, followed by the data call-out brackets for A are executed. These are followed by the code call-in brackets for B then the data call-in brackets for B; subsequently module B is executed.

1.5 Qualifier List Modules

Qualifier list modules (QLMs), which contain module capabilities for the individual qualifier modules with the appropriate bracket routine numbers, have the following duties:

i) to provide the owner of the module with a convenient interface for managing the lists (e.g. adding new entries, removing existing entries, reorganising the order);

ii) to organise the mapping of symbolic names known to the user (e.g. for the semantic routines, the bracket routines and parameter names) onto the entry point numbers needed by the kernel. For this purpose it calls a further module which will be described in connection with command interpretation (see Chapter 32 section 2);

iii) to provide the kernel with information via which it can quickly discover the details it requires to invoke the bracket routines in the correct order;

iv) to indicate to the kernel whether a bracket routine requires no access, read-only access or read-write access to the qualified module's parameters;

v) to advise the kernel whether a bracket routine can make normal inter-module calls.

The details of the QLM interface for the kernel are predefined, but the remaining aspects of qualifier list modules are freely programmable and can differ from module to module.

Since the kernel relies on these modules in ways which affect system security, the Co-Module and Code Managers should take special precautions to ensure their trustworthiness, when setting up entries for them.

1.6 Bracket Routines and Parameters

Bracket routines do not have parameters of their own, but can in some cases ac-

118 The sequence for the call-in brackets differs from that proposed by Espenlaub in his thesis [4, p. 183].

cess the parameters of the modules which they are qualifying, depending on the way the way the bracket routines have been defined. They can be defined to be activated

- when *any* method of a qualified module is invoked[119]. In this case they have very limited access to the parameters being passed or returned, since they have no knowledge of the structure and purpose of the parameters.
- specifically for reader routines (which can be recognised in Timor by the keyword **`enq`** (i.e. *enquiries* which do not modify the state data of their module, i.e. readers) or for writer routines by the keyword **`op`** (i.e. *operations* which can modify their state data)[120]. Also in such cases the bracket routine has very limited access to the parameters passed by the caller (since these vary from routine to routine).
- when a *specifically named* routine is called, in which case they may have access to the routine's parameters (in read-only or read-write mode, as defined in the qualifier list module).

The limited forms of access available to the first two categories are described in section 3.3 below.

Permitting read-write mode for parameters gives a bracket routine the opportunity to modify the input or return parameters which are passed via a `body` or `call` statement to/from a qualified module.

2 An Overview of the Execution of Bracket Routines

When a module executes an inter-module or similar call, the kernel checks the Co-Module and Code Tables of the module to ascertain whether there are associated qualifiers, and if so it activates the bracket routines in the defined sequence (i.e. the call-out brackets of the calling module followed by the call-in brackets of the called module).

After the call-in brackets have been executed (and assuming that no bracket routine has issued a `bracket_return` statement without issuing a `call` or `body` instruction), the target module is activated in the usual way.

If an inter-module call instruction is executed in a qualified module, the kernel checks whether this is allowed, and if so places the normal IMC linkage segment on the current thread stack (see Figure 20.6). It then checks whether the *calling* module has call-out brackets, and if so executes these one by one on the user thread's stack, separating them by bracket linkage.

After the last call-out bracket has completed its prelude via a `call` instruc-

[119] Note that **open** and **close** routines are never bracketed.

[120] The rights with respect to accessing state data are set for `enq` routines to read-only and for `op` routines to read-write.

tion (i.e. assuming that it does not make an early return), the kernel checks whether the *called* module has call-in brackets. If so, they are executed after the last call-out bracket of the calling module (if any). Again these are separated by bracket linkage segments.

When a *module* returns, bracket execution resumes in the postlude of the last called bracket routine, and when this issues a `bracket_return` instruction control returns to the postlude of the previous bracket routine, etc. until the calling module is reached. This continues its normal execution and may call further modules. In this case also, call-out bracket routines might be needed and in some cases these might differ from those previously used (e.g. if specifically named modules/semantic routines exist in the call-out list).

A bracket routine may also call a further *module* (which might or might not be qualified) while it is executing a postlude, and this is handled in the usual way.

3 Managing Bracket Parameters:

The handling of parameters in bracket routines is not as straightforward as one might think. Here are Espenlaub's comments about isolating brackets from each other. He points out a potential security risk with the straightforward approach:

> "The brackets having access to parameter information must be isolated against each other, i.e. the modifications to the parameter list by an inner bracket must not be visible to an outer bracket. This requires saving of the received parameter list and restoring it when the next inner bracket returns. Information leaks through manipulated parameter lists must be prevented, e.g. by marking parameter lists as read-only data once `body` has been invoked." [4, p. 136]

In fact not only the outer brackets are affected, but also the calling module. For example, if a module passes a capability as a parameter to a further module and a call-out bracket invalidates this for security reasons, the caller (possibly a hacker attempting to pass (dis)information via the capability) would be able to detect that this has happened, thus arousing his suspicion that he is being monitored, whereas the intention of the hacked site may be to continue to document his hacking efforts.

The above comments are based on the view that the parameters in bracket routines, when modified by one routine, are visible in the next bracket routine(s) and later in the postludes of all the bracket routines. However, the design for parameter passing protocols has changed in the current version of SPEEDOS, as is described in chapter 20 section 6.2. The key change from the present standpoint is that *after* an inter-module call instruction has been executed Segment Register 2 (which had been used to prepare the parameters for the call which has now terminated) is always invalidated before the called module returns. Hence on return to a calling module the code of the module cannot see whether or not the input parameters which it prepared for the call have since been changed by a bracket routine.

3.1 Managing Input Parameters in Bracket Routines

We now turn to the actual management of parameter segments for bracket routines. It is important to note at this point that many (probably most) bracket routines, including many of those which have the aim of improving security, have *no* access to the parameters created by a calling module. And even those which do have such access often merely read the parameters. Hence only in fairly rare cases are parameters modified. If they were never modified at all, it would be sensible simply to allow the preludes of those bracket routines which read the parameters to do so directly from the parameter segment which the calling module has created via Segment Register 2, by pointing Segment Register 0 of each bracket routine to the same segment on the stack.

So long as no modifications are made, this is a sensible policy. It has the advantage that parameter segments need not be copied (possibly several times) on the thread stack. Furthermore new output parameters (normally addressable by Segment Registers 2 and 3) would never have to be created. Instead, on a bracket routine invocation the kernel would simply copy the previous caller's Segment Register 0 value for use in the new routine, but changing the access rights in conformance with the information which it has received from the qualifier list module for the new bracket routine. (Since a routine *always* at most has read-only access to its input parameters the access rights will vary between no access and read-only, but *never* read-write.)

But if this policy is pursued, what happens when a bracket routine needs to modify the input parameters? Here is the solution.

Initially when a bracket routine is invoked, the kernel invalidates Segment Registers 1, 2 and 3. It sets Segment Register 0 to point either to the original input parameter segment or to a modified version of this, now to be described.

Just as there is a kernel instruction `create_imc_params` which allows a *module* to create an output parameter pair when needed[121], so also there is a further kernel instruction `change_bracket_input` (with a single operand specifying the length of the data partition[122]). This can only be executed in a *bracket routine*, and only in cases where the bracket routine plans to change the input

[121] Bracket routines may not execute the kernel instruction `create_imc_params`.

[122] The structure of the new segment is based on that of the input parameter segment addressed via Segment Register 0, but the length of the data can be varied, e.g. in order to vary the length of a string.

parameters which it receives from either the calling module or an earlier bracket in the list (i.e. those which it addresses via Segment Register 0). On receiving such an instruction the kernel first checks the information which it has received from the qualifying list module to ensure that this bracket routine has permission to make changes. If so, it creates a new input parameter segment at the top of the thread stack which it makes addressable to the bracket routine via Segment Register 2 in read-write mode. The new segment has the same structure (e.g. number of module capabilities) as that which it received via its Segment Register 0, except that the length of the data partition may differ. Into the new segment it can then write new parameters (which may in part be copied from its own input parameter segment), see Figure 24.1.

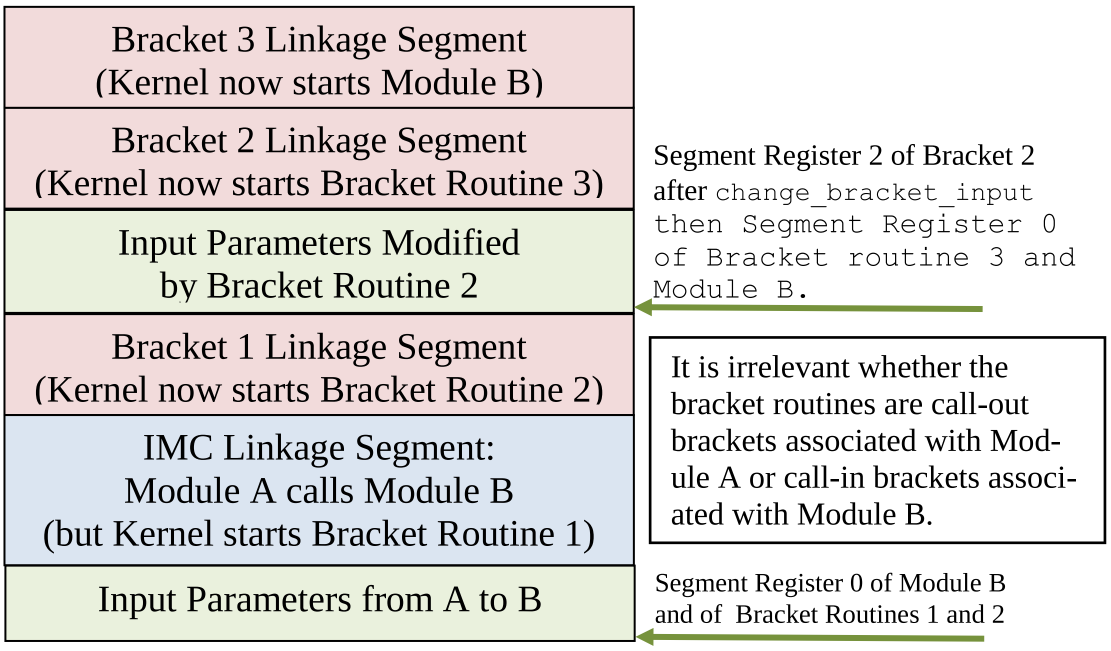

Bracket 3 Linkage Segment
(Kernel now starts Module B)
Bracket 2 Linkage Segment
(Kernel now starts Bracket Routine 3)
Input Parameters Modified by Bracket Routine 2
Bracket 1 Linkage Segment
(Kernel now starts Bracket Routine 2)
IMC Linkage Segment: Module A calls Module B (but Kernel starts Bracket Routine 1)
Input Parameters from A to B
Segment Register 2 of Bracket 2 after change_bracket_input then Segment Register 0 of Bracket routine 3 and Module B.
It is irrelevant whether the bracket routines are call-out brackets associated with Module A or call-in brackets associated with Module B.
Segment Register 0 of Module B and of Bracket Routines 1 and 2

Figure 24.1: A Thread Call Stack with Bracket Routines which modify the Caller's Input Parameters

When the bracket routine issues a `body` or `call` instruction, the kernel

a) stores Segment Registers 0, 1 and 3 in the bracket linkage segment with the access rights which they currently have, and stores Segment Register 2 in the linkage with the access rights set to *no access* (thus ensuring that it cannot change this segment in the postlude).

b) stores appropriate other register values of the bracket routine in the bracket linkage area.

c) sets the called[123] routine's Segment Register 0 to the values in the previous routine's Segment Register 2, setting the access rights to read-only or no access as appropriate. In this way the input parameters for the called mod-

[123] This may be either a bracket routine or the target module's semantic routine.

ule may be modified more than once, but the called module will see only the final version.

3.2 Return Parameters

Assuming that the target (called) module is eventually activated[124], it receives its input parameters (in read-only mode) as described above, and before returning it may place some results in its output segment via Segment Register 1 (in read-write mode). Some bracket routines may wish to examine and possibly change these (e.g. by invalidating a module capability).

Notice at this point that we cannot simply reverse the technique described above when dealing with returned parameters, since (unlike the input parameter case) the return parameter segment would normally be deleted (as part of the normal stack discipline) when an exit is made to a bracket routine! Instead we use the *original return parameter* segment of the caller as the anchor segment.

What this means is that when a called (target) module wishes to return parameters, it stores these in the segment which the calling module actually created for its return parameters, using Segment Register 1 as usual. Notice that this may be some distance down the stack, with bracket linkage and possibly modified input parameters separating it from the current stack top, but reachable by Segment Register 1.

This will work well so long as bracket routines do not modify the result parameters. However if a bracket routine (with appropriate access rights) wishes to modify the return parameters, it executes the kernel instruction `change_ bracket_result` (with a single operand specifying the length of the data partition) which may reduce the length of the data section while adhering otherwise to the original segment structure. This instruction first checks that the bracket routine in question has permission to change the result, and if so creates a new return segment immediately above the existing return parameter segment. To make this possible the first bracket linkage segment is placed after a gap on the stack top which is sufficiently large to hold a copy of the return segment[125]. The kernel then makes the segment addressable to the bracket routine in the normal way, via Segment Register 1, which is set to read-write access (only if read-write access is allowed for the bracket routine). When the bracket routine returns to its caller (which could be the original calling module or a lower bracket routine) the kernel makes this segment addressable via Segment Register 3 (in read-only or no access mode).

[124] This is the case if all the bracket routines issued a `call` or `body` statement.

[125] The stack space for a second return segment must have the same structure as that of the original return parameter segment.

If more than one bracket routine exercises its right to modify the return parameter segment the two return parameter segments are organised by the kernel to function alternately in a flip-flop manner, simply by clearing the segment which holds the older information and resetting the segment register values accordingly. Notice that by using this technique the original information passed as return parameters can get overwritten. This would be the case anyway if a normal stack discipline were followed. But if such information is important (e.g. for debugging purposes) then a bracket routine can make an inter-module call to a logging module to preserve information which might otherwise be lost.

Figure 24.2 shows the effect of Bracket Routine 2 modifying the return parameters, using the kernel instruction `change_bracket_result`.

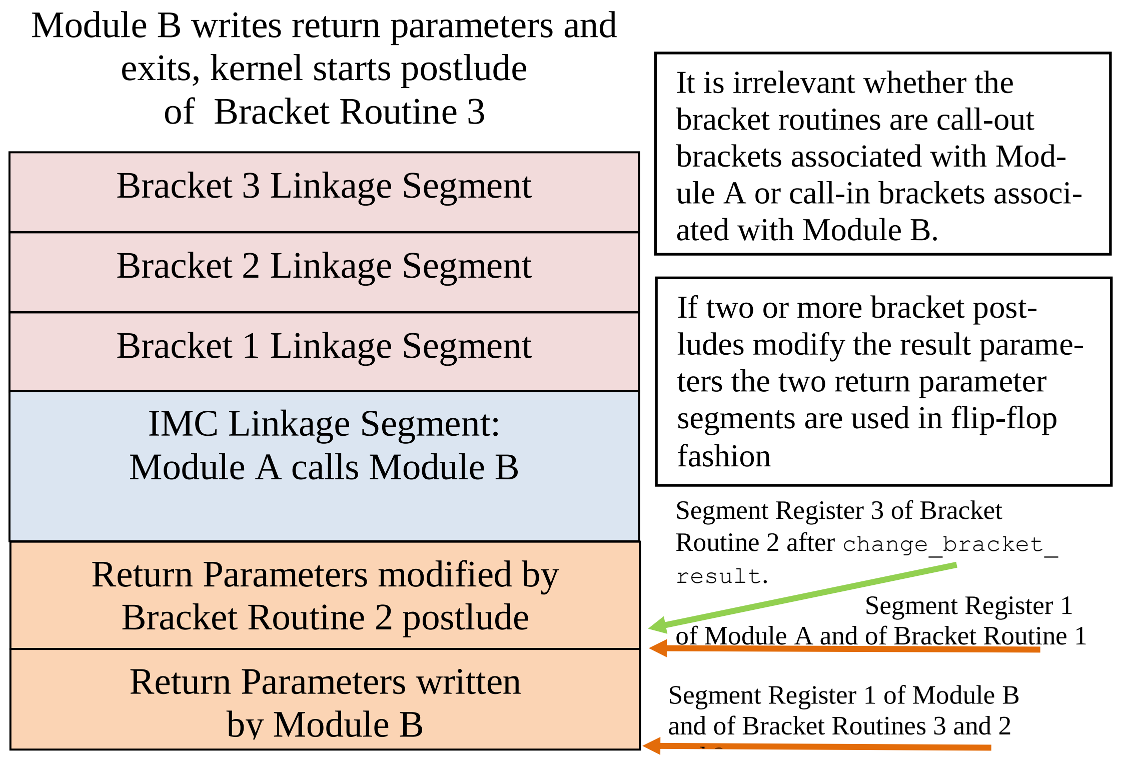

Module B writes return parameters and exits, kernel starts postlude of Bracket Routine 3
Bracket 3 Linkage Segment
Bracket 2 Linkage Segment
Bracket 1 Linkage Segment
IMC Linkage Segment: Module A calls Module B
Return Parameters modified by Bracket Routine 2 postlude
Return Parameters written by Module B
It is irrelevant whether the bracket routines are call-out brackets associated with Module A or call-in brackets associated with Module B.
If two or more bracket postludes modify the result parameters the two return parameter segments are used in flip-flop fashion
Segment Register 3 of Bracket Routine 2 after change_bracket_result.
Segment Register 1 of Module A and of Bracket Routine 1
Segment Register 1 of Module B and of Bracket Routines 3 and 2

Figure 24.2: A Thread Call Stack with Bracket Routines which modify the Callee's Return Parameters

In this scheme, which ensures high efficiency whilst addressing Espenlaub's point at the beginning of section 3, the input and return parameters are decoupled for bracket routines.

3.3 Access to Parameters in Routines which are not Specifically Named

The main reason why bracket routines which are not specifically named have very limited access to parameters and returned results is because these have no knowledge of the structure (e.g. number of capabilities, length and purpose of the data partition and (in the case of co-module calls and library calls) the structure and purpose of pointers). However, the kernel knows the structure of parameter segments, and can therefore carry out simple security tasks on behalf of such bracket routines. In particular it can invalidate capabilities in the capability partition, pointers in the pointer partition (if any) and data in the data partition[126]. Such an action can be useful in general brackets in order to ensure that no information leaks from a module (e.g. if the owner of the data suspects that the code of his module is releasing its file information to the manufacturer of the code). Since such information might be released in calls to modules and in returns from modules, two kernel bracket instructions `invalidate_output` and `invalidate_input` are provided by the kernel. These instructions (which have operands to further specify the relevant partitions) can only be used in the bracket routines associated with the data file of a module (i.e. those specified in the module's Co-Module Table).

4 Implementing Bracket Routines

Having established the basic principles of qualifying types, we now consider in more detail how the kernel can implement bracket routines and how it acquires the information needed to do this from qualifier list modules. We begin by considering what happens when an IMC is received by the kernel for a module which is qualified by bracket routines.

4.1 Handling Inter-Module Calls

When the user request process receives a request for an inter-module call, it first creates a new inter-module call linkage segment for the calling segment. It then ensures that the request is valid (e.g. that the requested semantic routine is allowed by the access rights in the capability). The kernel then creates an IMC record on the stack which serves as an indication of the thread's progress; the kernel stores into it the operands of the IMC (capability, semantic routine number, read-only flag). If either or both the Co-Module Table and the Code Table include a capability for a QLM, these are also copied into the IMC stack record.

At this stage the kernel cannot simply execute the inter-module call, because bracket routines may first need to be activated. These bracket routines (as illustrated in Figure 24.2) may in the order of their execution be

- call-out brackets associated with the code of the calling module,
- call-out brackets associated with the data of the calling module,
- call-in brackets associated with the code of the called module,
- call-in brackets associated with the data of the called module.

[126] The easiest way to invalidate data is to change its length in the red tape to 0.

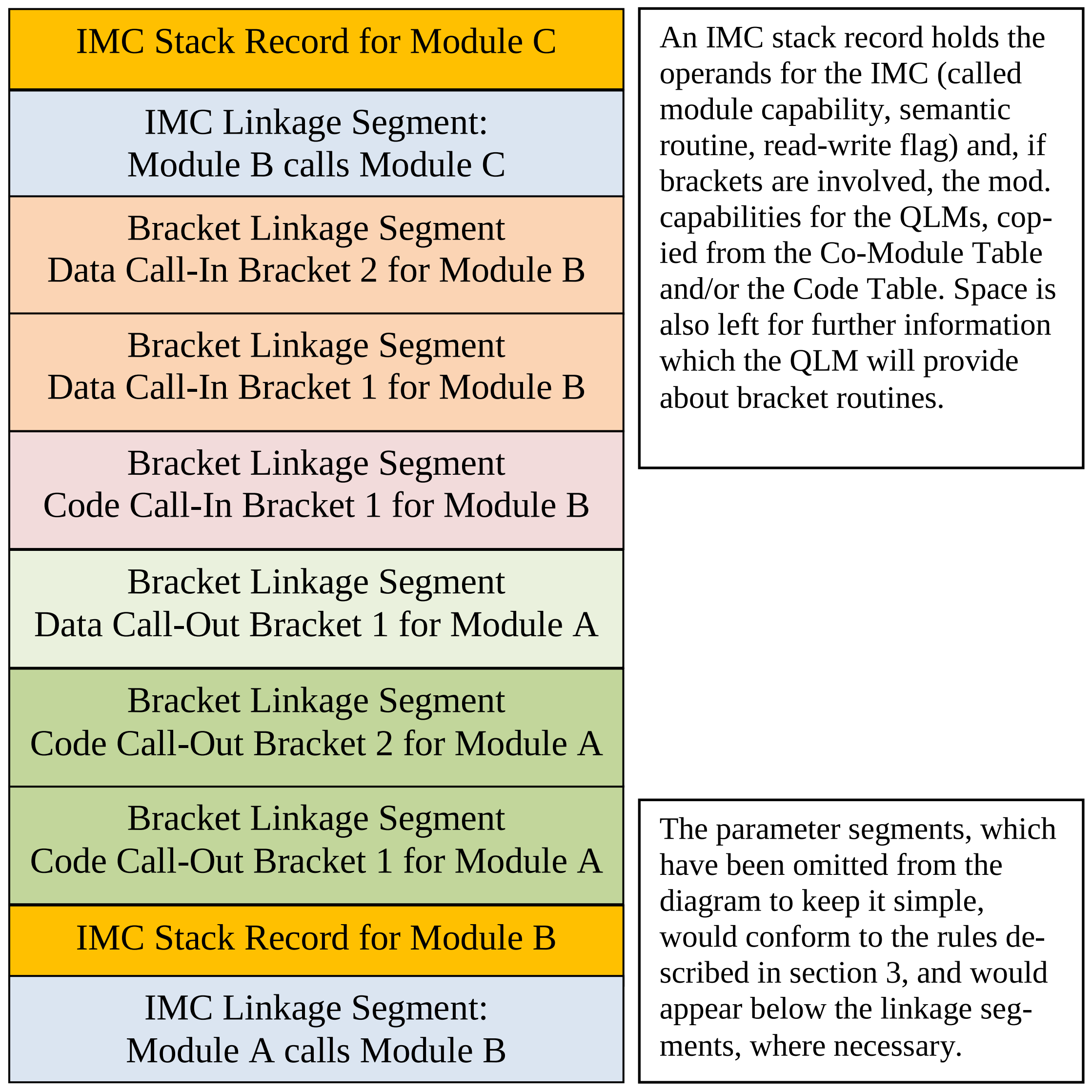

IMC Stack Record for Module C
IMC Linkage Segment: Module B calls Module C
Bracket Linkage Segment Data Call-In Bracket 2 for Module B
Bracket Linkage Segment Data Call-In Bracket 1 for Module B
Bracket Linkage Segment Code Call-In Bracket 1 for Module B
Bracket Linkage Segment Data Call-Out Bracket 1 for Module A
Bracket Linkage Segment Code Call-Out Bracket 2 for Module A
Bracket Linkage Segment Code Call-Out Bracket 1 for Module A
IMC Stack Record for Module B
IMC Linkage Segment: Module A calls Module B
An IMC stack record holds the operands for the IMC (called module capability, semantic routine, read-write flag) and, if brackets are involved, the mod. capabilities for the QLMs, copied from the Co-Module Table and/or the Code Table. Space is also left for further information which the QLM will provide about bracket routines.
The parameter segments, which have been omitted from the diagram to keep it simple, would conform to the rules described in section 3, and would appear below the linkage segments, where necessary.

Figure 24.3: A Thread Call Stack with Bracket Routines

Figure 24.3 illustrates a thread stack after a module A, with three call-out brackets, has called a module B, which has three call-in brackets, and module B then calls a further module C.

4.2 Summarising the Handling of Brackets

When a new IMC occurs, the user request process checks whether any call-out brackets are required by the calling module. These (and any call-in brackets associated with the called module) must be executed before the new IMC can be executed.

It can establish this by examining the *previous* IMC stack record on the stack. (At the time of the call this is actually the "current" IMC stack record.) If this indicates that call-out routines are defined, it executes these, and then evaluate whether call-in bracket routines are required for the called module by examining the new IMC stack record. If not, it can then execute the IMC as normal. If call-in bracket routines are required the user request process must acquire the information from the appropriate QLMs and add it to the new IMC stack record. It must then execute the required call-in brackets if necessary and can then execute the new IMC itself.

4.3 How the Kernel Obtains Bracket Information from QLMs

The kernel's user request thread uses surrogate threads, in this case called QLM threads, to obtain the information which it needs from QLMs. We now describe this procedure in more detail. Since the basic working of surrogate threads has already been explained in principle (see chapter 22 section 8.2) it is now simply assumed that a pool of QLMs has been set up and partly prepared at system initialisation in a separate container (the QLM process container), and that the kernel

- allocates and deallocates the individual threads as required, using a resource set semaphore;
- completes their initialisation before activating them[127];
- schedules and suspends them via requests to the UTS.

4.4 Acquiring General Bracket Routine Info from a QLM Thread

We now describe how the kernel acquires general information about the bracket status of a new IMC.

When the new IMC first reaches the kernel, it does not know whether call-in or call-out brackets or both are defined in the QLM file(s). (This cannot be defined statically, since the QLM file can be modified – subject to synchronisation requirements – at any time by the user.) Hence when the kernel has activated the surrogate threads and has requested the kernel interrupt process to suspend the current thread, each QLM thread begins to execute; its first task is to advise the kernel how many call-in and call-out brackets are defined in its QLM for the target module.

It supplies this information in a kernel instruction `bracket_info`, which passes a bracket parameter block with the appropriate information, including (for identification purposes) a copy of the module capability for the QLM and a copy of its own surrogate thread capability to the kernel (see Figure 24.4).

[127] In this case a QLM thread needs only two input parameters (its own QLM capability and the user thread capability, both for identification purposes). Otherwise its job is already well defined and only varies in that each can access its individual QLM via segment register 5, which has been initialised to address the root data segment of the QLM in question.

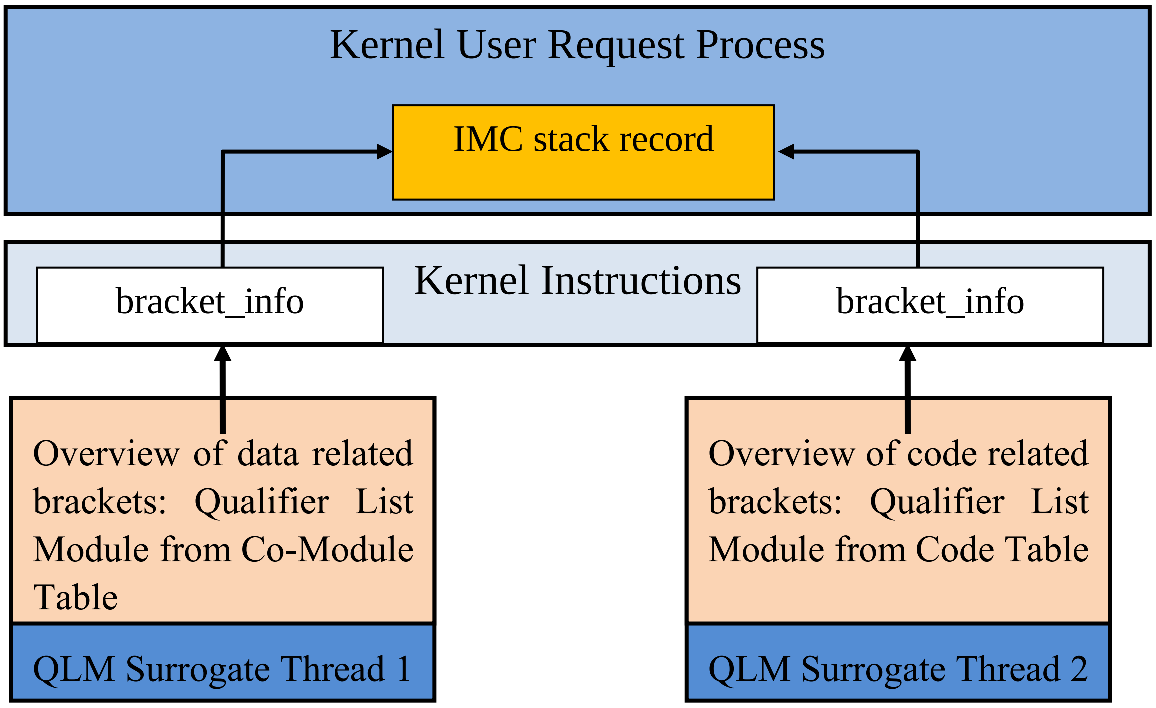

Kernel User Request Process
IMC stack record
Kernel Instructions
bracket_info
bracket_info
Overview of data related brackets: Qualifier List Module from Co-Module Table
QLM Surrogate Thread 1
Overview of code related brackets: Qualifier List Module from Code Table
QLM Surrogate Thread 2

Figure 24.4: General Bracket Info from QLM Threads to Kernel

When each QLM thread has provided its bracket parameter block, it suspends itself by calling the UTS. But before we can consider in more detail what follows next, we need to know how bracket routines affect linkage segments on the user thread stack.

4.5 Controlling the Execution of the Bracket Routines

In order to discover the details of the individual bracket routines, the kernel must activate the individual QLM thread(s) in the defined sequence to obtain the information about the first/next bracket routine. What now follows can therefore be viewed as something similar to co-routine activity or to a producer-consumer situation between a QLM thread (advising of the next bracket) and the user thread (executing this bracket) operating repeatedly until the last bracket has been executed. But this simple view is an oversimplification, because the kernel has to sit between the two and organise the execution of the bracket routines.

4.6 Managing the Linkage

Before a bracket routine can be executed the kernel must first store linkage for the routine which was previously executing. In the case of the first call-out bracket (or, if there are no call-out brackets, the first call-in bracket) this will be the normal linkage for an inter-module call (see Figure 20.5).

When a bracket routine issues a `body` or `call` instruction the kernel likewise places a linkage segment on the stack. This has a different form from an

IMC linkage segment, as is illustrated in Figure 24.5.

Linkage Type
Kind of Bracket Routine
Info on Parameter and Call restrictions
Kernel Pseudo-Register Values
Code Segment Register and Program Counter
Parameter Segment Registers 0 – 3
Saved Segment Registers
Bit List of valid Segment Registers
Caller's Local Base on Kernel's Linkage Stack
Previous Top of Stack

Figure 24.5: A Bracket Routine Linkage Segment

4.7 The QLM Thread Provides Information about a Bracket Routine

The kernel's user request process determines which QLM thread should be restarted[128] first, and activates it by placing a request in the input buffer of the user interrupt process. It then issues a kernel reschedule.

When the QLM thread starts executing, it examines its list and selects the first bracket routine to be applied, setting up the following parameter block details for the kernel:

a) The module capability by which its bracket routines can be invoked.

b) The entry point number of the appropriate bracket routine.

c) An indication whether the bracket routine has no access, read-only access or read-write access to the input and return parameters.

d) An indication whether the bracket routine is designed always, sometimes or never to invoke the kernel's `body` instruction.

e) An indication whether the bracket routine is permitted to make normal inter-module calls.

f) An indication whether the current bracket is the last call-in (or call-out) bracket in the list.

g) Its own surrogate thread capability and QLM capability as well as the user thread capability as identification.

The QLM thread passes this information to the kernel, using the kernel instruc-

[128] The QLM thread already exists and is suspended awaiting a request for an individual bracket.

tion `next_bracket` (see Figure 24.6). It then suspends itself with the UTS (until the kernel releases it to supply information for the next bracket routine).

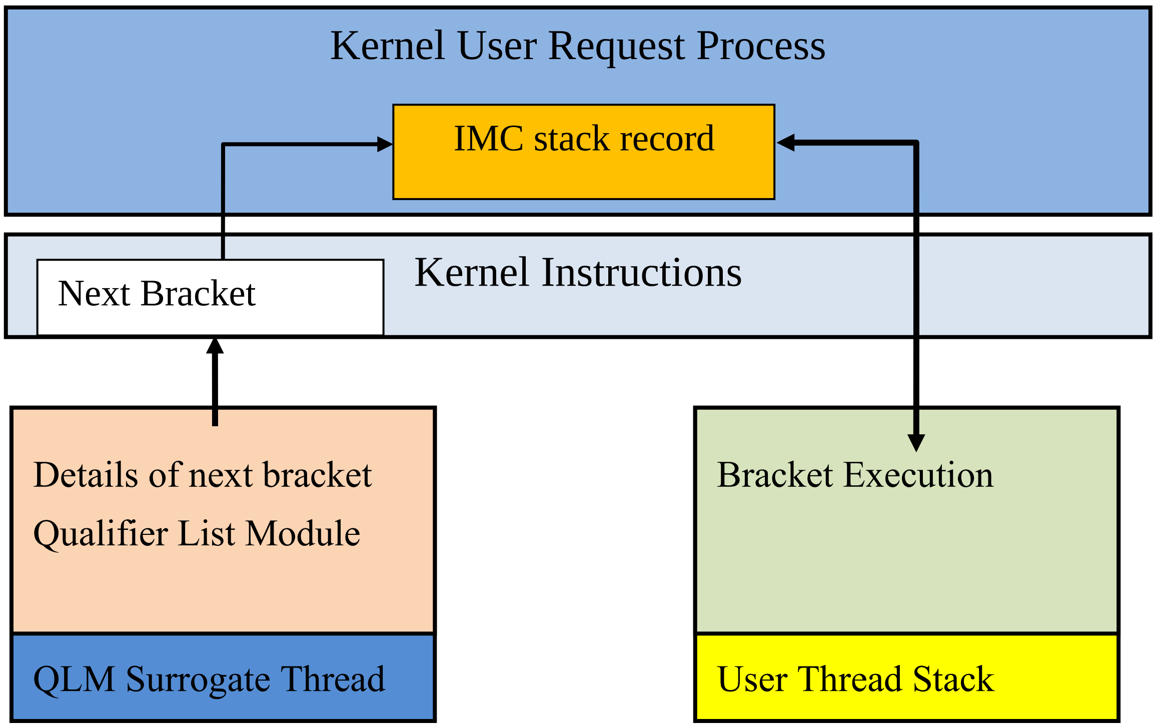


Figure 24.6: Information Flow from QLM to User Thread

4.8 Executing a Bracket Routine

The information available to the kernel's user request process is

a) that received from the QLM parameter block details in its current `next_bracket` instruction,

b) information about the qualifier module which it can obtain via the latter's module capability (e.g. the address of its root data and the start address of the bracket routine's code from its bracket entry point list (see chapter 19 section 9.3), and

c) information in the IMC stack record.

The user request sets up the register save area of the stack to point the image of segment register 5 to the bracket routine's root data (which is *not* that of the calling or called module!), its code segment register image and offset to the address at which it begins execution, and its parameter segment registers to address the input parameters supplied by the user thread (with the access rights indicated in section 3). It then places an interrupt message block in the kernel user interrupt process's buffer area to advise the UTS to activate the user thread.

When the bracket routine is executing this can use other kernel instructions where it has permission to do so (e.g. to discover environmental information (see chapter 26). If it attempts to make an inter-module (or equivalent) call, the kernel's user request process can check whether this bracket has permission to make inter-module calls. If so the kernel executes this as normal.[129] If it is not permitted the kernel generates a synchronous interrupt (see chapter 22 section 8.2)[130].

4.9 Executing `Body` and `Call` Instructions

When a bracket executes a `body` or `call` instruction, the kernel

- checks whether this is the appropriate instruction and that it is not being called for a second time. (If so, a synchronous interrupt is activated.)
- places a bracket linkage segment on the user thread stack (see Figure 24.3), paying attention to the parameter addressing rules described in section 3 above.
- locates the appropriate IMC stack record and checks whether this is the last call-in (or call-out) bracket to have completed executing its prelude. If not, it activates the next bracket by activating the appropriate QLM thread[131] (via the user interrupt process) and exits.

When the QLM thread issues a `next_bracket` instruction, the above procedure is repeated until all the call-in or call-out brackets have been executed.

If the final call-out bracket issues a `call` instruction this signals to the user request process that a new series of call-in bracket routines (if any) should begin.

If the final call-in bracket issues a `body` instruction this signals to the kernel that the target inter-module call should now happen.

4.10 The Postlude Phase

When a called module executes a `return` instruction, this might signal the start of the postlude phase for bracket routines. Hence the user request process checks at this point whether the last linkage segment on the user stack is a normal linkage segment or a bracket linkage segment.

If a bracket routine's postlude needs to be activated, the kernel's user request process loads the registers for the postlude from the linkage segment, paying attention to the rules for managing parameters in brackets, as described in section 3.3. The bracket linkage is then deleted from the top of the stack. The

[129] If so the new IMC may also be bracketed and the kernel must nest this on the stack in the usual way.

[130] The action taken depends on a setting in the Thread Security Register (see chapter 26).

[131] If the bracket issuing the `body` statement is associated with the data file and there are call-in brackets associated with the code file, it activates the code file QLM thread.

postlude is then executed and the bracket routine eventually exits using a kernel `bracket_return` instruction. This procedure is then repeated for each outstanding bracket postlude, until the linkage for the module to which the return is made is reached. In this case the user request process organises the resumption of the module's execution (including the deletion of the IMC stack record on the stack).

4.11 Executing an Inter-Module Call in a Bracket Routine

If a bracket routine issues an IMC, this is as usual directed to the user request process. Since bracket routines need permission to execute IMCs (even if they have a valid capability), the user request process checks whether this call is permitted. If it detects an illegal IMC attempt it generates a synchronous error interrupt. If all is well the kernel places bracket linkage for the calling bracket routine on the stack and causes the user request process to be reactivated to carry out the legal IMC. This then executes the IMC as normal (including any required bracket handling), setting up an initial secondary link in the current IMC message block (which will be returned to null after the new IMC sequence ends).

4.12 A Bracket Routine Executes a Bracket Return in its Prelude

This can occur, for example, when the bracket routine has detected a serious problems and wishes to abandon the call to the target module before this takes place. It is not particularly significant for the kernel which kind of bracket routine this was. It is merely a sign that the postlude phase has begun, and return postludes are then executed until the last call-out bracket (if any) exits. The kernel then deletes the IMC stack record on the stack and allows the calling module to execute its next instruction. The calling module is unaware that the call was abandoned unless the module abandoning the call indicates this in the return parameters (or logs an entry in a module accessible to the caller).

5 Bracket Routines and Free Capabilities

Free capabilities were introduced in chapter 18 section 8 and chapter 19 section 13.3 as a technique which allows n-ary access to files in order to overcome the strict rules of information hiding in situations such as the copying or conversion of a file, by permitting a module (under restricted circumstances) to use the kernel instruction `load_free_cap` in order to gain direct read-only access to the content of a file module. Write access is not permitted for free capabilities.

But this creates a rather tricky problem associated with the use of qualifiers. What happens when a free capability refers to a data file which is protected by qualifiers? For example, it may have an associated Qualifier List Module containing bracket routines which revoke capabilities, provide access control lists, log the use of the qualified module, synchronise access to the file or even prevent its use entirely.

Because access to free capability parameters is not via inter-module calls, existing bracket routines cannot be used in the normal way. There are several reasons for this, e.g.

- The concept of a `body` or `call` statement has no direct significance when applied to the loading of a free capability using a `load_free_cap` instruction (see chapter 18 section 8.1).
- Since a semantic routine is not being bracketed (as is the case in normal bracketing as described above), there are no parameters which might be read or modified.

For such reasons the solution is to allow a further kind of bracket routine to be defined.

5.1 Free Capability Brackets

Free capability brackets are defined in a similar way to other brackets except that they have no `body` or `call` statements (and therefore there are no prelude or postlude phases) and they have no access to inter-module call parameters nor to the file being passed.

It would be theoretically possible to give free capability brackets optional read-only access to the free capability file (e.g. to allows the bracket routine to scan for viruses, etc.) However the free capability mechanism itself can be used for such purposes, so that this would introduce a duplicated mechanism which would itself have to be protected!

When the kernel receives a `load_free_cap` instruction it first determines from the Co-Module Table of the (free parameter's) file container and from the Code Table (of the code module about to access the file) whether a QLM or QLMs exist for free capabilities (see "Free Cap QL Cap" in Figures 19.6 and 19.8). If so it activates a QLM thread.

The user request process activates each free capability bracket in turn. When the bracket module invokes the kernel's `bracket_return` instruction, the kernel proceeds to the next bracket. When the final bracket executes a `bracket _return` instruction, the kernel then executes the `load_free_cap` instruction and the application continues as normal.

Free capability brackets can prevent the use of a free capability (e.g. if a security breach is discovered) in that they can simply execute the kernel's `abandon _call` instruction, passing a code to identify the problem which it has found. This causes the thread to generate a synchronous interrupt.

It would be theoretically possible to introduce a further kind of bracket routine more or less equivalent to a postlude phase. But this raises the question at what point the "postludes" should be executed. In principle they could be triggered when the kernel recognises that the segment register which addresses the free parameter

- is re-loaded via a further `load_free_cap` instruction,
- is invalidated either from within the module in which it is initialised (as a result of an explicit request to invalidate it via the kernel's `reduce_ access_rights` instruction[132] or
- implicitly when the thread exits from the application module via an inter-module return[133].

This may appear to be overkill, since it is unlikely that such routines could increase the security of a system. Security tests (e.g. based on access control lists, capability revocation) are usually applied *before* a potentially dangerous situation arises, and this would imply that in the present context the bracket routines applied when the `load_free_cap` instruction is executed. However the lack of a postlude phase has disadvantages. For example it eliminates the possibility that brackets could be used to synchronise access to an entire file using bracket routines. This is unfortunate and therefore I propose that a group of "postlude" bracket routines also are made available.

With this solution all the normal security checks applying to the module can be made, e.g. to revoke capabilities, to apply access control lists, etc.

5.2 Effects of Free Parameter Bracket Routines on a User Thread Stack

The execution of "prelude" free capability brackets takes place at the current stack top, and proceeds in a similar manner to call-in brackets, except that they have no input or return parameters, hence the values for segment registers 0 to 3 are invalid. They use the same bracket linkage segments as normal bracket routines (see Figure 24.5), except that the "Kind of Bracket Routine" is not call-in or call-out but "free capability prelude"[134] brackets, which have no `body` or `call` statement (and therefore no "official" postlude).

The second group, free capability "postlude" brackets, which will probably by quite rare, should be executed when one of the circumstances listed in section 5.1 arises. But because of the stack discipline these cannot formally be regarded as postludes!

[132] This kernel instruction was not envisaged in Espenlaub's thesis.
[133] A call to a further module is not considered to be an exit from the module.
[134] An implementer might choose to reduce the size by allocating a new linkage type and eliminating the input and output segment registers, etc.

Separating "postlude" brackets for free capabilities in this way also solves another stack problem. A module can concurrently load and access more than one free parameter, with the result that more than one set of bracket routines may exist at the top of a thread stack. The first set of stack frames to be deallocated must be that at the top of the stack. Hence if a thread explicitly invalidates free parameter segment registers this would always have to take place in the reverse order from that in which they were loaded. But if postludes are separate routines, this problem does not arise.

I suggest that at the programming language level free capability bracket routines could still be defined as having a prelude and a postlude section, separated by a keyword such as `file`. But the compiler should be aware that it must compile these sections as *separate* free capability routines.

I have not provided an implementation but this should be fairly clear based on the analogy of the main bracket routine implementations described in earlier sections.

Chapter 25 The Confinement Problem: Some Principles

The confinement problem is one of the more serious problems which conventional security mechanisms fail to solve in a general way. As was mentioned in volume 1 chapter 3, it cannot be solved simply by setting appropriate access rights (in the sense of Lampson's Matrix), because the risk can arise in situations in which a subject has a legitimate right to access information, but must be prevented from passing it to unauthorised third parties.

How might information reach unauthorised third parties? In a conventional system the information of interest is primarily stored in persistent files in the file system. When this is stored in a *conventional file system*, hackers have more than one way of achieving their aims. They can search for weaknesses in the file system, or in conventional (non-persistent) virtual memory, or in the interfaces between the two. By contrast, persistent information in SPEEDOS is always held in the persistent virtual memory in information hiding files with their own semantic routines.

Information held in the *SPEEDOS persistent virtual memory* is made available to subjects in the system primarily via inter-module calls. This gives file owners an assurance that only those users whom the owner has authorised (by providing them with capabilities[135]) can invoke these semantic routines to obtain information directly. But one important guarantee is *not* provided by the capability and module calling mechanisms, viz. that the code of the semantic routines correctly implements its specification (and nothing more or less than the specification)!

This is one of the reasons why a confinement problem exists at all. In

[135] Chapter 26 describes how capabilities and other mechanisms in SPEEDOS can be used to prevent users entitled to access to a file from passing on these rights.

SPEEDOS, a code implementation containing a trojan horse (or an error, or maliciously programmed code) might attempt to release information in several ways. It might, for example

– pass information directly back to a caller who has a capability entitling him to call the module, although not to obtain the information in question;
– secretly pass information from the file to another file module, via which it can be accessed later by a hacker;
– display information on an output device, e.g. a monitor screen or printer.

Similarly, incorrectly implemented code of a file module might violate the integrity and/or availability of information held in a file by accidentally or deliberately changing or destroying it.

These are not the only – nor even the usual – problems which attract attention in discussions of how information can be confined. In conventional systems the issue is not how the implementation routines of a file module can be confined (because the SPEEDOS concept of a file module with semantic routines is not found in such systems). The issue for such systems is rather how programs which illegitimately gain access to data in the file system can be confined, but also how programs which legitimately need access to a file in order to carry out services for applications and their users, such as a spooler module in the operating system, can be confined so that they do not misuse the information which users provide to them.

The first of these problems should not arise in SPEEDOS, because its implementation of the information hiding principle causes a single authorised program to be tightly bound to a file, thus excluding other programs from directly accessing the file data. While there are occasions when the program in question must be replaced by another, this change is carried out via a semantic routine of the appropriate co-module manager. The latter can only be called by a subject in possession of a valid capability for this co-module. Therefore this kind of problem should only arise as a result of a human error or a deliberate criminal act carried out by an insider. However, human errors occur, and unfortunately insiders can be persuaded by greed or bribes or even blackmail to commit criminal acts and to help others to do so. Consequently it must be anticipated that further technical measures must be taken to ensure the confinement of information.

The second problem, that software with legitimate access to information can illegitimately release this to third parties, could just as easily occur in SPEEDOS as in conventional systems if the latter did not provide confinement mechanisms. This is the main subject of the present chapter, although the solutions described can be equally applied to programs which illegitimately gain direct access to persistent data.

The remaining sections of this chapter take a closer look at the information channels which might be used to release information against the will of its owners, and how the tools available in SPEEDOS can help confine programs. Finally some examples are presented.

1 Information Channels

Neither a system architecture nor an operating system can ever guarantee that operating system and/or application code is fully correct. At best, a system architecture can ensure that certain basic actions are prevented while a thread is executing in a module. For example it can enforce basic access rights which

– prevent write operations to particular data segments when "read-only" mode is set, but

– allow writes when "read-write" mode is set.

Because of the SPEEDOS structure, these controls can be applied separately for the different kinds of segments associated with a module, as follows.

1.1 Persistent Data Segments

File modules have persistent data segments, which may be shared by many users with capabilities (via their semantic routines). Access to these segments can be variously set individually to "read-only" or to "read-write" for executing threads, i.e. some threads can write to them, while other threads can only read them[136].

Persistent data segments represent a security problem for two reasons:

– Because they are shareable by many threads (both concurrently and over time) they can serve as an information channel which might be misused.

– Because they contain persistent data, this must be considered to be "useful" data, i.e. data potentially of interest to third parties.

Hence persistent data segments provide a potential *communication channel between user threads* of the same or different processes, and an ideal target for hackers.

1.2 Temporary Data Segments

Thread-local data segments hold three kinds of non-persistent (i.e. temporary) data associated with a particular thread, i.e. its internal computational stack, its local heap data and its inter-module parameter segments. The kernel ensures that the local data segments of an executing thread cannot be shared with other threads, since each thread has a separate root for its temporary segments which

[136] Such accesses should of course be synchronised.

the Segment Manager dynamically allocates on request from the thread. They may therefore be modified without creating the possibility that information can be passed from one thread to another. However, inter-module parameter segments provide a *communication channel between modules*.

1.3 Code Segments

The segments of a module include both local and externally accessible code routines which may for example consist of the routines for accessing a file module, or they may implement a program or library routines. These segments exist in one or more code modules.

A code module starts its life as a persistent data module for a compiler, i.e. as a file which is the output of the compiler. But the kernel, with the help of its security sensitive co-modules, ensures that once a module has been designated as code it cannot be modified while being executed.

A code module may have its own data segments but these are always set to read-only while the module is being executed as code, and therefore they cannot be used as a communication channel. They may, however, hold capabilities which can be used to make inter-module calls.

1.4 Communication Channels Relevant to the Confinement Problem

The above points lead to the conclusion that there are three basic information channels which could be used by malicious code to release information illicitly: persistent data segments, inter-module parameter segments and module capabilities held in constant segments of the code. In this chapter SPEEDOS mechanisms are described which can restrict the passing of information via these segments. We now consider in more detail how these mechanisms can be used to prevent information from being stolen.

2 Bracket Routines

Both call-in and call-out bracket routines[137] can be used as tools for confining modules, especially where the owner of a file module does not trust the code which manages it.

2.1 Call-Out Brackets

An obvious way of preventing information leakage is to use call-out brackets. A call-out bracket is activated when a qualified module attempts to call other modules. It intercepts outgoing calls and executes a prelude. If this determines that the call can proceed, the call-out bracket executes a `call` instruction, which allows the appropriate routine of the destination module to be entered. When this

[137] see chapter 24.

finally returns, the call-out bracket routine continues at the postlude following the `call` instruction, and finally a return is made to the qualified module. If a call statement is not executed in the call-out routine, the call never reaches the intended module.

There are at least two ways in which a call-out bracket method can confine information. It can simply prevent its qualified module from making calls to some or all other modules, or it can examine and where appropriate change the parameters being passed to the destination module, e.g. by reducing the access rights in a module capability or by replacing the module capability with another, less dangerous one. Call-out brackets cannot increase the rights in a capability.

Call-out brackets can only be applied by the owner of a qualified module, or by a subject authorised by the owner to access the qualified module's qualifier list module in an appropriate way. Consequently this technique is most useful for cases where the owner of the module (i.e. the owner of the module's data file) does not trust the software which manages his data and/or that which is used in the destination module. Since the vast majority of computer users either cannot or do not want to write all their own programs, most of the software which they use is written by other (usually unknown) programmers and is therefore potentially a source of data leaks.

2.2 Call-In Brackets

Call-in bracket routines can also be used to help prevent information leakage. But whereas call-out brackets are primarily concerned with protecting information held in the qualified module from being passed on to third parties via inter-module calls, call-in bracket routines are usually concerned with preventing modules which have called the qualified module from receiving information via return parameters. In this case unwanted callers, who have either legitimately or illegitimately obtained a capability to call the qualified module, may attempt to trick the qualified module into returning information in return parameters which can later be used to access and even modify information illegally.

If in its postlude a call-in bracket detects a risk, it can, for example, change the return parameters, signal an error or simply log its suspicions.

3 Information Confinement Rights

As indicated above, there are three basic information channels which could be used by malicious code to release information illicitly: persistent data segments, parameter segments and constant segments in the code holding module capabilities. We now describe how confinement rights can play a role in restricting the passing of information via these information channels.

The first group of rights, the *information confinement* rights, is designed to prevent information flow from a module via these channels. Information in this context includes both module capabilities and other data. These rights are summarised in Figure 25.1.

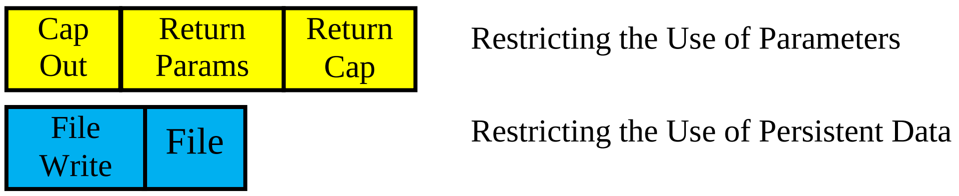

Cap Out
Return Params
Return Cap
Restricting the Use of Parameters
File Write
File
Restricting the Use of Persistent Data

Figure 25.1: Information Confinement Rights

The information confinement rights work either by ensuring that segments or capability partitions therein are inaccessible (Cap Out, Return Params, Return Cap, File), or by setting the access rights of appropriate segments to read-only (File Write), thus preventing the writing of information to them[138]. These rights can be unset in a capability or directly in the Thread Security Register (TSR), for example by a bracket routine.[139]

3.1 Restricting the Use of Parameters

In SPEEDOS the kernel can easily distinguish between input parameters and return parameters on external calls (i.e. inter-module calls and co-module calls) because these are held in different parameter segments.

The basic SPEEDOS call mechanism (see Chapter 19) already ensures that a called module cannot write to its input parameters and that a calling module cannot write to the return parameters which are passed back to it. Furthermore, there is little point in restricting a caller from writing to the output parameter segment while it is being prepared for calling a module, since virtually every routine needs these parameters, which become its input parameters.

It does, however, make sense in some cases to ensure that code does not attempt to pass a capability as an output parameter if it is only supposed to be passing normal data. If the right `permit_cap_out` is unset, the kernel prevents this (Figure 25.2).

It can also make sense to prevent a called module from writing to the parameter segment via which it returns parameters to its caller (when it should not be passing information back). This is achieved by unsetting the confinement right `permit_return_params`. When this confinement right is permitted, a

[138] They follow the same principle as other access right settings, i.e. if the corresponding bit is set, the thread has access, if it is unset, the corresponding action is prevented. A thread can only reduce access (by setting a 1 bit to 0). It can never increase access (by setting a 0 bit to 1).

[139] These possibilities are described in more detail in the next chapter.

called module can return results; when it is unset the return parameter segment is set to read-only, preventing the called module from copying data and capabilities (e.g. from its file data) into the result parameter segment, see Figure 25.3.

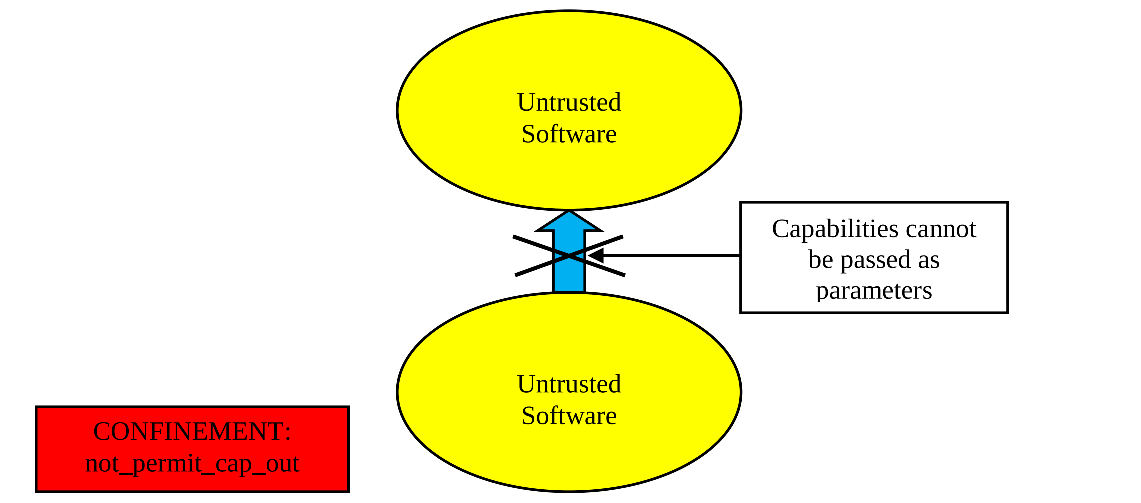

Untrusted Software
Capabilities cannot be passed as parameters
Untrusted Software
CONFINEMENT: not_permit_cap_out

Figure 25.2: The `permit_cap_out` Confinement Right

In the case of a co-module call, the unsetting of `permit_return_params` also prevents a module from returning pointers as parameters. (A similar result can also be achieved if the caller sets the lengths of the return parameter segment in the kernel instructions `create_imc_params` and `create_pc_params` to zero, but this cannot be done in bracket routines, and there is no guarantee that untrusted software will do this.)

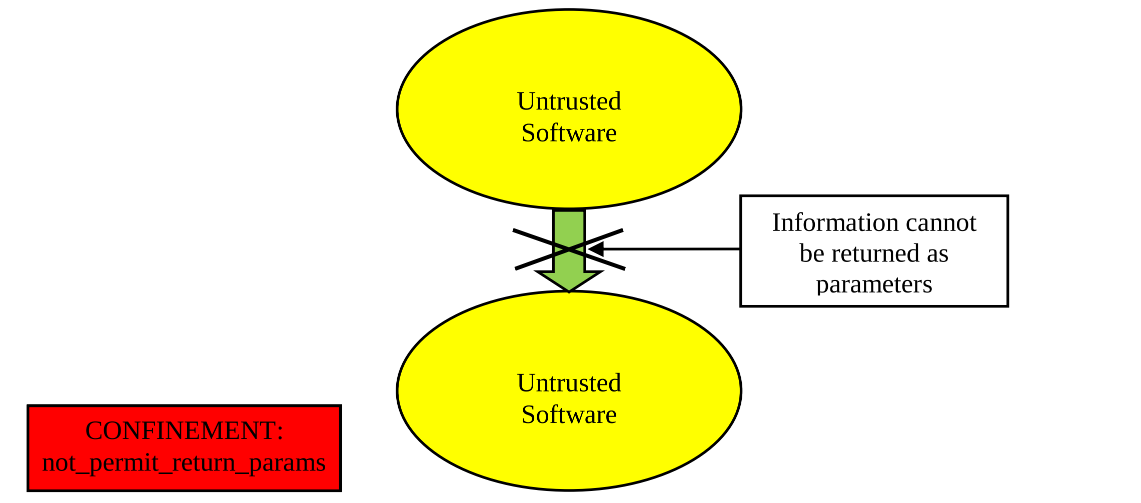

Untrusted Software
Information cannot be returned as parameters
Untrusted Software
CONFINEMENT: not_permit_return_params

Figure 25.3: The `permit_return_params` Confinement Right

The less restrictive right `permit_return_cap` can be used to prevent capabilities, but not data, from being returned (Figure 25.4).

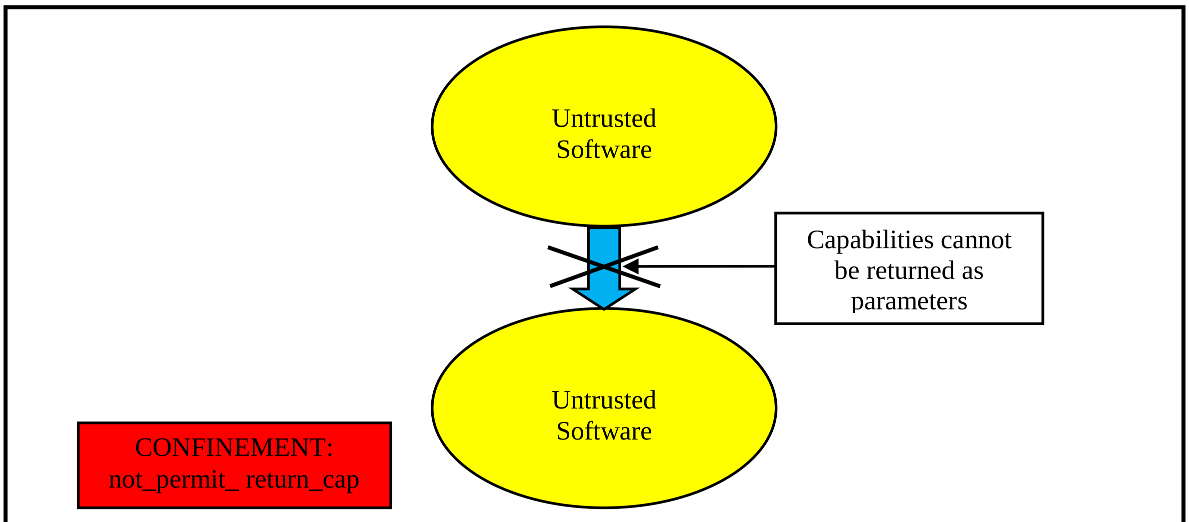

Untrusted Software
Capabilities cannot be returned as parameters
Untrusted Software
CONFINEMENT: not_permit_ return_cap

Figure 25.4: The `permit_return_cap` Confinement Right

3.2 Controlling Access to Persistent Information

The entry point list of a module indicates whether the segments of state data should be set for the thread to read-only or to read-write access for each semantic routine individually. However, the read-write setting can be overridden by unsetting the confinement right `permit_file_write`. When this confinement right is permitted, a called module can read and write the state data of the module, subject to the setting in the entry point list of the called routine. Access to all the state segments is set to read-only when `permit_file_write` is unset, regardless of the setting in the EPL entry for the called routine (Figure 25.5).

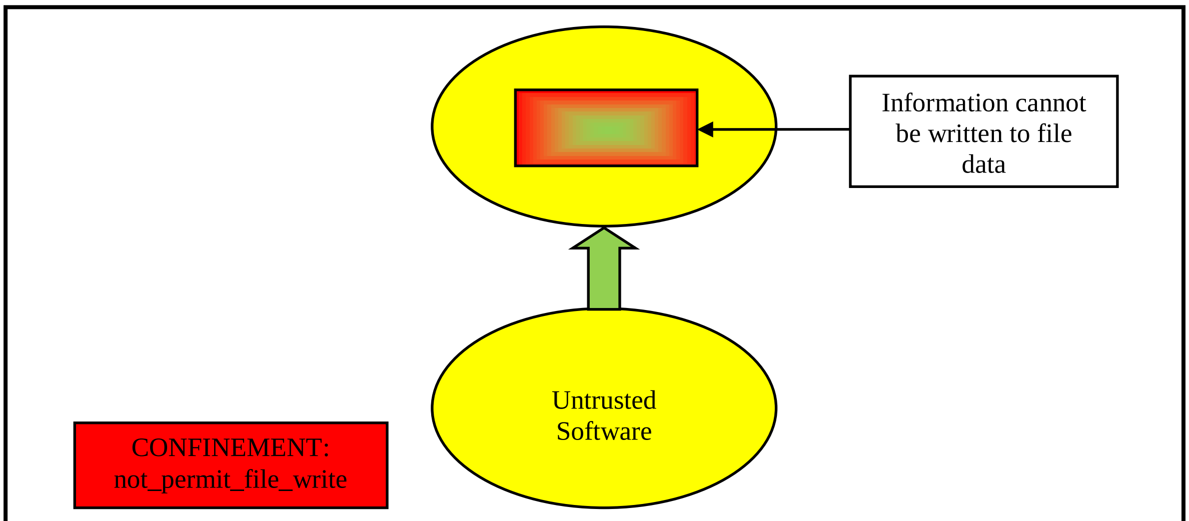

Information cannot be written to file data
Untrusted Software
CONFINEMENT: not_permit_file_write

Figure 25.5: The `permit_file_write` Confinement Right

From the viewpoint of confining information, unsetting this permission has the advantage that the software associated with a called module cannot secretly write information to a file, even if the file exists and contains valid information which can be modified by other callers.

It also has the advantage that it can prevent threats to the integrity of information and its availability.

A more drastic way of controlling access to persistent data is provided by the `permit_file` confinement right, since this, if unset, prevents a module from accessing persistent data segments. If a module containing file data is called, the kernel sets access to persistent file segments as "no access" (i.e. both read and write permissions are unset). Calls to independent program modules (see Chapter 18) are allowed (see Figure 25.6).

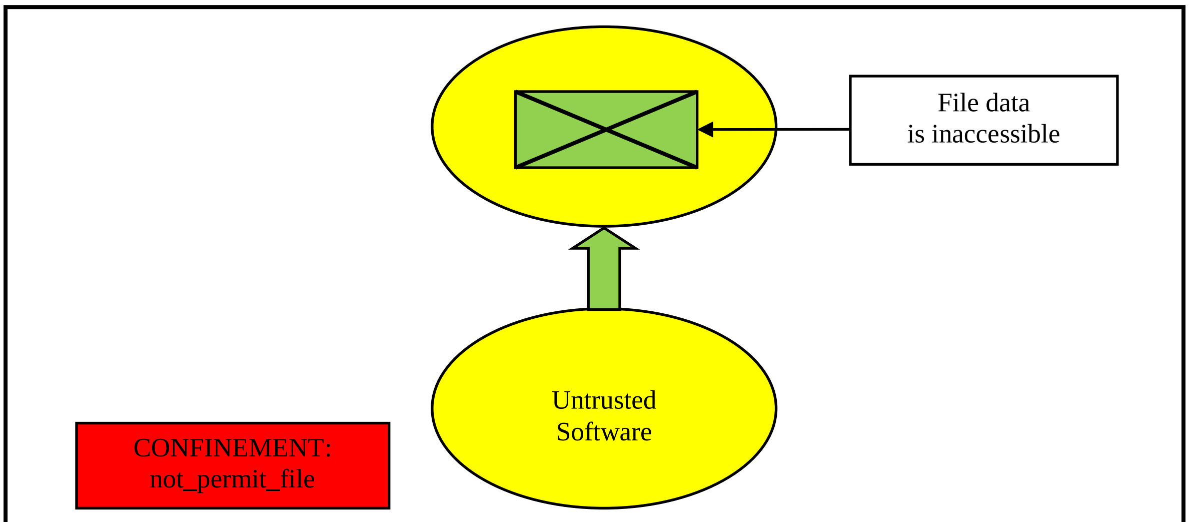


Figure 25.6: The `permit_file` Confinement Right

Both normal application modules with file data and independent program modules use a data container for storing temporary thread-oriented structures such as local stack and heap data. The unsetting of the `permit_file_write` and/or `permit_file` rights does not prevent either from creating such temporary segments, which have no persistent root and which are either deleted or become inaccessible when the thread exits the module. Furthermore no thread can access the temporary segments of another thread. Hence they cannot secretly store information which will survive after an inter-module return.

4 Module Call Confinement Rights

Call-out brackets provide the simplest way of restricting calls from a qualified module, and they have the advantage that the decision to make such a restriction can be finely controlled, e.g. on the basis of an examination of the parameters being passed or the identity of the caller. However, they have the disadvantage that they can only be used when the caller wishing to make the restriction is also the owner of the module in question or has a capability for the qualifier list module. Hence a second group of rights, the *module call confinement* rights, is aimed at preventing an executing module from making particular classes of call. These rights are summarised in Figure 25.7.

Calls	Const Calls	Param Calls	Nonparam calls	Comod calls	Sync Calls	Restricting Calls

Figure 25.7: Module Call Confinement Rights

These restrictions cannot be so finely tuned as controlling calls via bracket routines, but they have the advantage that once applied to a module, they can be automatically applied in appropriate circumstances to modules called subsequently, as will be described in the next chapter.

The call restrictions fall into six categories:

– all calls (`permit_calls`). If this permission is unset, the called module can make no further calls of any kind, regardless of the other call permissions;

– calls (`permit_const_calls`) made using module capabilities located in or previously copied from code constant segments;

– calls to modules via a module capability which has been explicitly passed to the module via an input parameter segment (`permit_param_calls`);

– calls to modules for which no module capability has been explicitly passed and which are not "constant" calls (`permit_nonparam_calls`);

– calls to user co-modules for which no module capability has been explicitly passed (`permit_comodule_calls`);

– calls to synchronising modules (which is explained below).

These are summarised in Figure 25.7 and illustrated in Figures 25.8 to 25.13. In some cases the restrictions refer to the source (segment) of the capability used to make the call. Since user code can attempt to disguise such origins by moving the capability in question from one kind of segment to another kind, the status bits in the capability are used to indicate the origin in the case that a capability is moved. This is more fully described in chapter 26.

4.1 The Permit Calls Right

Unsetting `permit_calls` has the very drastic effect. A module which has been subjected to this restriction cannot make any calls whatsoever.

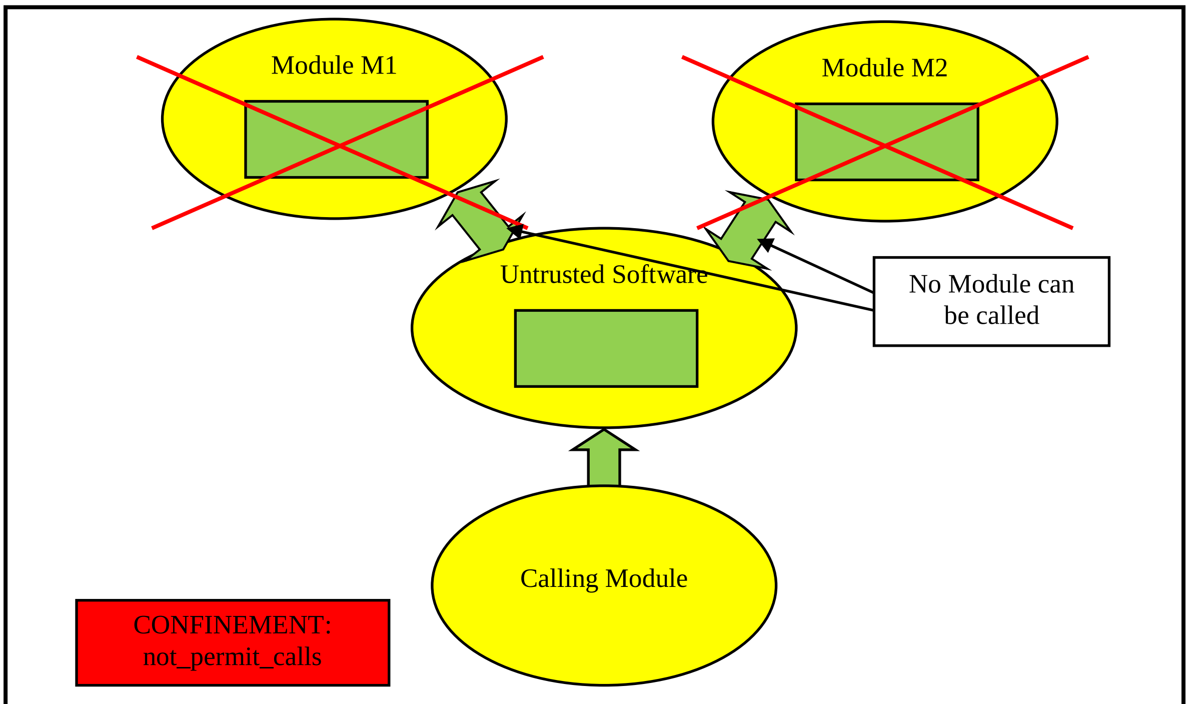


Figure 25.8: The `permit_calls` Confinement Right

4.2 The Permit Constant Calls Right

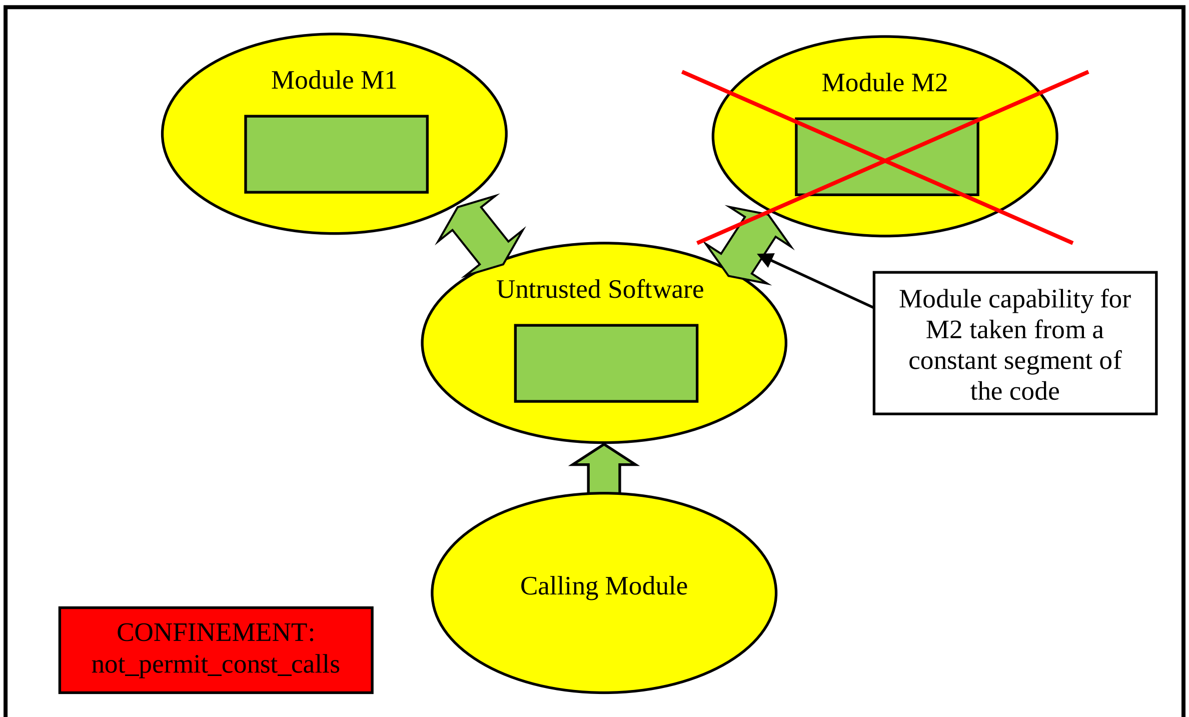

Module M1
Module M2
Untrusted Software
Module capability for M2 taken from a constant segment of the code
Calling Module
CONFINEMENT: not_permit_const_calls

Figure 25.9: The `permit_const_calls` Confinement Right

In the case of restricting `permit_const_calls`, attempts to call modules using capabilities embedded within the code segments of a module are forbidden. This can be used to prevent a capability embedded in a code module (e.g. for a file module at the home base of a software design company) from being used to make a call.

4.3 The Permit Non-Parameter Calls Right

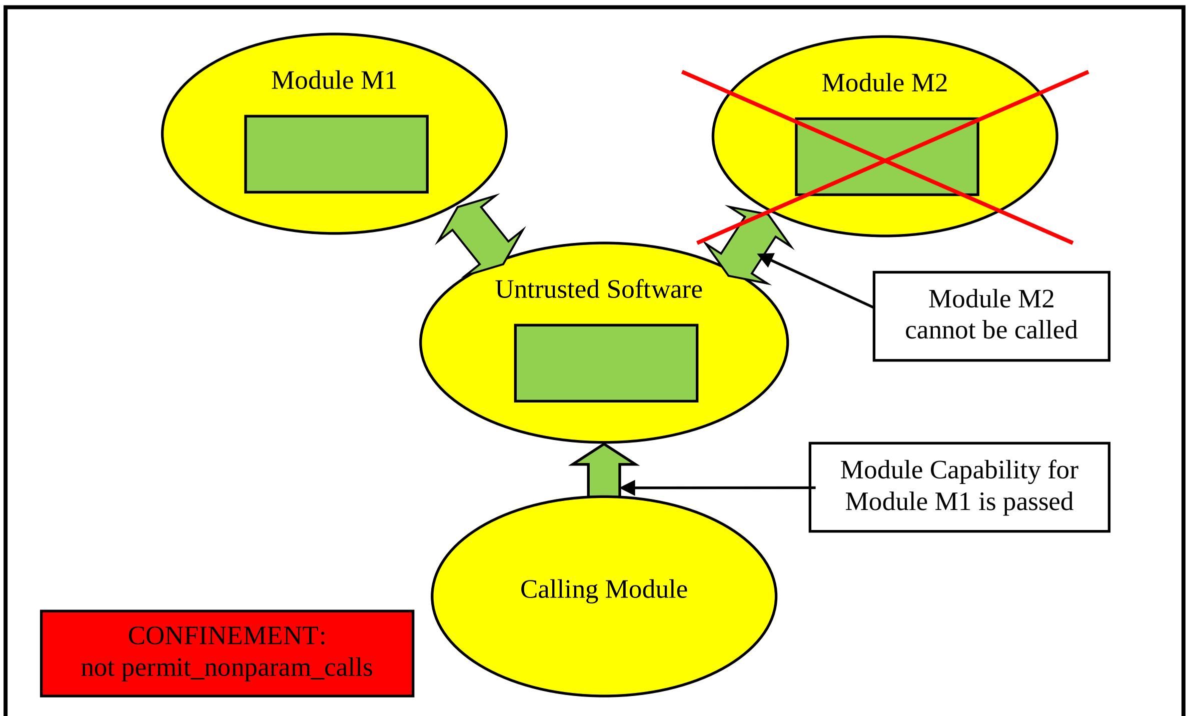

Module M1
Module M2
Untrusted Software
Module M2 cannot be called
Module Capability for Module M1 is passed
Calling Module
CONFINEMENT: not permit_nonparam_calls

Figure 25.10: The `permit_nonparam_calls` Confinement Right

Unsetting the `permit_nonparam_calls` right has the advantage that a module capability hidden in the module being called (e.g. in proprietary software which has its own persistent data) cannot be used to make further calls (e.g. back to the database of the proprietary company), while the thread can still call the caller's own modules passed as parameters.

4.4 The Permit Parameter Calls Right

The unsetting of the `permit_param_calls` right prevents a module from using capabilities passed to it as parameters from using these capabilities to make calls (Figure 25.11). This might for example be used if a module A calls a further module B which then calls a further module C. Since the owner of A may not be aware of the call to C and therefore cannot know whether B is releasing his information to C in parameter form, he might use this confinement to ensure that his information is not released.

It might be thought that in cases where `permit_nonparam_calls` is set the `permit_param_calls` will also always be set, on the assumption that it makes no sense to allow calls to unknown modules while not allowing calls to modules deliberately passed as parameters. However there are at least two reasons why this might not be appropriate. First, the module capability passed as a parameter may be intended for use by the module at some later time and possibly while it is executing in a different thread. Second, if the caller does not need to pass capabilities as parameters then `permit_param_calls` can be unset as a precaution against a bracket routine which might surreptitiously add a module capability as an input parameter.

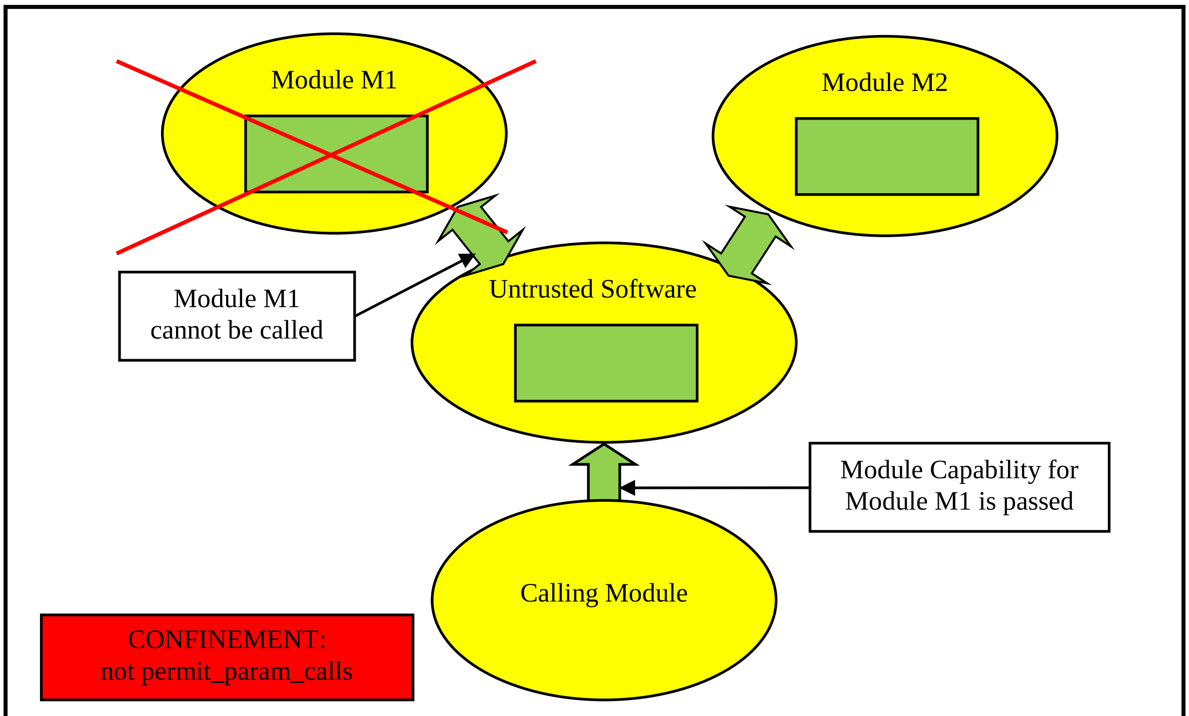

Module M1
Module M2
Module M1 cannot be called
Untrusted Software
Module Capability for Module M1 is passed
Calling Module
CONFINEMENT: not permit_param_calls

Figure 25.11: The `permit_param_calls` Confinement Right

4.5 The Permit Co-Module Calls Right

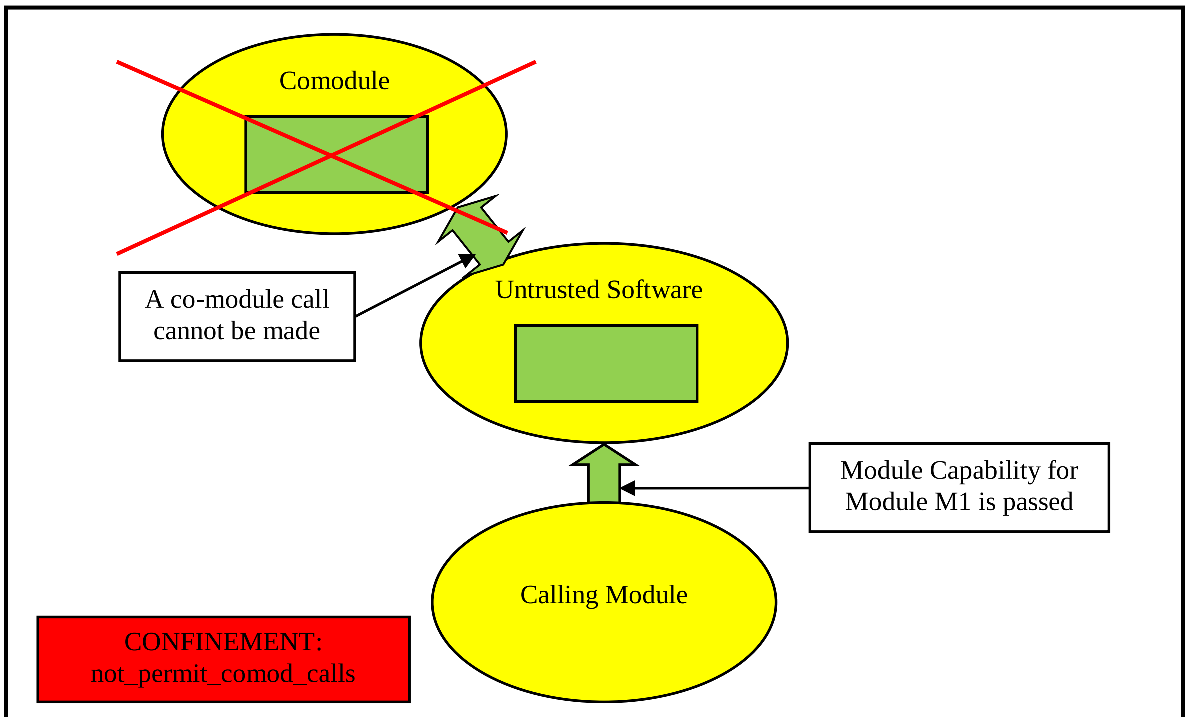

Comodule
A co-module call cannot be made
Untrusted Software
Module Capability for Module M1 is passed
Calling Module
CONFINEMENT: not_permit_comod_calls

Figure 25.12: The `permit_comod_calls` Confinement Right

If `permit_comod_calls` (see Figure 25.12) is unset, a called module cannot make co-module calls using the CMC instruction, i.e. to another module in the same container (but this does not prevent it from making normal inter-module calls without pointer parameters) to such a module, provided that it has a capability and there are no other restrictions which would prevent such a call. Without unsetting this permission it might be possible for modules to pass pointers for a database to co-conspiring modules, which provides the latter with direct access to the database.

4.6 The Permit Synchronisation Calls Right

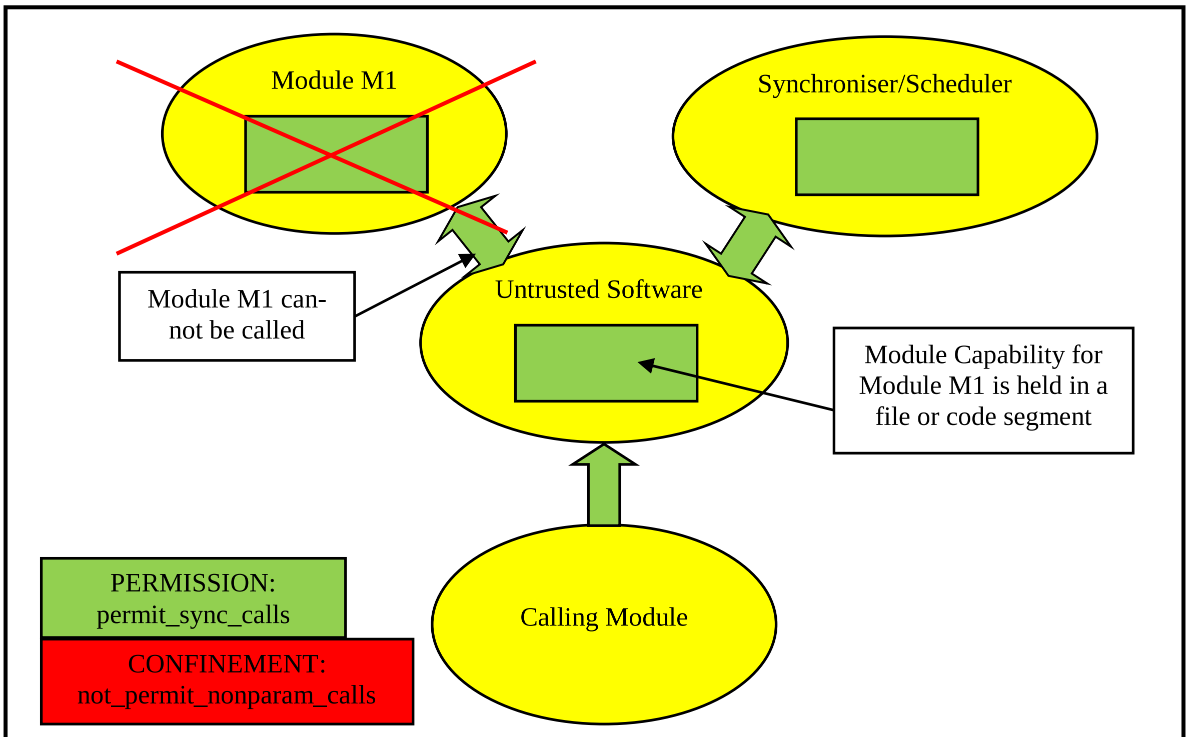

Module M1
Synchroniser/Scheduler
Module M1 cannot be called
Untrusted Software
Module Capability for Module M1 is held in a file or code segment
PERMISSION: permit_sync_calls
CONFINEMENT: not_permit_nonparam_calls
Calling Module

Figure 25.13: The `permit_sync_calls` Confinement Right

The right `permit_sync_calls` (see Figure 25.13) has been introduced to simplify the use of `permit_nonparam_calls`. Leaving the latter permission set (as a right) increases the possibility that arbitrary capabilities which have not been passed as parameters to a called module (which might therefore have been illegally obtained by the called module) may be used to gain access to a file. This risk cannot be avoided in some cases, e.g. when a called module uses capabilities which were legitimately passed to the module as parameters to a constructor call. However there is one common situation where this risk can be avoided. This arises from the decision described in Chapter 21 that synchronising operations which involve suspending and re-activating threads must do this via routines of the appropriate Thread Control Manager. In order to do this, threads must not only suspend themselves but must also activate other threads, possibly belonging to different users. Without adding a confinement permission `permit_sync_calls` it would quite frequently be necessary to leave `permit_nonparam_calls` unset when a thread needs to invoke methods of the scheduler/synchroniser modules, since the thread capability needed to re-activate a thread is not acquired via parameters received from a call to the scheduler, but typically from a queue of suspended threads. If `permit_sync_calls` is set, the kernel permits calls to a thread scheduler, even when `permit_nonparam_calls` is unset. (The kernel can distinguish calls to a thread scheduler, since these are invoked by presenting a thread capability, rather than a normal capability.)

4.7 Note on Library Calls

Library calls are in effect simply extensions of the main code file from which they are called. Hence they are excepted from the restrictions which would result on them being called. However if they themselves attempt to make further calls to other modules, any call restrictions which have been imposed on the main code file apply to these calls.

5 Conclusion

This chapter has described some basic principles for solving the confinement problem in SPEEDOS and has hinted at some solutions in cases where bracket routines cannot always be used. However, it has not concretised how the proposed rights can be implemented in detail. In the following chapter further rights are described and more attention is given to implementation details.

Chapter 26
Some Confinement and Access Controls

The previous chapter looked at some basic principles and techniques for solving the confinement problem. This chapter continues and expands the basic theme by reiterating, bringing together and expanding upon those protection mechanisms which have already been discussed in earlier chapters. We begin by explaining further mechanisms which can contribute to a safer environment, viz. the rights which under certain circumstances

- allow a thread to acquire information about the environment in which it is working, thus allowing appropriate software to carry out protection and other checks, and
- prevent modules from gaining access to certain capabilities.

We then provide details of the access rights in capabilities and introduce a new mechanism, the Thread Security Register (TSR), which is an essential part of the state of each user thread and is held at the base of the thread's kernel thread stack.

The rights themselves are held in three locations. Those held in the Thread Security Register allow the owner of a thread to set rights which apply to the thread while it is executing. Those held in capabilities allow the owner of the object addressed by the capability to determine how the capability and the object which it addresses can be used. The container rights are held in page 0 of each container.

The basic principle in all the SPEEDOS protection mechanisms is that rights can always be reduced but never increased by a thread. The kernel ensures this by using an intersection instruction to reduce rights.

1 Environmental Checks

In order to carry out security checks, software (e.g. bracket routines, but also normal modules) often needs information about the environment in which it is working. For example when checking whether a capability has been revoked for a user, it is necessary to know who the user is, and whether he owns the thread currently trying to use the capability. To satisfy this and many similar requirements, there are kernel instructions which provide such information.

1.1 Checking Application Modules

These kernel instructions are listed in three groups by Espenlaub [4, pp. 235-236][140], and a fourth group is added below.

i) The first group of kernel calls returns information directly related to the current environment of a thread, in the form of world-wide unique module or thread identifiers[141] (including the index value):

```
unique_id current_thread();
unique_id current_file();
unique_id current_code();
unique_id calling_file();
unique_id calling_code();
unique_id target_file();
unique_id target_code();
```

If these instructions are executed in an invalid situation, they return the value 0. When they are executed by an application module, their meaning is straightforward:

- `current_thread` returns the unique identifier of the thread in which the instruction is executed;
- `current_file` and `current_code` return the unique identifiers of the file module and code module of the currently active application module;
- `calling_file` and `calling_code` return the unique identifiers of the file module and code module of the module which called the currently active application module;
- `target_file` and `target_code` refer to the unique identifiers of the file module and code module about to be called by the currently active module. In practice these are only known when the currently active module has already issued a call instruction; hence if these instructions are used by nor-

[140] Most of these instructions are based on my lecture slides on "Secure System Architecture" at the University of Ulm, but Espenlaub added some additional instructions for use in call-out brackets, i.e. those referring to the target module (the destination module of an inter-module call). These are only relevant for use in call-out brackets, and return a value of 0 if used in any other context.

[141] A module/thread identifier is its full 192 bit identifier, i.e. Node #, Disc # and Container # (including index). It is not a capability.

mal application code, the result returned is always in practice 0.

The two instructions `target_file` and `target_code` encode into the return value (`unique_id`) a bit indicating whether the module being called was passed to its calling module as a parameter, thus enabling the caller to check restrictions on the use of parameters (see, chapter 25 section 3.1).

If these instructions are executed in bracket routines

- `current_file` and `current_code` return the value 0;
- `calling_file` and `calling_code` refer to the module which called the module now making a call;
- `target_file` and `target_code` refer to the module currently being called.

ii) The second group of kernel instructions identifies the owner of each of the above, in the form of a unique container number (showing an index value of -2)[142]:

```
container_id current_thread_owner()
container_id current_file_owner()
container_id current_code_owner();
container_id calling_file_owner();
container_id calling_code_owner();
container_id target_file_owner();
container_id target_code_owner();
```

The kernel obtains this information from the red tape at the beginning of the corresponding container (see Figure 19.2). For example `current_file_owner` is the owner of the container in which the currently active data file is located.

iii) The third instruction group returns the number of the semantic routine (entry point) of the currently active module, of the semantic routine which called this, and of the semantic routine currently being invoked.

```
int current_ep()
int calling_ep()
int target_ep()
```

iv) A fourth group of environmental instructions, known as the *calling rights*, is needed in order that bracket routines can more thoroughly check the access rights associated with the *target call* than was envisaged by Espenlaub.

```
bitlist semantic_rights()
bitlist metarights()
bitlist capability_rights();
bitlist environmental_rights();
bitlist confinement_rights();
```

[142] The index field, which normally signifies the module number within a container, is set to -1 when an entire container is being identified. The number of the first container for a new user identifies the user uniquely throughout his existence in the system and has the index value -2.

```
bitlist thread_rights();
```

The semantic rights, metarights and capability rights are obtained by the kernel from the calling capability, the remaining rights from the Thread Security Register, which is described in section 4 below.

1.2 Checking Bracket Routines

In principle bracket routines should be checkable in the same way as application modules. However, it would not only be considerably more difficult actually to check these but also to specify kernel instructions for this purpose. However, an alternative technique is possible. As was described earlier, the bracket routines which qualify a module are held in a list, which is itself a module known as a qualifier list module (QLM). Information provided by this module is accessible to the kernel when the qualified module is executed or is being called.

The kernel can make the following information about this list module available at run-time to applications and bracket routines:

```
unique_id calling_QLM_file()
unique_id calling_QLM_code()
unique_id target_QLM_file()
unique_id target_QLM_code()
```

where `QLM` is an abbreviation for a qualifier list module and refers to the appropriate qualifier list module (see Figure 19.6, Figure 19.8 and chapter 24).

These instructions allow an application module or bracket routine to log this information for further security checks, or – if it has a capability for the appropriate module – to call its semantic routines to obtain further information.

There are corresponding ownership checks:

```
container_id calling_QLM_file_owner()
container_id calling_QLM_code_owner()
container_id target_QLM_file_owner()
container_id target_QLM_code_owner()
```

These can be useful in cases where certain users are not known or are considered to be completely untrustworthy.

1.3 Rights for Environmental Checking

The environmental instructions return sensitive information to callers and hence their use must be controlled. Normally the right to use kernel instructions is controlled via kernel capabilities, but this method is not sufficiently dynamic for the present purpose, so that another technique is used.

Two sets of rights are maintained (see Figure 26.1). The first lists the rights which an application module can exercise; the second lists the rights which bracket routines can exercise. Although both sets of rights might be set and unset in the same way, they can differ, because the security aims of bracket routines can differ substantially from the aims of normal applications.

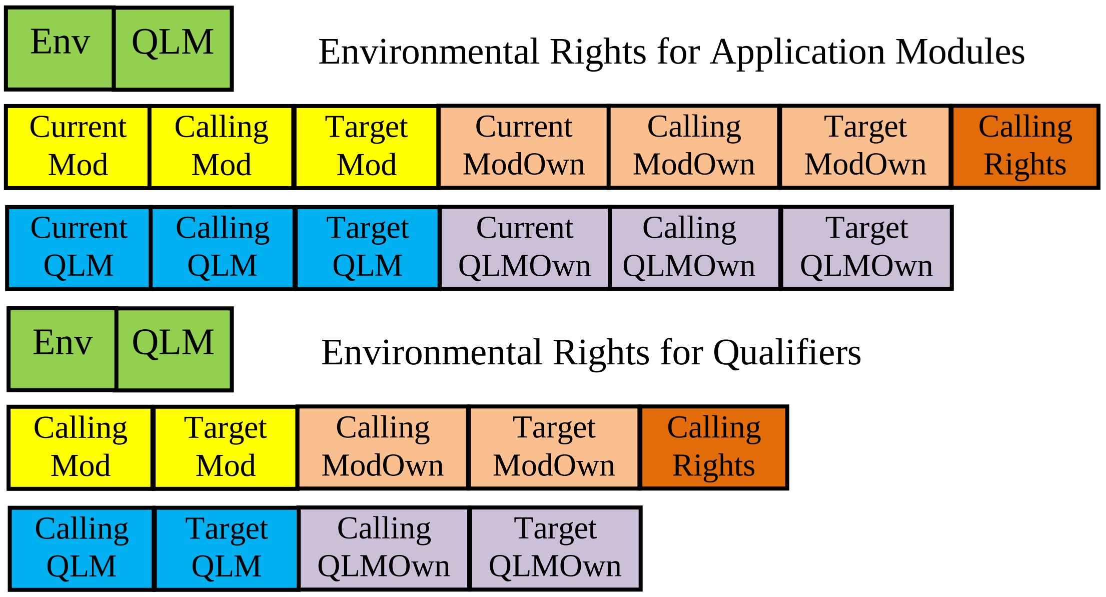


Figure 26.1: Environmental Rights

The environmental rights are summarised in Figure 26.1:

```
permit_env_checks
permit_QLM_checks
```

These are general rights which together allow all environmental checks to be turned off together (for application modules and/or for bracket routines). If these rights are permitted, individual groups of rights can be turned off by unsetting the following permissions.

```
permit_current_module_checks
permit_calling_module_checks
permit_target_module_checks
permit_current_owner_checks
permit_calling_owner_checks
permit_target_owner_checks
permit_calling_rights
permit_calling_QLM_checks
permit_target_QLM_checks
permit_calling_QLM_owner_checks
permit_target_QLM_owner_checks
```

These rights can appear in capabilities and also in the Thread Security Register (see sections 3 and 4). If the corresponding right is unset in either or both, the action is prohibited.

2 Capability Accessibility and Use Rights

Section 5 of chapter 19 explained how certain capabilities can be made accessible to the threads which need them. However, not every thread needs, nor should have the right to obtain, these capabilities. Known as the *capability accessibility* *rights*, these are concerned with controlling their accessibility and use (see Figure 26.2). The sixth accessibility right can control access to free capabilities (which were introduced in chapter 18 section 8), while the seventh and eighth rights define levels of privileges possessed by the holder of a capability.

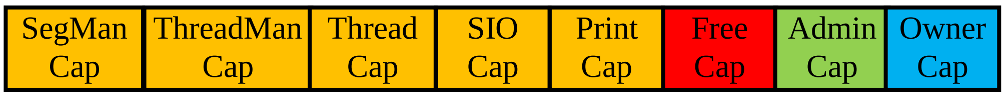


Figure 26.2: Capability Accessibility and Use Rights: An Overview

The first five rights (`permit_segman_cap`, `permit_threadman_cap`, `permit_thread_cap`, `permit_SIO_cap`, `permit_print_cap`) determine whether the potentially restricted module is permitted to obtain

– a module capability for the Segment Manager associated with the current module, allowing the thread to create segments explicitly,

– a module capability for the Thread Manager associated with the current thread (in order to create subthreads),

– a thread capability for the currently executing thread, to allow it to synchronise with other threads,

– module capabilities for the standard input and output modules associated with the current thread, and

– a module capability for the current thread's print request module (see chapter 33).

Unsetting the `permit_free_cap` permission is an important precaution which can be applied in most situations, since providing a module with direct access to the root persistent data segment of another module is not only a violation of the information-hiding principle but if misused it provides a hacker with unlimited access to all the information in the module. However, it is not intended that free capabilities will be widely used as a normal way of accessing modules; rather it is intended that they will be used only in special situations such as the conversion of files or to enable them to be efficiently copied or compared. Hence by default this right should normally be turned off by users. However, it cannot be unset as a system default, since it could then never be turned on (or a mechanism would have to be devised to allow it as a special case).

An administrator capability confers certain administrative rights on the holder of a capability in which the administrator right set. These are defined at the operating system level. There is only one owner capability for a container, file or process which is set when the relevant object is created.

3 Rights in Capabilities

Module capabilities hold six groups of rights:

– semantic access rights (the right to access the interface routines of the module to which it refers),

– generic rights (rights which are needed for controlling actions common to all modules),

– metarights (the rights which determine how the capability can be used),

– environmental rights (see section 1),

– the various confinement rights (see the previous chapter), and

– the capability accessibility rights (see section 2).

Some of these rights have a direct effect on the use of the capability, whereas others have an effect on the actions which can be taken by modules which are invoked via the capability.

3.1 Semantic Rights

These rights indicate on an individual basis which entry points to a module can be called using the capability. In addition they include two bits which allow the list to be overridden by the following special bit settings:

00 = none, i.e. no semantic routine can be called;

01 = all, i.e. all the semantic rights can be called;

10 = read only, i.e. only enquiries[143] can be called;

11 = use the list of semantic rights.

The first three of these are useful shortcuts for users.

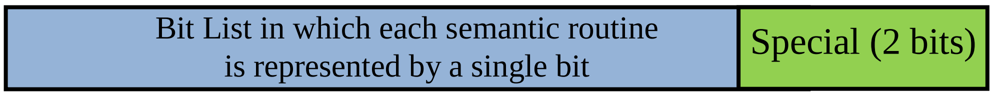

Bit List in which each semantic routine is represented by a single bit
Special (2 bits)

Figure 26.3: Semantic Rights in Capabilities

Bracket routines are not considered to be semantic routines and can never be invoked directly. However, if an executing bracket routine has a capability to call a module the above rules apply as normal.

3.2 Generic Rights

Espenlaub has argued that in SPEEDOS, capabilities should not hold generic

143 'Enquiries' is the name used in Timor to signify routines which do not modify the state data of a module.

rights (see Chapter 2), explaining this as follows:

> "Operations such as creating, copying, renaming and deleting modules are the task of other modules that relate to the implementation of the virtual memory, and may thus be controlled by the semantic access rights to these management modules. However renaming and deleting a module implicitly makes the module capabilities associated with the original module unusable, as the container associated with the module will no longer exist." [4, p. 179]

While it is correct that the operations listed below are implemented in the Container Manager co-module, there is not a separate Container Manager co-module for each container. Consequently access to its routines is via capabilities which are unspecific with respect to the module on which the action (e.g. copying) should be carried out. Hence a user requiring a generic service can only achieve this by passing a capability as a parameter to the routine. Consequently the Container Manager can only determine whether the requested action is permitted by examining an access right in the capability, i.e. a generic access right.

Figure 26.4 shows which generic access rights are supported by the Container Manager:

Copy	Copy with owner change	Delete	Download	Upload	Rename	Change Owner

Figure 26.4: Generic Access Rights in Capabilities

Their meanings are as follows.

`copy`: If set, the Container Manager's copy operation can be invoked to create a copy of the container indicated in the capability, which must be a container capability. Process containers cannot be copied. The copy operation will be carried out as described in chapter 23 section 7. The owner of the copy becomes the owner of the container to be copied.

`copy with owner change`: same as for copy, except that the instigator of the copy becomes the owner of the copy.

`delete`: If set, the Container Manager's delete operation can be invoked to delete the file or the entire container indicated in the capability, taking care to warn the caller of any problems deletion would entail. One result of a delete operation is that all capabilities for the object will be implicitly revoked.

`download/upload`: If set, these operations of the Container Manager can be used to copy the nominated container, which will be transferred to a nominated computer (see chapter 29). The operations will only be carried out if all other permissions allow this (e.g. see the rights in section 5.1). In both cases the uploaded or downloaded file becomes the property of the recipient.

`rename/change_owner`: The meaning of these rights is self-evident.

3.3 Metarights and the 'Copy Cap' Kernel instruction

Metarights[144] define the access control rules which determine how a *capability* (*not* the associated container or module) can be accessed and changed. They affect the execution of kernel instructions for which module capabilities are used as operands, in particular the inter-module call and related calls, the `load_free_cap` instruction and the `copy_cap` instruction.

3.3.1 The 'Copy Cap' Instruction

The latter is defined as follows:

```
copy_cap(boolean data_copy, //copy to data or cap partition
   <segreg#, offset> source_cap,
   <segreg#, offset> destination_cap,
   <boolean> restrict,
   bitlist generic_rights, bitlist metarights,
   bitlist confinement, bitlist environment,
   bitlist semantic_rights)
```

The various access rights bit lists define for the kernel how the corresponding access rights in the destination capability are to be *reduced*. The mechanism is an intersection operation.

In the initial capability for a new object all the rights are set, i.e. all operations are permitted. Normally a capability is never changed, but the `copy_cap` instruction permits copies to be made with (or without) reduced access rights. A capability copy operation has its source address in the capability partition of a segment. Its destination address may be in either the capability partition or the data partition of a segment. Where it is in the data partition, all the `bitlist` parameters are ignored (i.e. the access rights in the capability are copied but not reduced); the copied capability can be read (and modified) as *data*, but it cannot be used as (nor converted back into) a capability.

Note that these operations can only be carried out when the source and destination segments are currently addressable at the same node, i.e. within the same module (including input and output parameter segments) or within another co-module in the same container. Whether they can be passed or returned as parameters to/from other modules depends on the following metarights.

The boolean parameter `restrict` does not apply directly to the copy operation as such, except that it causes the kernel to unset the first capability restriction *status bit* (see section 3.4). This affects further copy operations in that it prevents the holder of this capability from copying it *to a third party after it has*

144 Some of these rights are based on the list of capability confinements which appeared in lecture 12 of my lectures on Secure System Architecture at the University of Ulm, Germany. These were further developed in [4, pp. 178-9], but the final list provided here has been substantially revised and improved.

been copied once to another user, regardless of the settings the metarights `file_copy`, `in_param_copy` and `out_param_copy` in the *permissions for foreign owner* and *for foreign node owner* in Figure 25.6. An implementation is described in section 3.4.2.

3.3.2 The General and Once Only Permissions for Using a Capability

The following metaright permissions define how the capability which contains them can be used. They are checked on inter-module (and similar) calls and returns, by the `load_free_cap` instruction and by the `copy_cap` instruction, as appropriate.

Two possibilities are provided for each individual permission. In the first (general) case, the normal uses for the capability are defined. The same permissions are repeated as "once only" permissions. If a once only permission is set (regardless of the setting in the corresponding general permission) the kernel allows the action to be carried out once only. Both the general permission and the corresponding once only permission are then unset in the capability by the kernel, i.e. the once only permissions override the general permissions. These permissions are listed in Figure 26.5.

Their meanings are as follows.

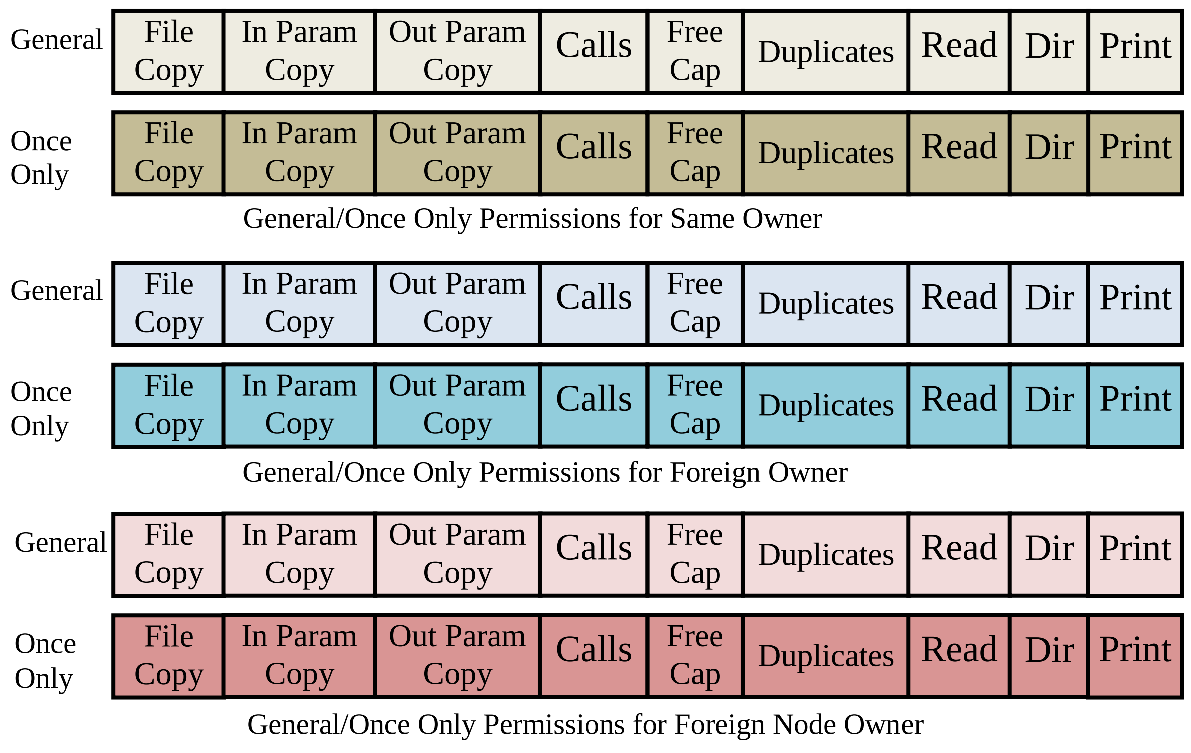


Figure 26.5: Metarights in Capabilities

`permit_file_copy`: If set, the module capability may be copied to a file segment, subject to the rights and restrictions defined below. If unset the capability cannot be used as the source capability for a `copy_cap` instruction where the destination segment is a file segment.

`permit_in_param_copy`: If set, the capability can be passed as a normal input parameter to another module (i.e. it may be copied to the segment addressed via SR2). If unset, it may not be copied to the parameter segment addressed by SR2.

`permit_out_param_copy`: If set, the capability can be passed as a normal return parameter to another module (i.e. it may be copied to the segment addressed via SR1). If unset, it may not be copied to a destination segment addressed by SR1.

`permit_calls`: If unset the capability cannot be used to make inter-module or similar calls. This might be appropriate, for example if the capability is a free capability.

`permit_free_cap`: If set, the capability can be used to access the content of the module which it names directly, i.e. providing access as a free capability parameter which can be loaded into a segment register. This is not possible if the right is unset.

`permit_duplicates`: If unset, the source module capability is invalidated when being copied, changing the behaviour of the `copy_cap` instruction from normal copy to destructive move.

`permit_read`: If set, the capability may be copied to the normal data part of any other segment. This allows the content of a module capability to be examined. If unset, it is not possible to store the content of a capability in the data partition of a segment, thus rendering it impossible to examine its contents.

`permit_dir`: is discussed below under "Directory Mode".

`permit_print`: If set, the file addressed by the capability may be printed. Notice that this right could in theory be classified as a generic right, unlike other generic rights the printing of files is not carried out by the Container Manager but directly by users (see chapter 31) and hence this right has been included with the general and once only metarights.

In a capability the above rights appear in three groups.

a) The first set of rights applies when the owner of the source segment and the owner of the destination segment are the same or when the owner of the current thread and the owner of the capability are the same. Allowing restrictions even when the same owner is involved allows a user to guard against his own potential mistakes, but also helps prevent software and hackers from misusing the capability if they gain access to it.

b) Subject to the rights in (a) not being infringed[145], this set of rights ("foreign owner") applies when the owner of the destination segment and the current thread's owner are different or when the owner of the capability and the current thread's owner differ.

c) Subject to the rights in (a) and in (b) not being infringed, this set of rights ("foreign node") applies when the home node of the owner of the capability is not the same as the node on which the action is being attempted.

3.3.3 Directory Mode

Directories (known as folders in some systems) are used in SPEEDOS analogously to their use in some conventional systems (i.e. to store access rights associated with the use of files). But since capabilities are separately protected by the SPEEDOS kernel, directories need not be special modules. (Directories are discussed in more detail in chapter 30.)

In the present context, they present a particular danger. It should be possible to store a capability in a directory (i.e. in *any* SPEEDOS module) with the intention of ensuring that while it is in the directory it cannot be secretly used by the directory software to invoke its associated module, nor to be passed on as a parameter to a further module, nor to be used in any other way except as a storage repository. Once it is taken from the directory, these restrictions should be removed.

The "once only" modes do not help in this case, neither does just unsetting `permit_calls` nor the other permissions. Similarly the confinement rights (see below) do not help, since they apply only to a particular thread, but a capability may be placed in a directory by one thread with the intention that threads of other users may use it. Hence a special `permit_dir` metaright is provided. When this metaright is set, the capability is treated as normal, but when it is unset the capability, once in directory mode, cannot be passed to another module as its input parameter (i.e. via SR2 for the caller) nor as the operand for an inter-module call.

To change a normal capability to directory mode the kernel first copies it (as a normal capability) into an input parameter segment (SR2 for the caller) of the directory module (later accessible via SR0 for the directory module). Then the user thread unsets the `permit_dir` metaright. The kernel checks that when this unset operation is requested, the capability is in an input parameter segment and then unsets the `permit_dir` metaright. (It cannot – and need not – check that the module being called is a "directory" module, since the kernel does not recognise such modules as special.)

145 It makes no sense to apply more stringent rights to oneself than to others.

When the receiving module copies the capability from the input segment, the kernel checks that the destination segment is a persistent file segment. If not, the copy operation fails. The kernel also allows the capability to be transferred from one file segment to another (to allow for directory re-organisation), but it may not be moved to an input parameter segment in preparation for an inter-module call. Nor can it be used for any other purpose while in directory mode, except to be transferred to a result parameter segment in preparation for an inter-module return. As part of the inter-module return itself, the kernel resets directory mode to normal mode, i.e. the `permit_dir` metaright is set again.

3.4 Status Bits

There are two pairs of status bits which can be set in a capability. These appear as subfields of the container number, not in the access rights fields (see Appendix 1).

3.4.1 The Capability Origin Status Bits

The module call confinement rights were described in chapter 25 section 4. In order for the kernel to determine whether a call is permitted, it must know the origin of the capability, e.g. if it is currently in or was moved from a constant segment of the code, or whether it was passed as a parameter to a module. Since the user might attempt to disguise this origin, the `copy_cap` instruction uses two status bits (the *origin* bits) in a module capability to record the source of its movements. If the capability has been copied from a parameter segment the first bit is unset (0); if the capability has been copied from a constant segment the second bit is unset.

3.4.2 The Capability Copy Restriction Status Bits

These two bits are used to prevent a user who has been given a copy of a capability from further distributing copies of this to other users (see section 3.3.1 above). To implement this two status bits in the capability are used. The first is unset by the kernel in the copy operation initiating the restriction. The second is unset by the kernel immediately following it being passed (or returned) as a parameter to a module not owned by the current user. This allows the original user to provide another user with a copy of the capability (which can be used by the latter as defined in the metarights) but it prevents this user from distributing it (or further copies which this user creates) to third parties.

3.5 Confinement Rights and Environmental Rights

The confinement rights were described in detail in chapter 25. These include:

- the information confinement rights (see Figure 25.1), and
- the module call confinement rights (see Figure 25.7).

The environmental rights were defined in section 1 above.

When confinement rights and/or environmental rights are unset in a capability, the corresponding restrictions are applied to user threads which are executing in a module (or bracket routine) which was activated as a result of a call which used the capability.

3.6 The Capability Accessibility Rights

The capability accessibility rights, when held as rights in a capability, determine whether a module called by the capability can access the relevant capabilities. Notice that in the case of the free capability right, this determines whether a module called via the capability can make use of free capabilities whereas the free capability bits in the metarights determine whether this capability can be used as a free capability in other modules. When appearing in a capability the administrator and owner rights indicate that this is an administrator/owner capability.

4 The Thread Security Register

The Thread Security Register (TSR) is a pseudo register maintained by the kernel as part of the state of each user thread. It holds a set of rights currently associated with the thread. Its current values are stored at the base of the thread stack and are recorded in the linkage segment on each inter-module-, co-module-, and library call and on bracket routine activations (and restored on the corresponding returns). Its current values are available only indirectly to active modules and bracket routines via kernel instructions. Its content is extremely security sensitive and it is fully protected from direct user access.

The permissions in the TSR follow the same rules as those for access rights in capabilities. Initially all the rights are set (implemented as 1 in the TSR), i.e. all permissions can initially be used, but can be reduced (unset/turned off, i.e. with the value 0). A permission which has been turned off cannot be explicitly turned back on. To reduce the rights, the kernel uses an intersection operation.

A summary of the TSR structure appears in Figure 26.6. The rights fall into four groups (thread control rights, confinement rights, environmental rights and capability accessibility rights).The thread control rights are described in section 4.1. The confinement rights were explained in Chapter 25 (see Figures 25.1 and 25.7). The environment rights were described in section 1.3 above and the accessibility rights in section 2 above. When they appear in the TSR the administrator/owner rights indicate whether the thread can make use of administrator/owner privileges.

Each set of rights is repeated in primary and secondary sections, as will be explained below. They are stored in the TSR as a bit list, with each bit representing a specific right. If a bit is set, the thread has the corresponding right; if the bit is not set, the right is denied.

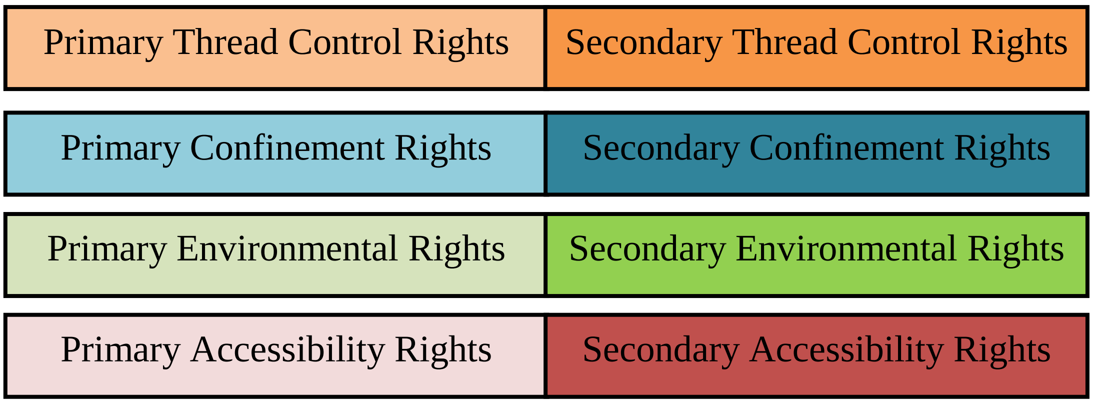

Primary Thread Control Rights
Secondary Thread Control Rights
Primary Confinement Rights
Secondary Confinement Rights
Primary Environmental Rights
Secondary Environmental Rights
Primary Accessibility Rights
Secondary Accessibility Rights

Figure 26.6: Thread Security Register

When a thread is initially created, all the rights are set. As the thread proceeds, some or even all of these rights may be removed or changed as described below; an application cannot restore its own rights. The rights are perpetuated from call to call in appropriate cases[146].

The removal of rights can be effected by an application or bracket routine using a kernel refinement instruction. Applications and bracket routines can also examine the current contents by executing kernel instructions. These possibilities are described below.

4.1 The Thread Control Rights

The owner of a thread may wish to control its use. There are two subgroups in this category (see Figure 26.7). The first group (coloured brown) applies to standard SPEEDOS operations. The second group (coloured red) applies to Internet operations involving non-SPEEDOS nodes. The latter are explained in chapter 34 section 7.3.2.

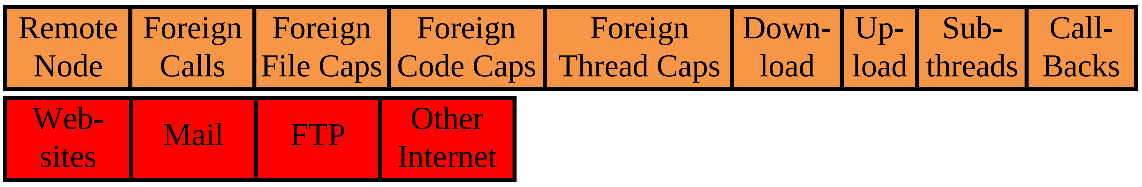


Figure 26.7: Thread Control Rights: An Overview

`permit_remote_node`: When unset, the kernel prevents a thread from being transferred to another node.

`permit_foreign_calls`: When unset, the kernel prevents calls to mod-

[146] The full rights are temporarily restored when the kernel makes a forced call to handle a synchronous error.

ules owned by users other than the owner of the thread.

`permit_foreign_file_caps`: When unset, the kernel prevents the thread from making use of *any* file capabilities for modules owned by users other than the owner of the thread.

`permit_foreign_code_caps`: When unset, the kernel prevents *any use* of code capabilities for modules owned by users other than the owner of the thread;

`permit_foreign_thread_caps`: When unset, the kernel prevents the thread from making use of *any* thread capabilities for modules owned by users other than the owner of the thread.

`permit_download`: When unset, the Container Manager (a privileged kernel co-module which organises downloads and uploads, see chapter 28) prevents the thread from initiating downloads.

`permit_upload`: When unset, the Container Manager prevents the thread from initiating uploads.

`permit_subthreads:` When unset, the thread cannot create subthreads.

`permit_callbacks:` When unset, the thread cannot invoke or support call-back routines[147].

`permit_websites:` When unset, the thread cannot access non-SPEEDOS websites.

`permit_mail:` When unset, the thread cannot access non-SPEEDOS email systems.

`permit_FTP:` When unset, the thread cannot access non-SPEEDOS FTP facilities.

`permit_other_internet:` When unset, the thread cannot access any non-SPEEDOS Internet facilities.

Initially these confinement rights are all set for all the threads of a process and are stored in page 0 of the process container, but the Container Manager provides a routine which allows them to be reduced (for all threads in the process). When a new thread is created, the current values held in page 0 of the process container are copied into the Thread Security Register, where they can be further reduced for an individual thread, using the instruction

```
 refine_tc_rights(bitlist tc_rights)
```

The `bitlist` parameter `tc_rights` provides a bit list of thread control rights in which the rights to be reduced are set to 0. The remaining bits are set to 1 in the input parameter and are not modified in the TSR.

[147] see chapter 20 section 8.5 and chapter 28 section 7.

4.2 Understanding the Rights in the TSR

Control over the environmental and confinement rights cannot be managed simply by initialising all the rights in the TSR for a new thread to the *permitted* state, then allowing refinement instructions to reduce these. The reason for this is illustrated by the following example.

A user has a general purpose thread in which it can invoke a command language interpreter (CLI) or an equivalent graphical interface to execute different commands. Suppose that via the CLI an edit interface of a text file module is invoked. This in turn calls a dictionary module used for spelling checking. The CLI module is the start-up module for the thread, which is invoked "automatically" when the thread is first activated, on the basis of the capability passed to the thread as part of the thread creation activity. Depending on its design, this could be an independent program module without file data, or an application module which uses file data to keep a log of the modules which it is required to invoke. If the former, then it might have a capability for a log file module (passed to it when the thread is created).

The CLI may invoke application modules for which it obtains capabilities from a directory (i.e. not passed to it as parameters). The applications which it invokes vary at the user's choice, and might be independent program modules and/or file applications. The edit command might be an independent program module which accesses text files via free capabilities, or it might be a semantic routine of a specific text file module. It will also need to communicate either directly or indirectly with a monitor/keyboard device driver module to receive and display text. And it will possibly need to access a dictionary file to check spelling, and a further file containing user preference settings. Eventually it (and the CLI) may need to access a logout module.

This variety of possibilities illustrates that it is not sufficient to use the same confinement permissions or simply to reduce them as a thread proceeds. A more dynamic mechanism for controlling these rights in the TSR is therefore necessary.

4.3 Primary and Secondary Confinement Rights

The following mechanism does not claim completely to solve the problem, and in some cases may need to be combined with the use of qualifiers and their bracket routines to achieve the desired security.

Because a user owns his own processes and their threads, he determines at least the first module to be called and therefore the capability used to call it[148],

[148] The only exception may be the first capability used to create the first process of a new user.

and can therefore reduce the rights in this capability to suit the environment in which it will execute. He can also pass further capabilities as parameters to this module, which can be used for making calls to further modules in the thread; hence he can also reduce the rights in these capabilities. Furthermore he can determine – or reduce – the confinement permissions of capabilities for the modules which he owns (using the capability copy operation), before placing them in a directory. In other words he has considerable control over the confinement permissions held in many, though not necessarily all, capabilities used in his threads to make calls.

Taking advantage of this, the proposed mechanism organises the environmental and confinement permissions in the TSR and in capabilities into two groups: primary and secondary environmental, confinement and accessibility rights. The primary rights apply when a user-controlled capability is used to invoke a module. The values of these are copied afresh into the TSR as part of the call mechanism. The secondary rights are used for all modules which are called by such modules on the basis of *uncontrolled* capabilities. The primary confinements are never carried over on an inter-module call, but the secondary confinements are copied into the TSR and applied to all modules called using uncontrolled capabilities. When a *controlled* capability is used to make a call, both the primary and secondary rights in its capability replace those in use up to that point.

4.4 Distinguishing Controlled from Uncontrolled Capabilities

In the sequel, a module which is invoked via a controlled capability is referred to as a *controlled module*. Otherwise it is an *uncontrolled module*. Capabilities are considered to be controlled if any of the following conditions is met.

a) The start-up capability is by definition a controlled capability.

b) A capability passed as an input parameter by a controlled module within the thread to another module in the thread is also considered to be a controlled capability. The receiving module might nevertheless be an uncontrolled module. (The justification for this is that the owner of the thread can reduce the rights in the capability, and is aware how the module is to be used.)

c) The owner of the module addressed by the capability is the owner of the thread in which the module is being activated. (The argument for this is that even if the capability has been passed to a different user, it must have been created by the owner of the module which it addresses. The other user cannot increase the rights beyond those which were in the capability when he received it.)

4.5 Examining and Reducing Rights in the TSR

There are 4 pairs of kernel instructions for examining and reducing the rights in the TSR. Each instruction pair consists of a refinement instruction which allows the rights to be reduced and a further instruction for examining the current state of the rights. There is such an instruction pair for each of the four rights categories. These conform to the following pattern.

(a) the rights refinement instructions:

```
refine_[tc|conf|env|acc]_rights
                         (bitlist rights; boolean primary)
```

The `bitlist` parameter `rights` provides a bit list of rights in which the rights to be reduced are set to 0. The remaining bits are set to 1 in the input parameter and are not modified in the TSR. The boolean parameter `primary` indicates whether the primary or secondary rights are to be refined.

(b) the rights enquiries:

```
bitlist query_[tc|conf|env|acc]_rights
                         (boolean primary)
```

On return the `bitlist` result shows the current settings in the TSR. The boolean parameters specify which subset of the environmental parameters is to be returned.

5 Container Confinement

Confinement techniques based on capabilities are intended to restrict unwanted activity by individual users or threads. But it is also possible to provide some blanket restrictions on containers, which apply globally. These can give the owners of containers control over the use of the container. They are stored in the protected area of page 0 of the container.

Illustrated in Figure 26.8, the container confinement rights determine whether information in the container can be transferred to another node via a download or upload, whether they can be used by a thread the owner of which is not the owner of the container and whether they can be used by a thread belonging to another node which has been transferred temporarily to the current node following a remote inter-module call.

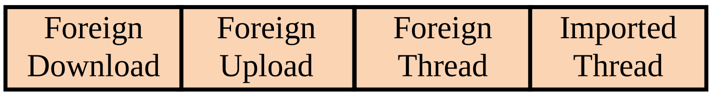


Figure 26.8: Container Confinement Rights: An Overview

These can be modified by the owner or an administrator of the container and they are held in the privileged area in the container's page 0. However, they are not transferred to the TSR. The first two permissions are checked by the Container Manager before initiating a transfer to another node. The third and fourth rights are checked by the kernel as part of an inter-module call to a destination in the container and in the `load_free_cap` instruction. These rights can only be changed by the container's administrator. There is no mechanism for them to be changed in any other way. They are not subject to the normal rule that rights can only be reduced, since allowing the administrator of a container to change them allows more flexibility.

6 Utility Programs

The mechanisms described so far are carried out by the kernel when a thread requests them. Here we show that it is also possible to supplement such checks by means of free-standing utility programs.

6.1 Examining New Code Files for Hidden Capabilities

Because the information in SPEEDOS modules is held in structured segments which have a known root segment, it is possible to write utility programs which can search these in a systematic way, as this example illustrates.

A utility program could be written which searches newly acquired code files before they are put into service. Such a program could, for example, search for constant segments and list (or invalidate) all the module capabilities which are embedded in the program.

In order to do this the code file would have to be viewed as a data file and the utility program would need a free capability for accessing it. There is no technical problem in achieving this, assuming that the ownership of the code file is transferred to the user or system manager, etc. This approach would reduce the need for some bracket routines.

6.2 Assistance in Setting Rights in Capabilities

Setting the rather daunting list of rights described above, if carried out directly by users, would be a tedious and error prone activity. For this reason SPEEDOS should be accompanied by a utility program which carries out much of the work involved.

This might be based on the provision of a formalised specification of programs provided by the vendors of program modules. It would in any case be an important step towards more transparency in computer systems to expect that a readable specification is provided by code developers, since it would greatly increase the transparency of code functionality and hence provide a significant step towards more secure systems.

One form that such a specification might take could be based on a template which requires the developer to

(a) provide an overall description of the program in *plain* English (or appropriate foreign language),

(b) list each semantic routine provided by the code;

(c) give each semantic routine a standard symbolic name;

(d) describe which semantic routines of other modules (including device drivers) each semantic routine needs to call *and why*;

(e) explain where it obtains the required capability to call each such module (e.g. as an input parameter supplied by the user, or from one of its constant segments);

(f) list which capabilities, if any, each semantic routine needs to return to its calling module;

(g) indicate for each semantic routine whether it places capabilities in its file segments *and why*;

(h) whether it creates and activates threads *and why*.

The above list is not exhaustive and should be extended by adding further points which can be automatically translated into settings for the rights within capabilities and the TSR.

Part 7
Basic Networking

Chapter 27
Partitioning and Relocating Discs

In chapter 23 a number of simplifying assumptions were made about the organisation of the virtual memory. One purpose of the present chapter is to replace these assumptions with a more realistic description of how SPEEDOS should really function. We begin by describing the partitioning of discs.

1 Partitioning Discs

When it is initialised a SPEEDOS disc can be subdivided into separate partitions. Each partition can be viewed more or less as a separated disc (except of course that such partitions are mounted and dismounted together). The question then arises how they are uniquely named. The unique internal name of a SPEEDOS disc consists of a 64 bit node number (of the creating node) and a 64 bit disc number, whereby the node number is system-wide unique and the disc number is unique within the node.

The final 4 bits of a disc number can be used as a partition number. In this way nothing changes substantially except that the system can immediately recognise which "disc numbers" belong together as logical partitions of the same physical disc. A disc which is not subdivided into partitions is regarded as a disc which only has a single partition numbered 0. Hence when numbering new physical discs, the actual discs have a 60 bit number which is incremented with each newly initialised disc (see Figure 27.1).

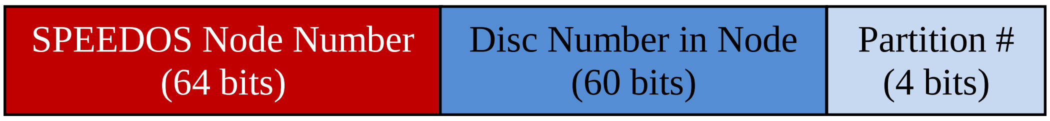


Figure 27.1: A SPEEDOS Partition Number

Each partition has its own separate disc directory and page tables (as described in chapter 23 sections 2 and 3), but of course these have different physical disc addresses. This is achieved in that page 0 of the actual physical disc becomes a directory for the actual disc start addresses and lengths of the different partitions. The structure of this partition directory is simply a list of entries, one per partition, whereby each entry (which is indexed by partition number) consists of fields <disc address of start of partition, length of partition, access properties>. The length of a partition is an integral number of 8 KB pages. The access properties include a "read only" bit, which if set indicates that the pages of the partition can only be read. This can be used to indicate devices which can only be read (such as CD ROMs, DVD ROMs), but also normally writeable disc partitions which must not be overwritten. Similarly it can be used to indicate that the partition holds the SPEEDOS kernel or the system co-modules needed to boot the system). The resolution of page faults, etc., as described in chapter 23, must be straightforwardly adjusted to take account of the existence of this table.

2 Moving Discs from one Computer to Another

So far it has been assumed that discs remain at the node on which they were created and that containers remain on the same disc throughout their existence. This simplified the description in chapter 23 because the unique identifiers of discs contain the unique node numbers of the creating node and the unique disc numbers of the disc (and partition) on which they are created. This is a very useful starting point for addressing containers since it helps to locate them rapidly. If, for example, a capability for a container has a node number 233, a disc number 11 and a container number 12345, then the obvious place to search for the container is by finding node 233, looking up disc number 11 and accessing container 12345. In the vast majority of cases this approach will be successful, but not always.

In reality users sometimes need to move discs from one computer to another (often close by, but possibly at the other side of the world). Of course in some cases copying a container or even a disc and deleting the original (and thus renaming the container and/or disc) would offer a satisfactory solution, but not always, because this would prevent all users who already have capabilities with old names from gaining access as a result of the changed name. Of course the users could continue to access the original disc or container if it were not deleted after making a copy, but then the other users would not see any updates made to the new version.

We need to tackle two problems associated with moving a disc to another computer. Since it has become commonplace physically to attach a removable disc to virtually any computer (e.g. via USB connections) it is of paramount importance to be able to ensure that unauthorised users cannot gain access to the information contained on the disc. But it is equally important that those users who *are* authorised can actually access the disc when it is mounted on another computer. (At this stage we do not consider access over a network since that issue is dealt with to the next chapter.)

2.1 Preventing Unauthorised Access to Information on Disc

In this section we discuss how it is possible to prevent hackers and others from accessing the information on a removable disc by mounting it on another computer and attempting to access it illegally via that computer (e.g. by using a hacking program to read it directly, e.g. disc block by disc block).

2.1.1 Accessing a Disc on an Unauthorised SPEEDOS Computer

If the computer on which the disc is mounted is a SPEEDOS computer, the following precaution can be taken[149].

When the disc is first mounted, page 0 of its partition directory must in any case be read. At this stage it is possible to check if the SPEEDOS node on which it is being mounted is authorised to access the disc. This is achieved by listing authorised SPEEDOS nodes in the disc's partition directory. The list, known as the Disc Authorisation List (DAL), can be set up by the owner of the disc (or another user with an appropriate capability). To do this he calls a semantic routine of the disc directory module while it is still on-line at an authorised computer. This mechanism will typically be used in situations such as a group of home computers, or when the disc's owner plans to use the disc on a computer which is geographically away from his home location. The node on which a disc is initialised will normally be the first entry in the DAL, but the owner can add entries to the list and remove them from the list by invoking routines of the appropriate disc manager co-module (see Figure 27.2).

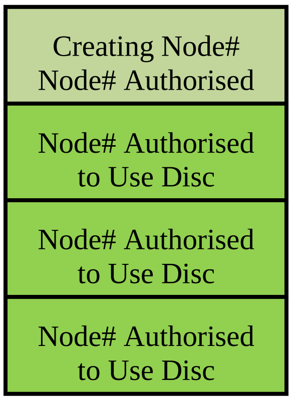


Figure 27.2: A Disc Authorisation List (DAL) [Version 1]

This mechanism provides an additional precaution over and above the other security tools available in SPEEDOS. But unfortunately, as described above it

[149] If an attempt is made to access a SPEEDOS disc on a non-SPEEDOS computer, the encryption technique now described should make this impossible.

does not solve all the problems. Hence it is called Version 1.

2.1.2 Encrypting Pages on Discs

A problem can arise if a hacker takes the disc to a non-SPEEDOS computer and reads the information block by block. To prevent this two of my former colleagues and I sketched out a plan how such protection could be effectively achieved on MONADS systems using encryption (and how the secure booting of a MONADS system could also be achieved by means of a similar technique) [24].

The basic ideas behind encryption techniques were introduced in volume 1 chapter 4 section 3. Readers not familiar with encryption concepts might like to read that section again. There it was explained that only symmetrical encryption (i.e. where the same key is used both to encrypt and to decrypt data) can reasonably be used to encrypt the content of discs, because the encrypted text (or program, etc.) has the same length as the unencrypted text, i.e. a page of encrypted text (on disc) has the same length as its unencrypted counterpart in the main memory of the computer.

However, this raises the problem how the symmetrical key can be distributed to authorised persons without allowing the same distribution route to be used by non-authorised persons. The normal solution is to use an *asymmetric* key on the symmetric key in order to distribute the latter securely. An asymmetric key has the disadvantage that the encrypted version of a text (or in this case a key) is not necessarily the same length as the plain text version (which is one reason why the asymmetric keys cannot be used to encode the page). But it does have the important advantage that the public key[150] used to encrypt a message (or text, etc.) may be publicly known (e.g. it might even be published in a newspaper or openly on the internet). But decryption can only be achieved via the corresponding private key (which is kept secret).

How can this be used to secure removable discs? We assume that the kernel at each node of a SPEEDOS network has its own (different) asymmetric key pair. The nodes on a SPEEDOS network can safely answer enquiries from other nodes about their public keys (since these can be publicly known). Hence a user wishing to take his disc (which was created and used on Node A) to Node B can ask that system manager (or other user) at Node B to provide him with its public key (even by post or in a plain text email).

Having received the public key of Node B, the kernel at Node A (on which the disc is still mounted) can use this to encode its own symmetric key and place the result in the DAL in the disc directory. Later, when the disc has been mount-

[150] see chapter 4 section 2.

ed on Node B, the kernel of Node B can, by using its own private key, learn the symmetrical key used to encrypt the disc and henceforth use it to work with the disc.

2.1.3 Encrypting the DAL

But there remains a small problem. Is the page containing the DAL itself encrypted, and if so using which technique(s)? Remember that the DAL may list keys for several destination nodes, which each have different asymmetric key pairs.

Under normal circumstances the page containing the DAL would be encrypted using the symmetric key of the source node, but precisely this symmetric key should be kept secret and therefore should not be readable on a computer which does not know the key! Remember also that Node B does not know the order in which the DAL entries are held, so that it cannot simply use its private key at a fixed position in the page! And remember that each entry in the DAL has been encoded using a different public key, one of which is its own (but only if it is authorised)! I suggest the following solution (see Figure 27.3).

Creating Node# A	Symmetric Key used to encrypt disc	Length of Public Key for Node A	Public Key for Node A
Node# B Authorised to Use Disc	Symmetric Key used to encrypt disc	Length of Public Key for Node B	Public Key for Node B
Node# C Authorised to Use Disc	Symmetric Key used to encrypt disc	Length of Public Key for Node C	Public Key for Node C
Node# D Authorised to Use Disc	Symmetric Key used to encrypt disc	Length of Public Key for Node D	Public Key for Node D

Figure 27.3: Protecting Entries in the Disc Authorisation List (DAL) [Version 2]

The start of the DAL is always in a fixed position in page 0 of the partition directory of the disc, and it has a fixed number of entries, each with a fixed *maximum* length (because the public keys used to encode the symmetrical code may not produce entries of the same length). Each publicly encoded entry has a length field. The DAL itself (in contrast with the rest of page 0) is NOT encoded, except via the various public key entries, but the rest of the page is encoded using the symmetric key of the source node[151]. This can then only be read using the symmetric key, after it has been recovered. Figure 27.3 uses colour coding to show that the different entries in the list have been encrypted by different public

[151] It is probably more convenient to place the DAL in a separate page (e.g. page 2 of the disc directory.

keys.

Finally in order to discover whether a disc is authorised the destination kernel decodes (only) the DAL using its private key. If it discovers that its node number is on the list and its public key details match, it is authorised to mount the disc. It can use information in the corresponding DAL entry as the symmetric key for the disc. If it is not in the list a failure message is generated and the disc cannot be used. The public key details are included not because they are needed as such, but to ensure that a different node's encoding does not decode to another node number by chance. (The likelihood is very small, but cannot be fully excluded.)

This procedure not only ensures that the destination computer, if authorised, can discover the symmetric key used to encode the rest of the disc, but also that no other computer – not even other SPEEDOS computers listed in the DAL – can discover the node numbers of the other authorised nodes!

For the owner of the disc all that is involved in this process is to inform the disc directory of changes to the list of nodes at which the disc may be mounted. The kernel's disc process responsible for the disc must of course use the symmetric key which it has recovered for all further accesses to the disc.

The next chapter describes how networking is organised in SPEEDOS. In principle it would be possible to use this route to obtain a public key from a partner computer, *but only if that computer is on line*. The above method is therefore to be preferred for the task which we have described, i.e. accessing a removable disc which has been plugged into a different computer.

One final point on this theme: what we have described should prevent thieves from acquiring information from a stolen disc, but it does not prevent them from overwriting the information! This unfortunate fact reminds us that encryption alone is not the key to achieving high security, as some computer scientists and others tend to think...

2.2 How Authorised Users can Access the Content of a Moved Disk

After the kernel has recognised another SPEEDOS disc using the above procedure, the question still remains how a user can access its content.

In chapter 23 the simplifying assumption was made that all discs created at a node are always mounted and available at that node. That assumption is self-evidently too stringent. It is possible to create many more disks at a node than can be concurrently mounted. Furthermore discs sometimes fail and have to be destroyed. What is clearly needed is a further table, belonging to the kernel disc process, which we call a Local Mount Table (LMT)[152]. This lists all the discs and their partitions that are currently mounted at a SPEEDOS node, together with the address of the physical disc drive at which it is mounted and, for rapid access, the disc address of each partition's directory page 0 (see Figure 27.4).

Creating Node#	Disc/Partition # on creating node	Physical Drive#	Disc address of Directory Page 0

Figure 27.4: The Local Mount Table

The table is set up by the VM process when a node is initialised. At that point the process enters the details of the disc used to initialise the system (known as the boot disc) into the table, along with all its partitions, and those of any further discs which are already mounted at the node at initialisation. Thereafter the VM process enters any further discs/partitions when an interrupt indicates that a new disc has been mounted (after checking that it is accessible according to the procedure described in section 2.1.3) and removes entries as the corresponding discs are dismounted.

2.3 Resolving Page Faults on a Locally Mounted Foreign Disc

The first page fault for a module occurs on an inter-module call, as described in chapter 23 section 4.4. One of the tasks of the user request process handling the IMC is to check the local mount table is to establish whether the appropriate disc is on-line. It can do this because the capability passed as an operand to the IMC contains a creating node # and disc/partition #. This can be checked against the Local Mount Table to establish whether the required disc/partition is currently on-line, and if so it sends a "request and lock page 0" message to the VM process, and continues as described in chapter 23 section 4.4.

2.4 Accessing Moved Discs which were Created at the Current Node

So far we have considered the situation where a disc has moved from one node to another from the viewpoint of the node where the disc has been mounted. But

152 This name, and much of the remaining content of the present chapter, is heavily reliant on the MONADS design, as extended by Frans Henskens and described in his PhD thesis [20], which Prof. Henskens has kindly made available on the SPEEDOS website.

from the viewpoint of the node which created the disc, the latter is no longer directly available, despite the possibility that user threads on the creating node might still want to access it. However, this is self-evidently not possible without the availability of networking, which is the subject of the next two chapters.

Chapter 28
Accessing the Internet

This chapter examines how SPEEDOS can function in a network environment, in particular in the context of the Internet. It does not go into technical details of how exactly the Internet works, since there are many experts in this field[153], but merely shows how SPEEDOS ideas can be made to work in the Internet environment. The same principles apply, with detailed differences, to other networks.

1 Accessing the Internet

The first issue is to determine how a thread executing on one SPEEDOS node can access or modify information stored on a different node. We take as a starting point the transfer of information between SPEEDOS nodes.

Information transferred over the Internet must be secured to prevent unauthorised users from eavesdropping. The scheme described in chapter 27 for preventing discs from being read by unauthorised parties can easily be adapted to deal with this situation.

Since each node has its own asymmetrical key set, nodes can easily send secure messages to each other simply by each SPEEDOS sender encoding its messages using the receiving node's public key, and each SPEEDOS receiver can use its own private key to decrypt messages which it receives.

2 Remote Paging

In the late 1980s an ethernet-based network of three MONADS-PC systems, in which each system had its own ATU, was built to experiment with the issue of how the MONADS architecture (the predecessor of SPEEDOS, which also supported a persistent virtual memory) could be adapted to networking. The basic

[153] I am not an expert in the technicalities of the Internet, and it may be that some details in this chapter may need some small corrections. A further discussion of the Internet appears in chapters 34 and 35.

idea was to test out the feasibility of *remote paging*, a concept which my former student David Abramson and I first proposed in 1985 [25]. This concept was then developed in detail in the PhD thesis of Frans Henskens [20], another former student[154].

The basic idea behind remote paging is that pages are transferred across a network when a page fault occurs for a page which resides on a remote node. Such page faults are implemented by transferring the required page from the remote node (the server node, which owns the faulting page) to the page-faulting node (the client node). This should be entirely transparent to the user, who need not be aware of the fact that he is working in a network environment.

As the co-inventor (with David Abramson) of remote paging[155] I was keen to use this mechanism in SPEEDOS, following Frans Henskens' successful implementation of the idea in the MONADS context. However, when I again looked into this technique I realised that developments in SPEEDOS made it difficult to use the technique so cleanly and efficiently as had been possible in MONADS. Some of the relevant issues for this decision were:

a) the SPEEDOS implementation of semaphores[156], and

b) the introduction of security sensitive co-modules[157].

For these reasons it was decided to follow an alternative route for SPEEDOS, viz. remote inter-module calls.

3 Remote Inter-Module Calls

Whereas with remote paging the data and code are transferred across the net-

[154] This thesis was supervised by another of my former students, John Rosenberg. Henskens has kindly agreed to place a copy for download on the SPEEDOS website: http://www.speedos-security.org/

[155] Many U.S. researchers think of Kai Lee as the inventor, overlooking the fact that our publication of the idea in a Hawaii conference in 1985 preceded the publication of [28] in 1986.

[156] Semaphore operations modify the page in which the semaphore is held. Since in SPEEDOS they are held in the file pages of an application container (see chapter 21) the result is that each time such an operation occurs the page becomes a "writer" and thus prevents further readers from accessing the page; this makes the system, especially nodes which have many readers, inefficient by causing otherwise unnecessary page transfers and the delaying of threads..

[157] Containers contain both system information and application information, possibly in the same page. These would have to be kept in separate pages and only application data could be remote-paged. Furthermore if the MONADS design were used, the SPEEDOS kernels on all nodes would have to be identical, which in the Internet environment would be unacceptable (in contrast with a homogeneous local area network, for which Henskens' original system was designed). At the time the MONADS remote paging work was carried out, the Internet did not yet exist in its present form. With the solutions presented in this chapter only page 0 of a container has a fixed format.

work to the thread needing them, with remote inter-module calls (RIMCs), the thread is taken to the data and code. In other words RIMCs can be loosely considered as the SPEEDOS equivalent of remote procedure calls[158] and remote method invocations[159] in conventional systems (except of course that (a) the call is to a SPEEDOS module, with all the accompanying security measures, and (b) the implementation differs significantly from conventional RPCs).

The basic idea underlying an RIMC in SPEEDOS is that a semantic routine of a module which is located on a different SPEEDOS node can be called, without the caller necessarily being aware of this. From the viewpoint of the caller the call is exactly like a normal inter-module call (with exactly the same parameters).

From the system viewpoint the kernel recognises the difference via the capability presented to the IMC, which contains a node number that differs from the number of the node on which the call is made.[160] We refer to a node issuing an RIMC as node A, and the node carrying out the call as node B.

3.1 An Overview of RIMC Handling at the Client Node

The SPEEDOS technique selected to implement RIMCs is the use of surrogate threads at the destination node B. The thread which initiated the call (at node A) is referred to as T1 and the surrogate thread which implements the call (at node B) as T2.

When the kernel receives an inter-module call request at node A and recognises that the node on which the called module is located (the server node) is a different SPEEDOS node, it passes on the call to the server node (node B). But before it does this it must

a) store the registers of T1 in a new linkage segment on the T1 thread stack,

b) check that the call is valid (e.g. by comparing the called routine number with the access rights in the capability),

c) create an IMC stack record on the stack of T1, in which the operands of the IMC are noted[161],

d) establish whether the IMC needs to be handled as an RIMC, and if so carry out the actions listed in e) to i),

[158] see https://en.wikipedia.org/wiki/Remote_procedure_call

[159] see https://en.wikipedia.org/wiki/Distributed_object_communication

[160] In fact the situation is rather more complicated than this suggests, as will become clear when we later discuss the possibility that removable storage devices and/or containers can be moved from one node to another. But at this stage the remote IMC mechanism is far easier to understand provisionally if we temporarily put this issue aside.

[161] Before the RIMC is issued, the thread will have used the kernel call **`create_imc_params`** to prepare the parameter segments (see chapter 20 section 6.2).

e) check in the thread security register that T1 has permission to make RIMCs[162] (and if not it creates a synchronous error interrupt),

f) complete any call-out brackets[163] associated with the current (i.e. the calling) module,

g) send a message to the user interrupt process that the current thread should be suspended by the UTS,

h) create a top of stack record which indicates that the call is being handled as an RIMC together with an indication of the node to which the thread is being transferred, and

i) activate a surrogate thread to advise the thread's Thread Control Manager that the thread has been transferred to node B.

Node A then sends a message to node B in which it provides details of the IMC, requesting node B to accept and take responsibility for the RIMC.

When it later receives a message that the RIMC has completed, the kernel then finds the stack of T1 (which has been suspended during the RIMC execution) and brings this back to life by copying the RIMC result parameters into the local result parameters. It then requests the user interrupt process to re-activate T1 and exits. When this resumes it executes any call-out postludes, etc. and returns control to the module which made the RIMC call.

3.2 An Overview of RIMC Handling at the Server Node

When node B receives the request, it checks whether it can handle the IMC at its own node, and if so it sends a positive acknowledgement to node A. (If not it uses the mechanisms described in sections 8 and 9 to locate the module and passes on the RIMC request as appropriate, advising node A. If the module cannot be located it sends an error message back to node A.)

It then claims a surrogate thread (known as an RIMC thread) and sets this up, using the information which it has received, and initialises this as appropriate, including creating a bottom of stack record which indicates that this is an RIMC call, noting the home node of the thread (node A). The kernel then proceeds more or less as if one of its own user threads had executed an IMC from its own node.

[162] see chapter 26 section 4.1.

[163] When an IMC occurs the call-out brackets are associated with the caller and must therefore be completed before the actual RIMC occurs, whilst the call-in brackets are associated with the target module and must therefore be executed after the transfer to the target mode is initiated. Since call-out brackets can make environmental enquiries about the target module, the kernel instruction `target_code` requires the kernel at node A to make an advance enquiry to the kernel at node B regarding the code of the target module. The implementation of bracket routines is described in chapter 24.

When appropriate the surrogate thread (T2) uses its Thread Control Manager (at the new node B) as a local thread would, e.g. when executing semaphore operations. Similarly it uses the UTS at node B whenever it needs to be suspended. If the module has associated call-in brackets these are handled on the RIMC stack of T2 as normal.

When the module called via the RIMC has completed its task and has written its results (if any) to the output parameter segment (addressed via segment register 1) it issues a **`return`** instruction (or in the case that the module's call-in brackets exit, a **`bracket_return`** instruction). When the kernel receives this it recognises from the bottom of stack record that the thread is an RIMC thread which is now completing. After arranging that an appropriate message be passed back to the calling node, it deallocates the RIMC stack and forgets the thread.

4 Decisions Affecting the Interface between Client and Server Nodes

4.1 The Thread Control Manager and the Synchronisation Library

Thread control managers play a significant role in the execution of threads under their control. We have seen in chapter 21 how they are involved in the organisation of semaphores. They occupy an intermediary role between the threads under their control (i.e. those threads which share the same process container) and the User Thread Scheduler (UTS). Thus when a semaphore operation is carried out by a user thread any associated `suspend` and `activate` operations initiated by the thread are directed (via the synchronisation library routines) to the appropriate Thread Control Manager, which then calls the UTS. Similarly `suspend` and `activate` operations not involving semaphores normally[164] also follow a similar route to the UTS. This arrangement has two advantages. First it means that for each thread in the system its Thread Control Manager knows what is happening to the thread and can potentially help, especially in error situations involving the thread. Second, the synchronisation library and the Thread Control Managers relieve the UTS of considerable work in terms of managing thread queues, which is important in terms of efficient scheduling.

But what happens when a thread executes an IMC instruction which leads to the further execution of the thread at a different node? The simple answer is that the RIMC surrogate thread at the server node (T2) has a different Thread Control Manager, viz. that provided in the surrogate process container. There is no problem in T2 using this and the synchronisation library at node B. In fact it would be extraordinarily difficult if these attempted to use the software at their home node! The only additional problem that might arise is if the thread in some

[164] The exception is when the kernel directly suspends a thread or activates a thread.

way were to "get into trouble" at node B, for example if it had some sort of an error or became involved in a deadlock. The error itself would normally be detected at node B where the thread was currently executing. But if this were not its home node, the latter would need to be informed, especially if the thread were not able to continue. In this case a message is passed back to the home node with an error code and possibly with further information from the Thread Control Manager at the server node).

The thread's Thread Control Manager at each appropriate node should be informed of the thread's movements between nodes.

4.2 Handling an IMC called by an RIMC (Surrogate) Thread

There is clearly no reason to forbid a surrogate thread at node B from making local IMCs to other modules at node B, provided that all the normal security rules are followed. Furthermore library calls (which are not possible as RIMCs) and co-module calls[165] from the code at node B should be permitted as normal.

4.3 Handling an RIMC made by an RIMC (Surrogate) Thread

When an RIMC surrogate thread is executing at node B, this might in turn call a module at a different node. Two situations are conceivable. The first is that the new RIMC involves a call to a module at a third node (node C). This is probably the more usual case. To keep the situation simple, the obvious answer is for the surrogate thread making the call to be treated in exactly the same way as has already been described, i.e. a new surrogate RIMC thread is activated at node C.

The second situation is if the destination of a call from a surrogate RIMC thread is back to the home node of the original thread, i.e. node A. A first assumption might suggest that this can in some way be handled on the home stack of the thread T1, but a little reflection will show that this could add further complications which are best avoided. Instead such a call should be handled in a surrogate thread at the home node, just as in the first case. (A call-back mechanism is provided when a surrogate thread needs to communicate with the thread from which it was activated, see section 7.)

4.4 What About the Thread Security Register?

The thread security register (TSR) is a pseudo-register maintained at the bottom of its stack by the kernel for each thread (see chapter 26 section 4). This contains a significant number of access controls, which should continue to apply if a thread is transferred to another node. Consequently it should be transferred as part of the current state each time a thread is transferred between nodes.

[165] See chapter 18 section 7.1.

5 Communication between the Client and Server Nodes

5.1 The Request to make an RIMC

The information which node A transfers to node B (excluding packet information for the network protocol) includes:

a) the node number of the sender (i.e. node A),

b) the number of the recipient (i.e. node B),

c) an error code (0 signifies 'initial request'),

d) a copy of the thread capability for T1 (needed for identification es[166]),

e) a copy of the thread security register,

f) a copy of the input and output parameters which were created on the stack of T1 in preparation for the RIMC,

g) a copy of the operands used for the IMC call.

5.2 The Confirmation

When node B receives the request, it checks whether it is responsible for carrying out the RIMC and then responds with a confirmation (or rejection) as follows:

a) the number of the sender (node B),

b) the number of the recipient (node A),

c) an error code (1 signifies 'receipt confirmed and accepted'),

d) a copy of the thread capability for T1 (for identification purposes),

e) a copy of the surrogate thread capability (T2) in which the RIMC is carried out (or 0 if an error has occurred).

If the error code shows 'receipt confirmed and accepted' the kernel at node A copies the surrogate thread capability to the top of the stack of T1. Otherwise it sets up a synchronous error and requests the user interrupt process to have the thread activated, so that it can handle the error.

If the request was successful the kernel at the initiating node activates a surrogate thread to advise the thread's Thread Control Manager that the thread has migrated to node B. This is explained in section 6 below.

5.3 The Completion

When the kernel at node B receives the final return instruction (i.e. an IMC `re-`

[166] This and other capabilities used only for identification purposes are all invalidated.

`turn` or if there were call-in brackets associated with the called module the final **`bracket_return`**) from the surrogate thread or if a non-recoverable error occurs for the surrogate thread, the responsible kernel sends a completion message to the initiating node, as follows:

a) the number of the sender (node B),

b) the number of the recipient (node A),

c) an error code (2 signifies 'success', 3 or more is an error indicator),

d) a copy of the thread capability for T1 (to identify the thread which made the original IMC),

e) a copy of the thread security register,

f) a copy of the input and output parameters containing the results of the RIMC.

On receiving the completion confirmation the kernel at node A:

i) activates a surrogate thread to advise the thread's Thread Control Manager that the thread has now been returned from node B, providing it with a copy of the error code (see section 6 below),

ii) copies the updated thread security register into the stack of T1,

iii) copies the returned input and output parameters onto its stack.

If the error code signifies that the RIMC was successful, the kernel then prepares all the appropriate registers for a normal return. If the error code signifies that the RIMC has an error, the kernel prepares the registers for a synchronous error (including passing the error code to the error handling routine). In both cases it passes a message to the kernel's interrupt process to activate the thread which issued the RIMC.

6 Surrogate Threads for Advising the Thread Control Manager

When a thread is transferred to or returned from another node, the thread's Thread Control Manager is advised by the kernel, using surrogate threads. For this purpose the TCM has an entry point in the code which cannot be called by threads other than surrogate threads (which is marked as such in the module's entry point list).

When the kernel wishes to activate this entry point it allocates a surrogate thread (known as a TCM thread) from a list of threads, which have as usual been prepared at system initialisation. It then sets up the thread to begin executing at this special TCM entry point (setting the code segment register and program counter to the beginning of the routine and segment register 5 to address the root pointer in the TCM's data file) and requests the user interrupt process to have this thread started. When the TCM has completed its task it calls the UTS routine **`killMe()`** as was described in chapter 22 section 8.2 (g).

7 Remote Call-Back Modules

A different situation that can arise is if a surrogate thread (here T2) executing an RIMC on a remote node (here node B) wishes to call a routine located at the original node A. For example, the module at node B (e.g. a banking website module) wishes to display its results on the user's screen at node A and possibly obtain further instructions from the user at an interactive terminal. This is achieved via *call back* modules[167], which typically reside at the origin (client) node A. A remote call-back module is a "normal" module which also provides call-back routines for remote IMC modules which it has called. In this case execution *begins* in the call-back module at node A, which then instigates the RIMC at node B.

7.1 Remote Call-Back Calls

The relationship between a remote call-back module and its RIMC module is illustrated in Figure 28.1.

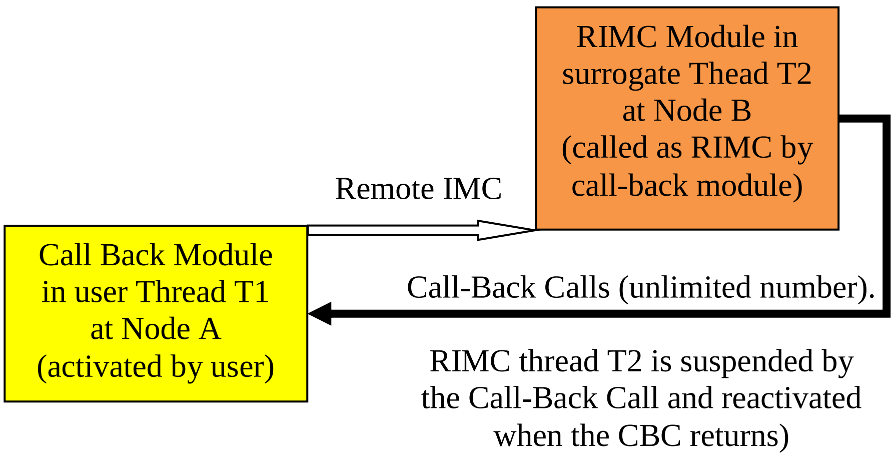

RIMC Module in surrogate Thead T2 at Node B (called as RIMC by call-back module)
Remote IMC
Call Back Module in user Thread T1 at Node A (activated by user)
Call-Back Calls (unlimited number).
RIMC thread T2 is suspended by the Call-Back Call and reactivated when the CBC returns)

Figure 28.1: Call Back Modules

When the RIMC module wishes to invoke a call back routine of the call back module it uses a kernel call **`CBC`**. This has an interface like a normal IMC which allows normal parameters (no pointers) to be passed back to the routine[168]. It indicates which routine is to be called by providing a routine number, which is an index into the call back entry point list for the module. The second parameter (as for a normal IMC, see chapter 20 section 8.1) is a boolean parameter indicating whether the caller is requesting read-only or read-write access to the module's file data.

[167] These are a remote version of the call back modules described in chapter 20, section 8.5).

[168] Hence it uses the kernel call **`create_imc_params`** to prepare for the call back call.

7.2 Handling the CBC at the Calling Node (Node B)

The kernel at the calling node B implements the CBC as follows. It first establishes whether a surrogate RIMC has issued the call and in this case:

a) stores the registers of T2 in a new linkage segment on the thread stack,

b) creates an IMC stack record on the stack of T2, in which the operands of the CBC are noted,

c) checks in the thread security register that T2 has permission to make CBCs (and if not it creates a synchronous error interrupt),

d) sends a message to the user interrupt process that the current thread should be suspended by the UTS,

e) creates a top of stack record which indicates that the call is being handled as a CBC together with an indication of the node to which the thread is being transferred, and

f) activates a surrogate thread to advise the thread's Thread Control Manager that the thread has been transferred back to node A.

Node B then sends a message to node A containing the following information:

a) the node number of the sender (i.e. node B),

b) the node number of the recipient (i.e. node A),

c) an error code (-1 signifies 'call-back'),

d) a copy of the thread capability for T2 (needed for identification es[169]),

e) a copy of the thread security register (which might have been modified by the RIMC),

f) a copy of the input and output parameters which were created on the stack of T2 in preparation for the CBC,

g) a copy of the operands used for the CBC call,

h) a copy of the thread capability for T1 (needed to locate the original stack).

When node A receives the message it sends a confirmation along the lines described in section 5.2 (*mutatis mutandis*).

7.3 Handling the CBC at the Called Node (Node A)

When the call-back message arrives at node A its kernel network process[170] passes this to a kernel call-back process, which uses the thread capability for T1

[169] This and other capabilities used only for identification purposes are all invalidated.
[170] See section 8.

to locate the original stack, and checks (by examining the stack) that the associated thread is suspended waiting for a return from the remote IMC.

The call-back process then sets up the stack to call the requested CBC routine, sends a confirmation message to node B and requests the user interrupt thread to activate thread T1.

7.4 Bracket Routines

Call-back modules may have both call-in and call-out bracket routines. The call-in brackets of the call-back module are executed in the thread of the call-back module (in this case on the thread stack of thread T1 at node A). They are identified in the Co-Module Table of the call-back module (see Figure 19.6). How these are applied at the node containing the call-back module follows the same pattern as usual, except that the call-out routines of the original RIMC call are *not* executed before the call-in routines associated with the CBC. These are only applied (as usual) when the RIMC module exits, returning back to the call-back module via a normal return. In other words CBCs appear to the surrogate RIMC thread (in this case T2) to be like internal subroutine calls from the viewpoint of bracket execution.

7.5 Application of Call-Back Routines

An important use of call-back routines is to allow websites to be designed in SPEEDOS without having to rely on the normal mechanisms currently used in the Internet. For example, the call-backs can contain code which allows them to display web pages at the website client node based on information passed as parameters to the CBC calls, without using HTML for this purpose. Of course this does not preclude the parameters of a CBC from including HTML[171] (or a capability for an HTML file which can then be downloaded). In this way the full range of SPEEDOS protection techniques can be used to implement websites in a secure manner (including secure downloads and uploads, as described in the next chapter).

A further possibility is to provide a general purpose call-back module which simply activates SPEEDOS websites, using HTML to display results. This will be discussed at the application level in chapter 35.

7.6 Call-Backs at a Single Node

We have described how call-back modules can be implemented as a remote activity, since it is anticipated that their main use will be for implementing SPEEDOS websites. But there is no reason why the same technique should be

[171] One advantage of initially using HTML would be quickly to convert non-SPEEDOS websites into secure SPEEDOS websites.

used to allow communication between a call-back module and a partner module to take place (with a much simpler implementation) at a single node. Such an implementation could be used conveniently to test website software without resorting at all to the Internet. And I expect that clever programmers will find other uses for this simpler mechanism.

8 The Network Process

The messages described above (and any further Internet communications) are transferred between nodes by the kernel's network process at the respective nodes. The network process is responsible not only for the transfers but also for encoding and decoding them. It also prepares the messages for the appropriate network, e.g. the Internet.

As such it has the asymmetrical key pair associated with the node and used over the Internet, but not the symmetrical key used to encode and decode pages on disc. It receives messages from other kernel processes at the same node in network message blocks. It is activated when it receives a new message block (as a result of a kernel reschedule). When a new Internet message arrives, the kernel interrupt analysis routine also activates it by placing the message in its input buffer.

The network process is also responsible for locating nodes to which messages are sent. For this purpose it maintains a Network Address Table (NAT), which holds the network addresses that it has so far acquired as a result of its users requesting and using such addresses. This simply consists of entries containing a unique SPEEDOS node number, an indication of the network to which a node is attached (which may for security reasons be a private network not reachable over the Internet), a network address within that network and the public key of the node[172] (see Figure 28.2). It may also hold a capability for a public directory (the "shared capability") at the node listed, thus enabling user level software at the current node to have a starting point for communicating with user level software at the node listed in the NAT (see chapter 31 section 9).

The first entry in the NAT is an entry for a SPEEDOS "directory" node, which can accept enquires about the location of other nodes. This can interface with other similar directory nodes (existing worldwide) to pass on enquiries received by it for which it currently has no entry. When a SPEEDOS node comes on line it should communicate its own details to its local directory node. In this way the information needed to communicate with other SPEEDOS nodes could grow rapidly. A security co-module initialises the NAT at system start-up and

[172] Although public keys can be generally known, it would of course be even more secure to keep public keys secret and made available only to other SPEEDOS kernels.

"shares" it with the kernel, securing it between system shutdowns and start-ups.

SPEEDOS Node#	Network to which attached	Network Address	Public Key	Shared Capability

Figure 28.2: The Network Address Table

9 A Note on Remote Login

Conventional systems provide a remote login facility which allows users to access their files from other systems. This is a dangerous facility, because it allows anyone who obtains a user's password secretly to access, copy and even destroy his files. SPEEDOS does not provide (and does not need) such a facility.

If a SPEEDOS user needs access to some of his files from a remote computer, as in a conventional computer, he first needs access to a thread on the remote computer. In SPEEDOS this will be a normal thread of a user process, possibly set up for this purpose. To give this thread access to the files on his main computer he simply needs an appropriate directory capability for these. He can supply the appropriate capability, e.g. on a memory stick, thus completely eliminating the need for a dangerous remote login facility.

10 Further Networking Activities Relevant to the Kernel

This chapter has shown how inter-module calls can be handled over the Internet as well as describing the basic functions of the kernel's Network Process, which is the process that handles network traffic at each node[173]. In the next chapter we build on this basic information to explain how the downloading and uploading of containers is organised by the kernel and how the kernel can efficiently locate discs and containers which have been moved between the nodes of a network.

[173] How the kernel actually uses the Internet to transfer its messages is discussed in Chapter 34, which also considers how other Internet activities, such as email, can be handled in SPEEDOS.

Chapter 29
Locating and Transferring Objects in the Internet

In chapter 28 a basic form of network activity ('remote inter-module calls') was described. This allows a user who is in possession of a capability for a module located at a different node to activate the module. It also described how a kernel can communicate with other kernels and it ended by explaining how SPEEDOS nodes could use "directory" nodes to help them locate other SPEEDOS nodes. In this chapter we continue the story by considering how discs which have been mounted on a 'foreign node', i.e. a node which is not the node on which the disc was created (the 'home node'), can be located and how containers can be moved.

1 Locating Moved Discs

Chapter 27 described how a user can take removable discs to other computers and use them directly, even if the disc's home node is not online. The inverse issue is that a user at the creating node cannot assume that all the discs which it created are available at its own computer. Thus if the kernel discovers from the Local Mount Table (LMT, see Figure 28.2) that one of the discs which it has created is not currently online at its own node, it cannot simply assume that it is offline. In this case it must be able to check whether its disc is mounted elsewhere. It would be possible to extend the NAT and the SPEEDOS directory nodes to provide information about moved discs, but that seems to be an overkill solution. I suggest the following alternative.

When a disc has been successfully mounted on a foreign node, the kernel at the foreign node attempts to send a message to the creating node to advise it where the disc is mounted. If this succeeds the home node notes this in a Moved Disc List, with entries indicating the disc number and the number of the node on which it is mounted. Later when it is dismounted the same kernel advises the creating disc's home node of this. Thus when the home node receives a request (e.g. a local IMC or a request by another node to accept a RIMC) it can establish the present whereabouts of the disc from its Moved Disc List and act accordingly by passing on the request to the foreign node on which it is mounted.[174]

If the disc's home node is not on-line to receive either of these messages, the appropriate message is added to a queue of unsuccessful messages, for which further attempts to send are made at regular intervals[175]. If the original mounting message for a disc cannot be sent before the corresponding dismounting message is due to be sent, both messages are cancelled.

2 Moving and Locating Containers

Containers must sometimes be moved from one disc to another. For example, a user of a computer utility on which his files are stored may for some reason need to change to another location. In this case his files might not be stored on a computer or disc which is entirely his property, so he might want to move those containers which are his onto a disc which he owns and can take with him. How can he go about this?

The simple answer is that he can make a destructive move (i.e. copy the container to the new disc and delete the old container). But that does not necessarily solve all his problems, since there could be capabilities (held both by the user moving the container and by other users at the original or another node) which address the moved container and which hold access rights that they may still wish to exercise.

If no further measures are taken to allow such users to gain access to the container at its new location, the effect is that the capabilities are revoked. Thus by relocating a container (even to another location on the same disc) a user has an effective way of revoking capabilities, which, it will be recalled[176], is one of the problems with using capabilities. Consequently it may be sensible to allow this situation, but only as an option, because this may not be the container owner's intention. So we now have two ways of handling the issue.

2.1 The Revocation Option

Assuming that the new location is to a disc mounted on the same node as the existing container, the move can be implemented by the owner simply calling the Container Manager's `copy` routine (see chapter 23 section 7). Then in the second stage it calls the Container Manager's `delete` routine for the original file. (The issues arising with copying have been partly dealt with in chapter 19

[174] A further possibility is that the users who own discs can provide information about their current whereabouts.

[175] The kernel's network process could delegate this activity to a surrogate thread.

[176] See chapter 2 section 4.1.

and for the Internet context will be further discussed later in the present chapter.)

2.2 The Re-Use Option

At this point the important issue is how a container can be moved but can remain accessible to other users despite the fact that it has changed its identifier. The preferred solution for SPEEDOS follows the route commonly used to solve many problems in current systems, viz. by using indirection. This is managed by the Container Manager's `rename_container` routine.

The technique proposed to organise the indirection is for the information about the new location of a container to be placed in page 0 of the old container, with a flag indicating that the container has been moved and its new location. In this case the disc directory for the disc which previously held the container would not be zeroed on "deletion" of the container, but an attempt to access the container (e.g. in an inter-module call) would see that it has been moved and could then provide the information to forward the thread to the possibly correct location. I say *possibly* correct, because the "same" container might be moved more than once. In this case the container could be located by following the "chain" of forward references in the various page 0s.

A further advantage of this solution is that page 0 of the old location could also contain an access control list (ACR) listing those users (e.g. by unique identifier) whose threads are allowed to be forwarded. In other words this technique could be used to revoke the capabilities of some users while allowing other user continued access to the container.

2.3 A Possible Optimisation

If users repeatedly needed to access the moved container their access could be speeded up by adding advisory fields to capabilities[177] which consist of a <node #, disc #, container #> triple. This would increase the size of capabilities by 192 bits (from the beginning, not just when something is moved). The idea is that these fields are initially zero, but they could be used as necessary (and overwritten with the latest information) to advise the kernel where to search for moved objects and thus avoid a chain of accesses possibly to different nodes. Nevertheless what really matters is the identifiers in the original fields of a capability. However we do not propose that this should be implemented in SPEEDOS, since the greater majority of containers are never moved, and the technique would considerably increase the size of capabilities.

What we have not described in this section is how the moving of a contain-

[177] The idea of having advisory fields in capabilities was first proposed by Henskens in section 6.2.1 of his thesis [20].

er to a new node actually happens. This brings us to the next theme: downloading and uploading of containers.

3 Downloading and Uploading of Containers

One of the most common activities in the Internet is the downloading and uploading of files. In SPEEDOS terms this means the copying of a container from another Internet node to one's own node (downloading) or copying a container to another Internet node from one's own node (uploading). Thus the fundamental question becomes, how can containers be copied over the Internet?

In the discussion of copying containers in chapter 19 section 13 several different reasons were mentioned as situations in which copying might take place. The first of these ("to make a copy which the owner, or some other user, can use independently of the original") is also normally functionally the same as downloading or uploading, with the extra premise that the transfer of data takes place over the Internet (or other network).

Before such an operation can begin the system must ensure that it does not contain "problematic" capabilities[178] (e.g. owner capabilities[179]). For this purpose a semantic routine of the container's segment manager can be invoked to confirm that the download or upload operation is "safe".

3.1 Downloading

On current systems a website offering a download facility the origin node, (node A) indicates this in such a way that a node wishing to take advantage of the download offer (the accepting node, node B) can select it. In SPEEDOS terms this means that node A provides a capability for the container on offer (containing an access right *download*), usually via a website for which node B already has a capability (or obtains it via a search machine)[180]. Optionally it may also offer a capability for the associated code module.

The download operation is thus instigated at the accepting node, which in SPEEDOS terms (at the kernel co-module level) means that the capability is provided to the Container Manager's `download` operation as a parameter at node B. There may be further parameters (e.g. a disc capability at node B indicating where the downloaded container should be located). After carrying out appropriate checks (e.g. that the capability's generic rights and the thread security register of the current thread include a *download* right, see chapter 26) the Contain-

[178] see chapter 19 section 13.1.

[179] Of course once a container has been downloaded to a user, that user will become the owner of the downloaded copy, but that is a quite different issue.

[180] Accessing websites and search machines, etc. is a user level matter (i.e. not directly of interest to the kernel design) which is discussed in a chapter 34.

er Manager routine at node B calls its kernel's **`download`** instruction, passing the capability to it. The kernel handles this like any other kernel instruction, except that it places the capability for the container to be downloaded into the input buffer of its *download process* and issues a reschedule of the kernel processes.

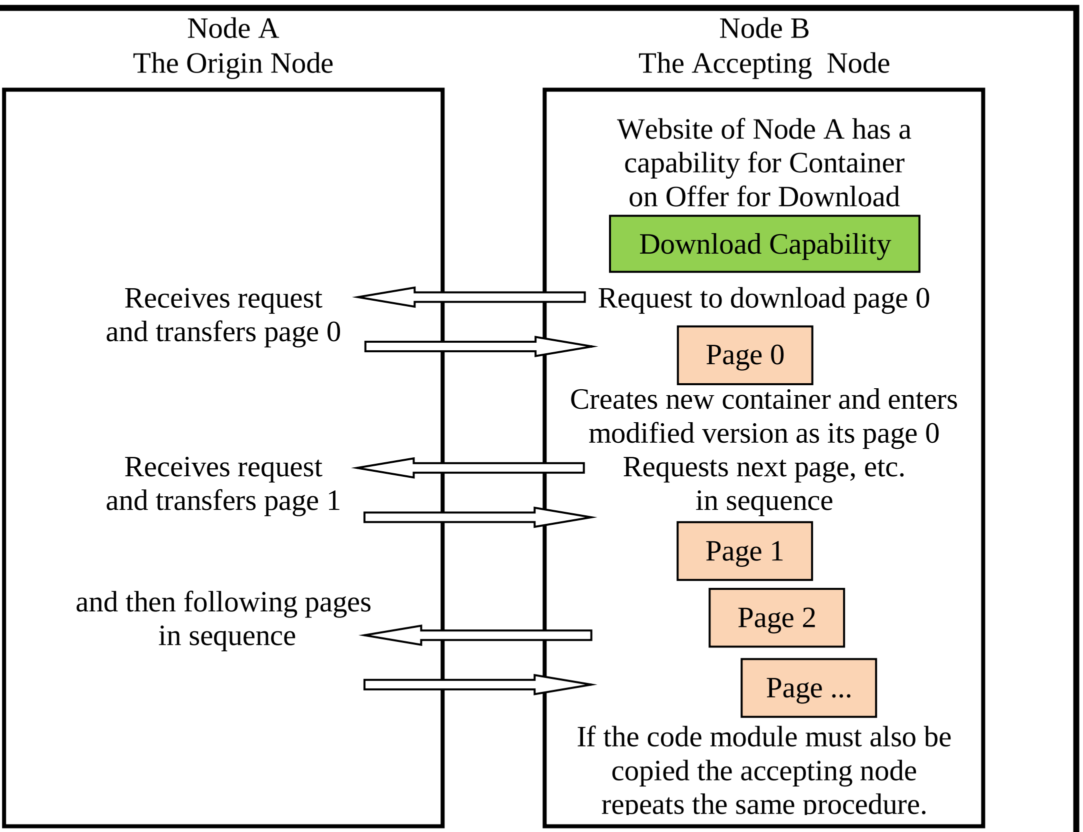

Node A
The Origin Node
Node B
The Accepting Node
Website of Node A has a capability for Container on Offer for Download
Download Capability
Receives request and transfers page 0
Request to download page 0
Page 0
Creates new container and enters modified version as its page 0
Receives request and transfers page 1
Requests next page, etc. in sequence
Page 1
Page 2
and then following pages in sequence
Page ...
If the code module must also be copied the accepting node repeats the same procedure.

Figure 29.1: An Overview of Downloading a Container

3.1.1 Downloading at the Accepting Node

Before starting the download operation the kernel at node B, the accepting node, first causes the user thread issuing the download operation to be suspended by the User Thread Scheduler (noting the thread capability for later reactivating the thread). It then starts the download procedure by sending a message to the origin node (node A), requesting a copy of page 0 of the container to be downloaded. The request contains a copy of the capability for the container to be downloaded, in which the download right is set. When page 0 arrives, the Container Manager at node B first checks whether it also has a copy of the associated code module, or if this is otherwise accessible at the downloading node. If so it creates a new container for the requesting user. It places in the new container's page 0 a copy of the page 0 which it received from node A, and modifies this by changing the identification fields to reflect the new situation. The activator of the download is regarded as the creator of the new container and the appropriate fields are modified accordingly[181]. It sets the address of the code module at node B into the copy of page 0. It also clears the qualifier entries in the co-module table as these are not downloaded. It then checks whether the code module needs to be downloaded and if so follows the same procedure to do this.

Node B then sends a request to node A to download the next page; when this arrives a free disc page is requested on the appropriate disc at node B and the page is copied into this and written to the free page; the disc location is noted in the page table. This procedure is followed for each page which arrives until the entire container has been downloaded.

When the download of the file container is completed, node B must check whether it already has a copy of the code module (which it can establish by examining the entry in the copied co-module table). If not, and if Node B has provided a capability for the software, the same procedure is repeated to download the code module, assuming that the settings in the capability allow this.

When the download has been completed the Container Manager creates capabilities for the new containers and returns them to the user who requested the download. It then reactivates the thread requesting the download.

3.1.2 Downloading at the Origin Node

When node A receives the initial request, which contains the page number required (initially 0) its download process checks that the requested container is on-line (by examining its local mount table), that the access rights in the capability allow downloading and that there are no problematic capabilities. If all is well it requests the network process to send page 0 to node B. It then exits.

Requests from node B for further pages contain the download capability and the page number and are handled in the same way. Thus node A's kernel need not concern itself with a loop and the activity is controlled entirely by node B. Not only does this simplify the task at node A but is also simplifies error handling (e.g. because node B can, for example, request the same page twice without creating a problem at node A, if a timeout indicates that a page has not been sent or received).

3.2 Uploading

This activity is similar to downloading a container, except that a copy of the container is transferred from one's own node to another Internet node. In this case a website typically offers an upload facility, which can be accepted over the website software by a user at the origin node. Thus the fundamental difference is

[181] The kernel designers might consider extending the identification fields to add information about the download source and to scan the download content for viruses.

that the roles of the two nodes are partially reversed.

The uploading node, Node A, instigates the uploading activity by passing a capability containing an upload access right to the `upload` semantic routine of its Container Manager. To do so it passes the following parameters to the Container Manager:

- a capability for the container which it wishes to upload,
- a capability for use at the destination site (e.g. a directory at the accepting site into which the newly uploaded file can be placed). (This is obtained from the website software at the origin node, and allows it to identify the purpose of the upload.)
- a message (e.g. a character string provided by the website to assist in identifying the upload container).

Normally the website software at the origin node will activate the Container Manager's upload routine, providing these parameters.

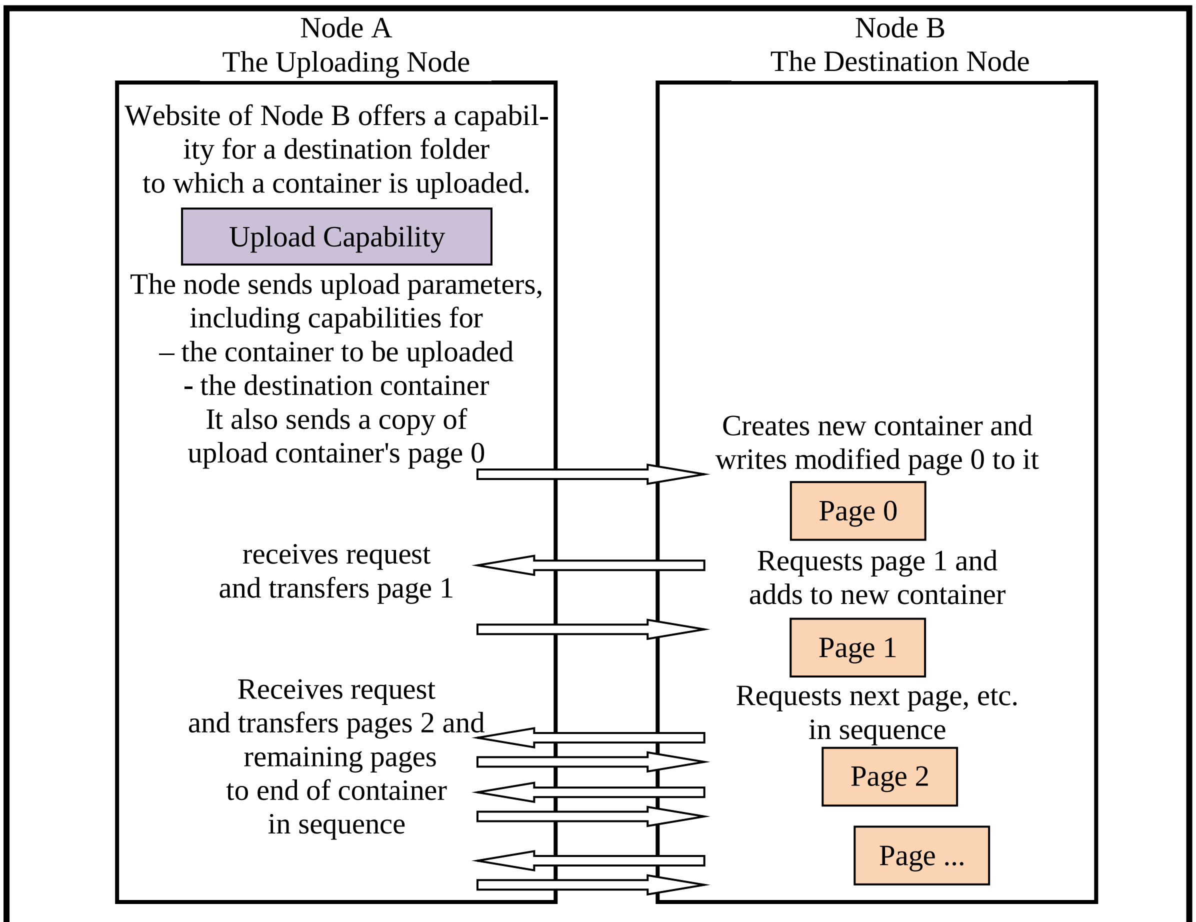

Node A
The Uploading Node
Node B
The Destination Node
Website of Node B offers a capability for a destination folder to which a container is uploaded.
Upload Capability
The node sends upload parameters, including capabilities for
– the container to be uploaded
- the destination container
It also sends a copy of upload container's page 0
Creates new container and writes modified page 0 to it
Page 0
receives request and transfers page 1
Requests page 1 and adds to new container
Page 1
Receives request and transfers pages 2 and remaining pages to end of container in sequence
Requests next page, etc. in sequence
Page 2
Page ...

Figure 29.2: An Overview of Uploading a Container

The Container Manager at the uploading node A now carries out a number of checks. It examines the first capability to ensure that the upload access rights are set. It ensures that the container to be uploaded has no "problematic" capabilities, and that the capability's generic rights and the thread security register of the current thread include an *upload* right. It then passes the parameters to the kernel's `upload` instruction.

3.2.1 The Uploading Procedure

Before starting the upload operation the kernel at node A, the requesting node, first causes the user thread issuing the upload operation to be suspended by the User Thread Scheduler (noting the thread capability for later reactivating the thread).

Node A, the uploading node, after checking that node B, the destination node, is on-line, starts the upload procedure by sending a copy of the parameters and a copy of page 0 of the container to be uploaded. The network process at node A passes this over the Internet to the upload process at node B. When this arrives, the kernel at node B creates a new container for the requesting user. It places in the new container's page 0 a copy of the page 0 which it received from node A, and modifies this, changing the identification fields to reflect the new situation. The recipient of the upload is regarded as the creator of the new container and the appropriate fields are modified accordingly.

Node B then sends a request to node A to upload the next page, and when this arrives a free disc page is requested on the appropriate disc at node B and the page is copied into this and written to the free page; the disc location is noted in the page table. This procedure is followed for each page which arrives until the entire container has been uploaded. If the code module has to be uploaded (cf. the downloading procedure) this is then carried out. On completion of the upload operation the kernel at node A then reactivates the user thread.

3.3 Encryption

As usual the encoding of information transferred over the Internet can be based on asymmetrical encryption, where the sending node uses the receiving node's public key to encrypt messages, while the receiving node uses its own private key to decode messages. If this method is considered to be too slow, then the initial request (which is encoded using the download node's public key) can include a symmetric key for use in the further exchange of pages.

Of course when a page is written to/read from disc to main memory it must be encoded or decoded using the node's own symmetric key.

3.4 Website Assistance

To simplify downloading and uploading for the end user, the kernel provides an instruction `access_container_manager`. This allows a website to obtain a capability that allows it to call the Container Manager directly, thus simplifying the work of users. The use of this kernel instruction is not limited to website use.

Any software at a node can use this instruction, provided that it has access rights allowing the following semantic routines of the Container Manager to be called: upload, download, copy, delete, rename. The security of the actions taken by these routines relies on the access right in the capabilities which are provided as parameters, not on the semantic routines themselves.

How the website software itself is uploaded to a user site will be explained in chapter 35.

Part 8
A Secure Operating System

Chapter 30
Capabilities and Directories

This chapter looks more closely at how modules and module capabilities can be organised in operating systems developed above the SPEEDOS kernel.

1 Handling Capabilities

We begin by reviewing some key aspects of what the possessor of a capability can do with it. Most of these possibilities were described in more detail in Chapter 26.

1.1 Examining Capabilities

Unprivileged programs can only directly access the contents of module capabilities by copying the capability into another capability partition or into a data partition of a segment, specifying this in the destination address operand of the kernel instruction `copy_cap`. The instruction only works if the appropriate capability metaright (see Chapter 26) allows this. If the copy destination is a data partition the capability can then be examined (and even modified) but it can no longer be used as a capability. (The format of module capabilities used by the kernel is not necessarily the same as the format which appears in the data partition of a segment.)

1.2 Creating Capabilities for New Containers and Modules

A capability for a new container or module cannot be directly created by normal modules, as a special kernel capability is required to do this. However, when a container is created and initialised by the Container Manager co-module (see Chapter 19), this returns a capability for the new container and for new modules in the container (see Chapter 23 section 6). In this sense the Container Manager acts as a bootstrapping device for other containers.

1.3 Distributing Capabilities

The possessor of a capability can execute the kernel operation `copy_cap` in order to create copies of the capability as a capability (e.g. for distribution to other users). Parameters for this instruction include a set of bit lists which are intersected with the bit lists representing the access rights in the source capability. In other words a bit has the value one set in the new capability only when the same bit is set both in the original capability *and* in the corresponding parameter list. The result is that the new capability contains only the access rights which appear both in the source capability and in the parameter list. Consequently access rights can be reduced (but not increased) using the copy capability instruction. For more details see chapter 26 section 3.3.1.

1.4 Changing the Ownership of a Container

As described in Chapter 19 section 17, the kernel provides a `change_owner` instruction, allowing the ownership of a container to be changed (after taking certain precautions). This affects the ownership of all the modules in the container. There is no mechanism for changing the ownership of individual co-modules in a container.

1.5 Restricting Capability Distribution

When a user makes a copy of a capability for use by another user, he may wish to restrict the right of that user to pass it to a third party. To achieve this, the *capability* restriction bits are set as described in chapter 26 section 3.4.2.

However, a user who has received a restricted capability is only restricted from passing it on to third parties. He may wish to store a copy of the capability into a (different) directory and/or use it in the context of several of his processes. For this reason the metarights in a capability are divided into three groups. The first group indicates whether the capability can be copied to other modules (e.g. directories) owned by the *same* owner. The second group determines whether it can be copied to modules of *other* owners, while the third group indicates whether it can be copied to users who were created at a different node.

In each of these three groups there are two parallel sets of rights. The first set indicates ongoing rights, while the parallel set indicates that the right in question can be used only once. In that way a user can, for example, provide another user with a copy of a capability while ensuring that the other user cannot distribute it further.

Each set of rights defines in more detail to which destinations (i.e. what kinds of segments, for example file segments, parameter segments) a capability can be copied. This arrangement gives a user very fine controls over how his capabilities (which give access to his files) can be used.

For more details see chapter 26 sections 3.3 and 3.4.

1.6 Deleting Capabilities

A capability may be explicitly deleted using the Container Manager's *delete capability* instruction. There is no problem with deleting a normal capability, either explicitly or implicitly (by deleting the segment or container in which it is stored). These operations have no consequences on the module to which the capability refers, except in the case of owner capabilities.

Making the deletion of objects explicit creates a potential problem: how can lost modules be deleted? A lost module is one which cannot be reached from a capability. This problem can in principle happen in two ways. The first case is when all the capabilities for a module have been deleted without the module itself being deleted. The second case occurs when capabilities still exist but have been stored in such a way that they are no longer reachable.

There is a relatively easy solution to the first problem, which corresponds more or less to what most users would want to happen, i.e. the deletion of an owner capability is regarded also as a request to delete the module. In this way there is always at least one capability – the owner capability – in existence for an existing object. (Issues such as whether the user is warned are questions for higher level software.)

The second problem, whereby capabilities exist but become unreachable, is trickier to deal with. It can happen when all the capabilities for an object end up in an unreachable circular structure. This can happen, for example, when the only capabilities for modules are placed in a lower level directory and then the only capability for this directory is deleted from a higher level directory. The MONADS systems introduced a complicated scheme in microcode to ensure that this could not happen. However that mechanism created severe overheads.

In SPEEDOS the problem is deliberately left unsolved at the kernel level. It can easily be avoided by higher level software storing at least one capability with module deletion rights (e.g. the owner capability) in a directory which is guaranteed to be reachable, because the capability for this directory is managed carefully (e.g. never deleted). In fact in many systems there is a system administrator who wants to retain a capability for each module for administrative purposes. This can easily be kept in a directory module which remains reachable. However, this solution is only a suggestion, not an integral part of the architecture.

1.7 Administering Capabilities

The kernel is designed to allow users to maintain control over their own information. However some system designers prefer to exercise a measure of control over their user community. The kernel makes no direct provision for this, leaving all such decisions to the operating system(s) above the kernel.

In a system designed to function with a "superuser", the following question arises. How can a system administrator ensure that he can obtain the object deletion right – and other rights which he probably thinks that he needs for user modules? The solution is in fact quite simple: he does not provide users with a capability containing the right to create containers and objects within them (i.e. he prevents users from calling the creation instructions of the Container Manager directly). Instead he distributes a capability or capabilities giving his users access to one of his own modules that can create objects for them (via a capability for the Container Manager). In other words the superuser reserves for himself the capability which enables him to create new containers and control similar actions.

When the superuser creates a new container for a user, he (the superuser) receives the owner capability and can retain this. This capability confers on him full rights over the module, including the object deletion right. Then he can later delete the user's module if and when this becomes necessary.

However, such a mechanism is not supplied as a compulsory part of the operating system, since not all systems have (or need) administrators.

1.8 Revoking Capabilities

The decision to base protection on capabilities confronts us with what some computer scientists see as the main drawback of the capability mechanism: the difficulty of revoking capabilities, a problem which was described in Chapter 2. Most early capability systems which allowed a flexible non-centralised, implementation of capabilities did not succeed in solving this problem[182]. However, in realistic systems it is essential that users can revoke the rights which they have bestowed on others to access their module.

A drastic way of solving this problem is to rename the module for which a capability should be revoked (see Chapter 26 section 3.2). This results in the capabilities for the original module becoming useless. It can be an expensive solution, especially if a number of capabilities have been issued to different users and the capabilities for only one of these is to be revoked. In order to restore the rights of other users, capabilities for the renamed module have to be redistributed to them. Alternatively, as described in chapter 29 section 2.2, an access control list might be maintained in page 0 at the original node of the moved module.

A flexible solution exists in SPEEDOS, using qualifiers. In chapter 13 section 10.4 the idea of "testing" bracket routines (known in Timor as "testing

[182] In fact some designers made a virtue out of the problem by arguing that the possession of a capability bestows an absolute right which cannot be revoked, e.g. [27].

methods") was introduced. This is repeated here for convenience as Figure 30.1. The basic idea is that when a module invokes another, the invocation is "caught" by a bracket routine of a qualifying module, which carries out a test. If the test fails, the bracket routine simply returns to the calling module without calling the intended target module.

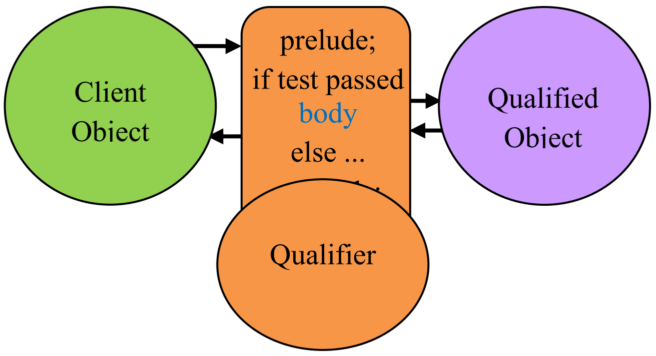


Figure 30.1: A Testing Bracket Method

To use this pattern as a revocation mechanism the owner of the qualified module defines a qualifier which has a call-in bracket (see chapter 26). He places in the persistent data of the qualifier a "revocation" list (via normal semantic routines of the qualifier) with entries consisting of some means by which the holder of a capability for the qualified module can be prevented from gaining access to the qualified module, and places a corresponding test in the bracket routine. This test might, for example, be to check the owner of the calling module (the client object in the pattern). To do this it could for example use the kernel instruction `calling_file_owner()` to test whether the owner of the calling module's file data is on the revocation list (see Figure 30.2). But there are several other tests which might be applied in a qualifier to revoke a capability. Furthermore the bracket routine might alternatively take some other action, such as causing a synchronous error interrupt and/or logging the error.

Which of these approaches is used, and how the lists are implemented, depends on the circumstances, as well as on the scope of the list. For example a situation might arise in which a user wishes to deny access to several of his files by all users except those whom he really trusts. This is best achieved by defining a qualifier with an ACL (access control list, see chapter 2 section 4.2) consisting simply of the users whom he trusts.

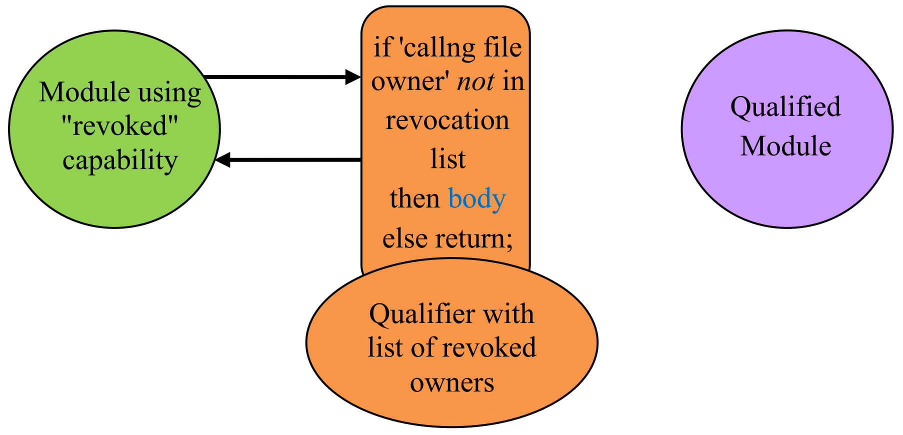

Module using "revoked" capability
if 'callng file owner' *not* in revocation list then body else return;
Qualified Module
Qualifier with list of revoked owners

Figure 30.2: Using a Revocation List in a Call-in Qualifier

Apart from ensuring that certain users who were at some stage legitimately provided with capabilities for the file are now prevented from accessing the module, such an ACL qualifier also denies access to hackers who have somehow acquired or succeeded in forging capabilities. Just a few lines of code, but a very powerful security tool!

By reversing the test in the bracket routine (i.e. by checking that the user attempting to access the file is not on the list and causing a synchronous error interrupt if he is listed) this ACL qualifier would function as a revocation list, denying listed users access to the file module.

1.9 Reducing Access Rights

Using ACL and revocation lists can have a further useful advantage. Not only can they be used to revoke capabilities, but also to reduce the access rights which a capability contains. In this case the above example is no longer adequate, since that simply views access rights in terms of the subjects attempting to access the module.

A solution tied specifically to the target module would be to maintain a list of all the users together with the access rights which they can exercise when calling the target module. Notice that the kernel's inter-module call mechanism will already have at least provisionally checked that the caller has the right to call the specific module and its designated semantic method. However by checking the associated list a qualifier could discover that although the caller has the right to call the target module the required access right for the semantic routine (which has passed the kernel's original test) might no longer be permitted to call the current semantic routine, even if it still has a right to call other semantic routines of the module.

2 Directories

A capability can be freely stored in a protected partition of any segment of any module, in a temporary heap or in a file (provided that the addressing register used has write access permission). This means that there is no prescribed method of managing capabilities. For example if a file module needs a capability for another file, it can store a capability for it in its own space and thus avoid having to go to a directory each time it needs to access the capability. This is not only efficient, but it also makes life more difficult for hackers trying to break into files if capabilities for them are not in directories.

Nevertheless users will often want to store capabilities in an orderly fashion in directories (folders). In SPEEDOS systems directories are not a special operating system feature but instead are normal file modules like other modules. Thus if a user or a system administrator chooses, he can define a private directory type with its own code implementation. This also makes it more difficult for a hacker, because he cannot assume a standard directory interface.

On the other hand there are some advantages of using standard interfaces. It then becomes easier for example for standard software, e.g. a command language interpreter (CLI) or graphics interface, to be built which searches directories to implement user commands. Some standard operating system modules are inevitable. However, it is emphasized that users have the alterative of programming (or buying) non-standard directory modules. Whereas the basic kernel should be viewed as a relatively fixed entity, its kernel co-modules are relatively flexible, and further modules comprising an operating system, such as directory modules, can be freely designed and different directories, for example, can even coexist at the same SPEEDOS system. What we now describe are merely examples of how such modules might be designed.

3 A Basic Directory

The most flexible way of storing directories involves each entry being held in a separate segment (although other organisations are possible).

3.1 A Directory Module

In its minimal form a directory module consists of a list of symbolic module names and the corresponding capabilities (together with some information about the date each entry was made, the type of module, etc.). Its interface routines include operations to create a new entry, to delete an entry, to get a capability, to reduce the access rights in a capability, to change the symbolic name in an entry, and to list a selection of (or all) the capabilities in a directory. Such a directory is illustrated in Figure 30.3.

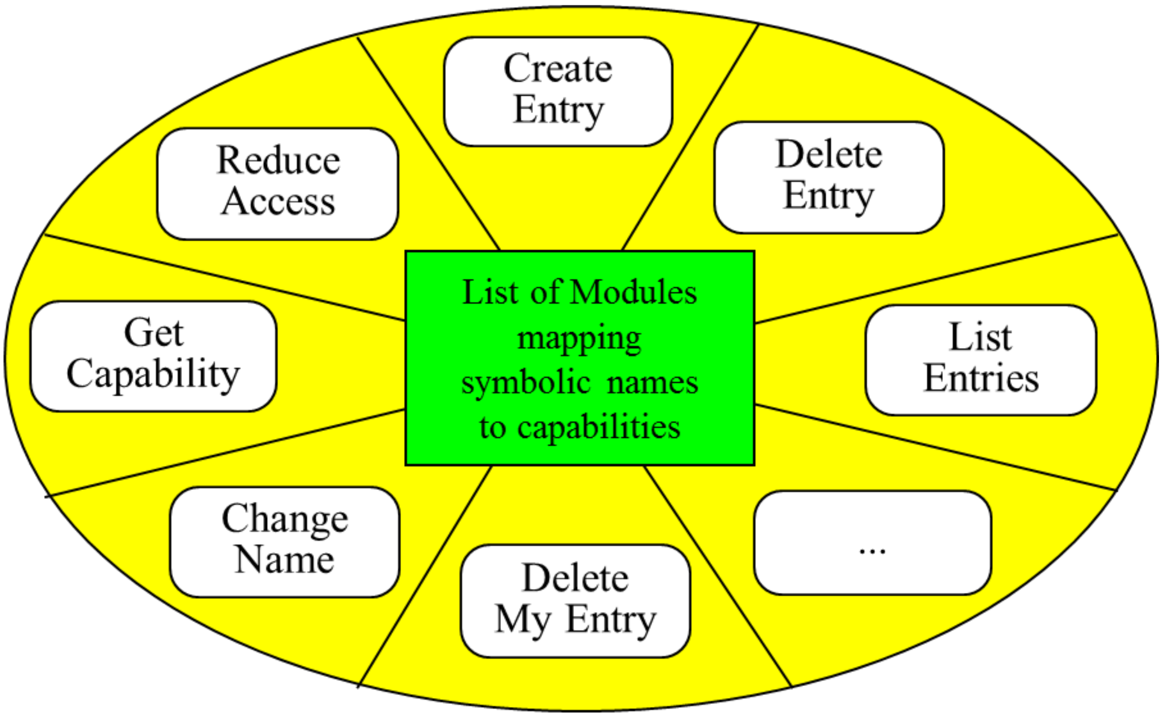


Figure 30.3: A Basic Directory Module

It is important to realise that the same module can have different symbolic names in different directories (and even in the same directory). It is similarly important that the concept of a "shortcut", found in some conventional systems, is superfluous in SPEEDOS. Where a user thinks that he needs a shortcut to provide a direct route to a file module of code module, he can simply place a copy of the appropriate capability in a directory or elsewhere, since a capability *is* a direct route to a module, and potentially there is no limit to the number of capabilities which can exist for the same module. This also has the advantage that the kinds of problems which occur with shortcuts (especially when these are placed on different discs) do not occur in SPEEDOS – unless of course a disc containing the destination module is not on-line. And it has the further advantage that capabilities (in contrast to shortcuts) contain access rights.

One semantic routine, `deleteMyEntry`, perhaps needs some explanation. Whereas the normal `deleteEntry` allows the holder of an appropriate capability to delete any entry in the directory, `deleteMyEntry` first checks that the caller created the corresponding entry before allowing the deletion to proceed. Uses for this will become apparent in later examples.

The `changeName` routine allows the caller to change the *symbolic* name of an entry in the directory. It does not affect the capability in any way. (Different directories can contain different names for entries containing capabilities for the same object.)

3.2 A Directory Entry

A directory entry might be organised as follows in the usual three partitions in a segment (see Figure 19.4). The data partition provides the user's description of the entry, e.g. the symbolic name by which the user identifies the entry, the date and time of creation, the date and time of last access, the unique identifier of the user who created the directory entry, etc.

The capability partition of the segment holds the capability providing access to the object associated with the entry (e.g. a file module, a code module, etc.). The capability might be for a directory, thus allowing the creation of hierarchical and network structured directories (see section 4 below).

The pointer partition of a directory entry can be used to link individual entries together. This can be used to allow several ways of viewing the directory. For example the first pointer in the directory might be used to provide an alphabetical view (whereby each directory entry points to the next entry in alphabetical order), the second could be used to order the entries by date and time of creation, a third could order entries by the unique identifier of users who created the entries, a fourth by the type of object to which the capability refers, etc.

3.3 Extending a Basic Directory

The directory which has just been described is very rudimentary, but from the viewpoint of this book it provides all that is needed for describing further concepts. However, it is an easy matter to extend such a basic directory structure using Timor, with more methods and/or with more information.

4 Hierarchical Directory Structures

In conventional systems a user typically has a hierarchical directory structure, which he can use to organize his files into groups, projects, programs or whatever. Because in SPEEDOS directories are just files, this is simply achieved by placing capabilities for directories in other directories.

However, this does not mean that SPEEDOS directory structures and conventional directory hierarchies are equivalent. In SPEEDOS it is normal for several capabilities for the same module to exist in different directories, often belonging to different users. These capabilities can contain different access rights and a user can only access a module if he can reach it via a capability.

But the fundamental difference between SPEEDOS and conventional systems is that each user can have his own root directory which is not necessarily reachable from the directories of others, as will be seen in the next chapter. This contrasts with conventional systems, in which there is usually a single root directory for the entire system. Thus in SPEEDOS modules are highly protected not only by the capability mechanism itself but also potentially by the fact that other users may not even be able to see the directories in which capabilities are stored. To access a directory requires access to a capability for the directory. Furthermore, capabilities need not even be stored in directories. Any file module can have capabilities stored in its segments.

Since directory modules contain capabilities which are in effect protected pointers to objects, the system cannot guarantee that directory structures are purely hierarchical, but may have more complicated network structures. Potentially this can lead the problem that some directories become unreachable and therefore that some objects cannot be deleted. However an organisational structure will be recommended in chapter 36 section 2 which solves this problem.

The next chapter describes how the directory structures of users might be organised, including how new users are introduced in a SPEEDOS system.

Chapter 31
Users and Processes

This chapter examines how users and their processes can be created in a SPEEDOS system, how a user might organise his directories, how he can log in and out and how a rudimentary mailing system might be organised. We begin by describing the creation of a process in a new container and how this might be converted into a new user.

1 Creating a New User

To create a new user, an existing user begins by creating a container for the new user (assuming that he has the right to create new users). This will serve as the latter's "root" container, and the world-wide unique identifier which it contains will also become the new user's identifier. To create this container the creating user calls the interface routine **`createContainer`** of the Container Manager, setting the parameter **`new_user`**, which then carries out the following steps.

- After carrying out various checks, it calls the kernel's `new_container` instruction (see chapter 23 section 6.1), which returns a container owner capability. The new container already contains the identification fields (see Figure 19.2).
- It sets up some basic software in the container including a Co-Module Manager and a Virtual Page Table Manager.
- It subsequently returns a capability for the container to the creating user.

This is the usual procedure for setting up a new data container, which ensures that the co-modules installed are standard co-modules.

2 Creating a New Process and its Threads

2.1 Creating the Process

The next step involves creating a process in the new root container. To do this the creating user calls the **`newProcess`** routine of the Container Manager (indicating in a parameter that this will become the root container of a new user).

The Container Manager (still operating in the creator's thread) then calls the Co-Module Manager of the new container to install a Thread Manager. It then installs a Thread Control Manager. Since these are also kernel co-modules the Container Manager is responsible for ensuring that they are correctly installed. It then uses the Thread Manager capability to call the latter's constructor (routine 0), passing to it a capability for the Thread Control Manager.

Executing in its constructor (still in the creating user's thread), the Thread Manager stores the Thread Control Manager capability in its persistent data for future use. It then sets up the stack management structure described in Chapter 20 and calls the constructor of the Thread Control Manager, which sets up its own data structures. The Container Manager then returns to the user, passing to it a capability for the Thread Manager.

At this point a new process has been created, but not yet activated.

2.2 Installing the New Process as a New User

Depending on the detailed design of the operating system, which is not determined by the kernel, the creating user (still executing in a thread of one of his own processes) might then call the User Manager module, passing to it various parameters, including a symbolic name to identify the new user (corresponding to the **`startThread`** parameter in the next section) and a copy of the new container capability. This module could, for example, determine whether the calling user has the right to create new users, determine resource quotas for the new user, etc. When the User Manager has completed setting up the details of the new user, but still executing in a thread of the creating used, returns to the creating user's thread.

2.3 Creating the Initial Thread for the New User

Executing in one of its own threads the creating user calls the **`createThread`** interface of the Thread Manager in the new process to create the new user's first thread, passing the following parameters:

– a capability for the module (**`startMod`**) in which the first thread should start executing;

– an integer **`startRoutine`** which indicates the semantic routine number of **`startMod`** that will first be invoked;

– a string parameter **`startThread`** providing a symbolic name for the first thread[183]; and

– a capability **`rootMod`** for what the first thread will regard as its root module.

Using this information the Thread Manager sets up a stack for the first thread such that it is ready to make the first inter-module call (to **`startMod`** at the routine **`startRoutine`** with a parameter segment containing the capability **`root Mod`**). It then advises the Thread Control Manager of this thread and the latter then sets up a thread state and calls the global User Thread Scheduler to schedule the thread when a CPU becomes free. Notice that up to this point all the activity has been carried out in a thread of the creating use, which might now return from the Thread Control Manager and go about its further business.

What happens next in the new thread depends on what the **`startRoutine`** of **`startMod`** is programmed to do. Some possibilities are discussed shortly.

2.4 Creating Further Threads

A user can create further threads for his process as described by repeating the steps described in section 2.3 (using his own thread). These can be created from another thread in the same process or by a thread of another of his processes.

2.5 Creating Subthreads

As was explained in Chapter 20 sections 5 and 8, the code of an application module called within a thread can create subthreads (subject to the restrictions in the thread security register, described in Chapter 26). It does so by executing the kernel instruction `get_subthread_cap` to obtain a Thread Manager capability and uses this to call the **`createSubthread`** interface of the Thread Manager, passing as a parameter the entry point number in the Subthread Entry Point List (see chapter 19 section 9.5), which determines where the subthread should start executing. Hence the **`createSubthread`** interface of the Thread Manager has a parameter which indicates the internal routine number in the list. It then obtains the identifier of the calling module (i.e. the module in which the subthread should start executing) by executing the kernel's `calling_file` instruction (see chapter 26 section 1). It then creates a new thread, sets up the initial register values for the thread and places a 'subthread' backstop on the new stack (which allows the kernel to carry out the final `return` correctly). The Thread Manager then calls the **`newSubthread`** interface of the Thread Control Manager, passing to it a thread capability for the new subthread, the identifier of the start module and the number in the Subthread EPL. This updates its own data structures then

[183] This serves as the user prefix described in chapter 22 section 11. It is passed to the Log-in Service Module, which checks that it is unique within the current node; if not it returns an error message to the Thread Manager; this then advises the user to supply a different symbolic name.

calls the User Thread Scheduler's **`newSubthread`** interface, passing on the parameters; this notes the new thread details and when the new subthread can be scheduled it calls the kernel's `new_subthread` interface (see chapter 20 section 8.2), which eventually activates the new subthread (see chapter 19 section 5).

2.6 Passing Parameters to Subthreads

The above scheme provides no mechanism for passing parameters to subthreads. However this can be organised in the code of the module in which a subthread is activated. For example the state data of the module could include a table which is protected by a reader-writer semaphore. In this table there could be an entry for each subthread, indexable by its subthread number. The subthread number itself can be obtained by the thread requesting its own thread capability.

Similarly subthreads can communicate with each other (and with their creating thread) via semaphores. They could thus advise their creator of their impending deletion, or co-operate on a common task.

2.7 Deleting Threads and Subthreads

Threads and subthreads are deleted when the kernel executes their final `return` instructions. The kernel recognises this condition from the corresponding backstop condition on the subthread's stack. In both cases the kernel then creates a surrogate thread in which the Thread Manager is activated in a **`deleteThread`**/**`delete Subthread`** interface routine (to which it passes the thread number of the thread to be deleted). This can carry out appropriate final activities such as deleting its data structures; it then calls the Thread Control Manager's **`deleteThread`**/**`delete Subthread`** interface routine which does likewise and then calls the User Thread Scheduler's **`killMe`** routine, which removes it from its scheduling data and finally executes the kernel's `switch_delete` instruction (cf. chapter 22 section 1).

3 The Initial Capabilities of a New User

As was described above, when a new thread is created one of the parameters which it passes to the **`startMod`** module is a capability (**`rootMod`**) which is typically for a directory.

3.1 Root Modules which are not Directories

If the thread's root module is *not* for a directory module then the thread has very little scope, except simply to execute **`startMod`**. This need not be a standard command language interpreter but can be an application module. This scenario might for example be useful in a public library setting, where the public is permitted to access the library catalogue online. Such a process/thread will have no capabilities except those embedded in its own segments. These can of course be exchanged or added to by other library threads (which are only accessible to library staff) invoking different interfaces of the module.

3.2 Root Modules which are Directories

In many cases the **`rootMod`** will be a directory which holds capabilities of further directories. Neither the kernel nor the operating system defines what these should be, and their content will in part depend on the nature and purpose of the operating system. What now follows is an example of how the directory might be organised for a general purpose multi-user environment.

3.2.1 Access to Public Software

In such an environment some of the threads of most processes are likely to need access to a variety of publicly available software (both code modules and data modules). This could include standard utilities (e.g. text editors, spelling checkers, dictionaries, etc.) and also system software (e.g. compilers, libraries, etc.). Such software can be made reachable to the new user via a capability in **`rootMod`** which leads to further directories. This capability is referred to as the user's *public* capability.

3.2.2 Access to Capabilities shared with the Creator

The creator of the process (who might or might not be its owner) might chose to share some of his own software and files from other processes which he owns. The directory via which these can be reached is accessible via a *creator* capability in **`rootMod`**.

There are a number of capabilities which the creating user holds that can be viewed (depending on the security model) as belonging to a new user. These include capabilities for the Thread Manager, the Thread Control Manager, the Co-Module Manager, the container and the initial thread capability. All these capabilities can be returned to the creating user via the module addressed by the creator capability.

Whether the creating user retains a copy of these capabilities is a further system design decision. If the new user wants to prevent the creating user from secretly accessing the latter via a secret copy of the capability, he has at least two possibilities. He can place revocation brackets around the module (if the owner capability has been passed to him), or more simply – he can copy the contents of **`rootMod`** to a new module which he creates and remove these capabilities from the old module.

3.2.3 Private Modules

Then finally the owner of the thread might want to create a root directory for his own personal files and programs. When he has created this directory (his *private* *root*) the capability for this could also be placed in **`rootMod`**, but might simply be stored in a private root directory which is not reachable by the creator.

3.2.4 Simpler Cases

Notice that even in the envisaged environment some threads will not need access to a root directory containing all of these items. For example, if a process consists of a number of cooperating threads where most are active in a particular module simply as a means of providing parallel processing, they might need access only to the module in question and perhaps to a further module which has been designed to co-ordinate their cooperation.

4 Different Kinds of Processes and Threads

4.1 Interactive Threads

A process can, but need not, have some interactive threads. For such threads the **`rootMod`** capability will provide access to an authentication module (for checking the authenticity of a user when he logs in) and some form of command language interpreter (which may of course use graphics to allow a user to select the commands which he wishes to invoke) in addition to capabilities for the commands themselves.

If the **`startMod`** for such a thread is the command language interpreter, this will normally carry out its own initialisation then call the logout module. The latter will long suspend until the owner of the thread logs in. The logout module can be a module of the user's choice which freely uses any criteria to establish his identity (see Chapter 22 section 11).

When a user logs in for the first time, he will typically use an authentication module provided by his creating user (and passed as a parameter by that user to his new Thread Control Manager) and will have to be informed by that user how he can pass the authentication test. But thereafter he can use an authentication module of his own choice, after passing a capability for this as a parameter to his Thread Control Manager. In order to minimise their security risk, users can use a different authentication module for each of their interactive threads, so that if a hacker succeeds in breaking into one thread, this does not help him to break into other threads of the same user.

4.2 Multiple Processes

A single user can have multiple processes, which might be dedicated to different activities or different projects, each with multiple threads. SPEEDOS gives him the opportunity to maximise the security of his individual processes/threads, simply by following the "need to know" rule (in this case providing each thread only with the capabilities which it needs).

And finally, it is even possible to give a new interactive thread access to only one module, without an authentication module. This could be suitable, for example in a public library system which allows users unchecked read-only access to a library catalogue.

5 Communication between Processes

In the SPEEDOS design the only way in which two threads can communicate with each other is via shared file data, but in order to gain access to a shared file, each user needs to have a capability for the file with appropriate access rights. How can these capabilities be distributed? Suppose for example that a user has created a message for another user in a file (his "email" message). How can he send a capability for this to him?

Section 3.2.1 above describes how a new user can gain access to standard capabilities when he is introduced to the system. What is now needed is a general mechanism which allows users to pass capabilities to each other and thus communicate in a general way. In principle no further software is required to achieve this. The solution is to use directories, and the trick is to set them up in the right way.

5.1 Sending Capabilities

Suppose that there is a capability reachable via the user's *public* directory (see section 3.2.1 above) – or a directory which can be reached from this, which we call the *public mailbox directory* (PMD). Like other directories it contains a list of symbolic names and capabilities corresponding to these names (see Figure 31.1). The capabilities in this directory are for the *private mailbox* directories of the named users. So if a user A1 wants to send a capability to another user A3, he calls the `getCapability` routine of the appropriate PMD, passing the name "A3" as a parameter, and back comes a capability for Y's private mailbox.

A3's private mailbox is a directory module, so to deposit the desired capability into it, A1 simply calls the `createEntry` interface of A3's private mailbox directory, passing the capability and giving it a symbolic name (e.g. "A1.mail"). A3 can use the `listEntries` interface of his mailbox directory to see who has sent him messages in this way, and he can call the `getCapability` interface to get the capability for A1's file. Then he can access the file like any other. Since the unique identifier of the user creating the entry is not supplied by the user, but instead is obtained by the directory software from the kernel, it is guaranteed (subject to the correctness of the directory implementation) that the user identifier is correct.

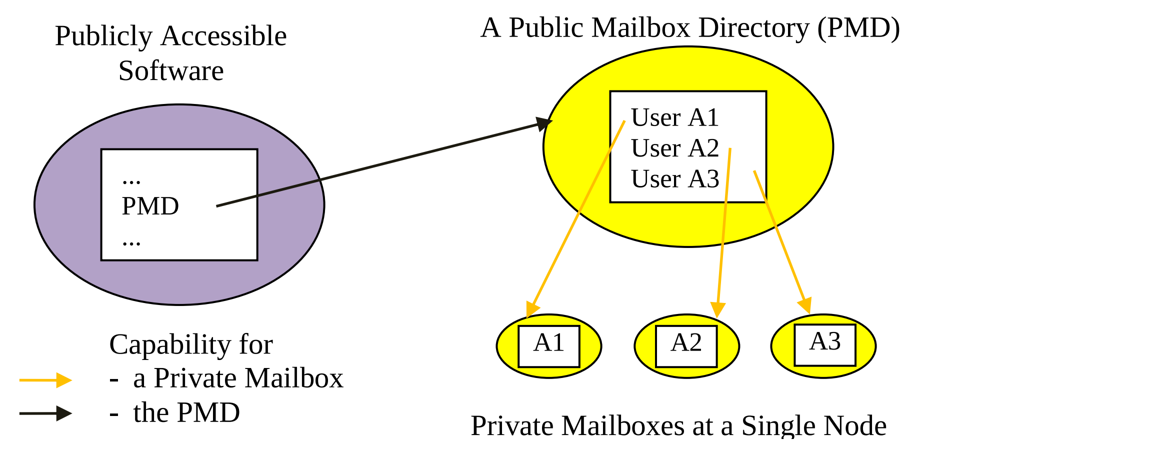

Publicly Accessible Software
...
PMD
...
A Public Mailbox Directory (PMD)
User A1
User A2
User A3
A1
A2
A3
Capability for
- a Private Mailbox
- the PMD
Private Mailboxes at a Single Node

Figure 31.1: A Simple Mail System for a Single Node

Interestingly, the aim at the start of this section was to distribute a capability to another user, but what has also been achieved (almost accidentally) is the design of a rudimentary local email system (as the names which were given to the various directories hinted at). Assuming that the capability is for a text file, the text is almost equivalent to a conventional email.

Apart from the fact that normal directories have been used, there is one other substantial difference from a normal email system, i.e. the capability for the message is copied, but not the "email" itself. This means that the receiver can decide whether he wants a copy (assuming that the access rights in the capability allow this) and consequently ensure that his "mailbox" is not cluttered up by rubbish sent by others.

5.2 Receiving Capabilities

It has been shown how A1 can send mail to others, but how can they send mail back to A1, a new user? First, A1 has to create a new mailbox directory for himself, make a copy of its capability with reduced access rights and then insert this into the PMD. To create his own mailbox he will need a capability which gives him access to a constructor for a directory module, which he can expect to find in the public directories available to him.

When he has created the mailbox he will need to insert a capability for it into the PMD. This means that the capability which he acquires from the public directory system for the PMD must not only give him `getCapability` access but also an access right for `createEntry`.

This raises the following question. If A1 can create an entry, shouldn't he also be allowed to delete it if he decides that he doesn't want to receive any more mail? Giving him access to the `deleteEntry` interface has the consequence that he can mischievously delete other entries. This is one reason why the special

`deleteMyEntry` routine was included in the semantic routines of directories (see Figure 30.3). It will be recalled that this allows a user to delete only those entries in a directory which he created.

So if A1 wishes to delete the entry which he has made in the PMD, he can do so by calling **`deleteMyEntry`**, which checks that the calling thread has the same user identifier as that recorded in the entry to be deleted. So in fact X needs three access rights for the PMD: **`getCapability`**, **`createEntry`** and **`deleteMyEntry`**. But he does not have an access right for the more general routine **`deleteEntry`**, which does not carry out this check. (In a system controlled by a system administrator, the latter (who created the PMD in the first place) will probably keep this interface restricted for his own use, to delete entries for users who deregister from the system.)

5.3 Deleting Capabilities Already Placed in a Foreign Directory

It turns out that **`deleteMyEntry`** is not really a special interface for this purpose, but is in fact generally useful. If it is available with individual mailboxes, for example, it will allow A1 to delete mail which he has deposited in another mailbox and then decided to revoke (e.g. because he realises that he has sent some wrong information), without allowing him to interfere with mail from other users which happens to be still in the mailbox.

Who has the right to delete the email message itself? That depends on the sender. He can send a capability with or without this right. He might even pass an owner capability.

5.4 Receiving a Copy of the Capability's Content

In conventional mail systems the recipient receives a copy of an email in his own file area. To achieve this in SPEEDOS, no further mechanism is required. Since at present we are only considering messages sent within a single system, the recipient will already have a capability for the Container Manager at that node. Hence assuming that the sender has set a copy access right in the capability the recipient can call the Container Manager to copy the module and store it in his own space.

The recipient thus has the choice of copying the content or not, which he would obviously not be necessary, for example, if he recognises that the mail is spam. (This would also make life more difficult for spammers!) Of course, if a user does not copy the content of the mail there is always the risk that he will later not be able to access it, if the sender decides to delete it.

6 Is the Communication Secure?

So without any extra software whatsoever, the basic directory module has provided us with a local mail system. But is it a secure system? What is to stop each user from reading the mail in other mailboxes, for example? The answer is simple: the access rights in the various capabilities can be restricted to doing the "right thing". For example the capability which user A1 receives from the PMD for a private mailbox can restrict him to inserting entries – he cannot use interfaces such as **`listEntries`** or **`getCapability`**; only the owner of the private mailbox can do that. It is of course the receiver's choice to decide whether he allows the sender access to the semantic routine **`deleteMyEntry`**.

What is to stop A1 from being mischievous and deleting other users' mailbox entries from the PMD? In this case A1 only has a right to call the **`get Capability`** and the **`listEntries`** interface routines, and possibly **`delete MyEntry`**. The access rights in his capability restrict him from calling **`delete Entry`**, **`changeName`** and other mischievous possibilities.

7 Mutually Suspicious Users

Including the unique identifier of the creator/modifier of each entry in directories has another security advantage. It allows mutually suspicious users to be sure about the sender and the receiver of a message, provided that they each already know the unique user identifier of the other (which need not be kept a secret).

A receiver can determine the unique identity of the sender, because this has been recorded in his mailbox entry. Assuming that the code of the directory module is correct, the sender's identity can be regarded as reliable, because it is supplied by the kernel.

But how can the sender be sure about the identity of the recipient, i.e. how can he be sure that the capability that he acquires from the PMD is for the person he intends to send mail to? This too is no problem, since the information is stored with the entry in the PMD.

Hence users of a SPEEDOS system can have far more confidence than in conventional systems about the identity of the sender or the receiver of messages[184].

It is in principle possible that a trojan horse could exist in the directory software, and this might change the unique identifiers. However, this could be checked by using a special bracket routine designed for the purpose. This intercepts calls to the create entry routine and logs the parameters and the caller's

[184] As was mentioned in Chapter 4, there are methods involving encryption which aim to dispel doubts about the identities of senders and receivers of messages, but these are by no means simple and are not used on a routine basis, in contrast with the mechanism just described.

identifier (which it obtains from the kernel) in a separate log file together with the date and time of each call. By comparing this file with the arrival of mail as reported by the directory module, it would be possible to ascertain that all mail is being recorded and that the sender is being correctly recorded.

Such a check would not have to be carried out manually; it could be automatically built into a more sophisticated mail system, as could many of the operations which we have explained above. For example when a user process is created the User Manager module which first executes in his process could create for him a private mailbox, make an entry for him in the PMD etc.

8 Further Mail Refinements

What we have so far described is a rudimentary email system, which works only within a single computer system. In the next section we shall consider how it might be extended to cover mail from users working at other computers. But before doing this let us consider how it might be used locally to serve users better.

You probably get annoyed with the amount of junk email which arrives in your mailbox. What can be done about this? A nice simple solution is to have *no* entry for yourself in the PMD at all (or remove it if the User Manager has put it there for you). Then you will not receive any email at all. This is perhaps a bit drastic. You might prefer to receive email from just a few special contacts. One way to ensure this is to create some private mailboxes and send capabilities for them to your contacts, and delete your original mailbox. Another way would be to bracket your mailbox with a qualifier which uses an access control list to determine from which senders you wish to receive mail. This is equivalent to using filters in conventional systems, but can be far more flexibly programmed, and much more reliable with respect to the identification of senders.

A system administrator also gets some powerful tools to control mail. For example if some user is not entitled to send and/or to receive mail at all – which is another possible approach for confining information – then the system administrator can create a directory system for users which does not include the right to create new directories. Then the user cannot create a mailbox. Or the new user does not receive a capability allowing him to make an entry in the PMD. Or it is possible to develop a more centralized mail system in which all mail has to be transferred through a central mailbox and then distributed further. This central mailbox could be bracketed by ACLs to determine which users can send or receive mail, or it might be a special module which even monitors the mail itself.

There are endless possibilities for improving or changing the basic mail system which we have described. Some improvements can serve to make the system more convenient for users, while in a draconian mandatory system very tight controls can be exercised.

9 Distributed Email and File Systems

Having seen how a basic email system might be implemented in a stand-alone SPEEDOS system, we can now consider how this can be extended to function across all the SPEEDOS nodes in a system.

9.1 A Distributed Email System Using Remote Inter-Module Calls

Figure 31.2 shows separate mail systems at three nodes based on the design described above. The only problem is that they are not connected with each other. The aim of this section is to show how they can be linked together.

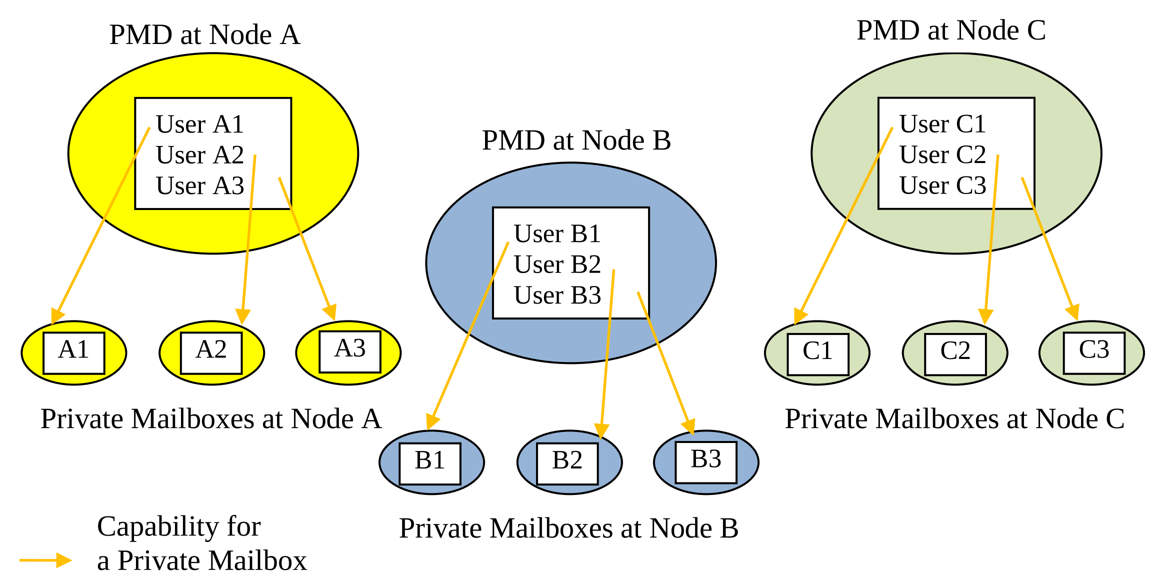


Figure 31.2: Three Unconnected Single Node Mail Systems

The solution is remarkably simple. In chapter 28 section 8 we described how the kernel's network process maintains and uses a network address table (NAT) which has an entry for each foreign SPEEDOS node for which the current node has contact details and how it has access to a world-wide network of 'directory nodes' in order to obtain similar information about nodes with which it has not yet obtained contact details.

One of the items in each NAT entry is a 'shared capability' the purpose of which is to provide a starting point for communicating with *user level software* at nodes listed in the NAT. The kernel design does not define more specifically what the capability actually addresses and at this stage I would not like to rigorously define its particular purpose. But what we can say is that the capability is either for the PMD at that node, or is a capability which leads to a directory, within the structure of which a capability exists for the PMD at that node. In other words the public mail directory at the node is reachable via this capability. Exactly how, is a design decision of the operating system above the kernel. And if the PMD for a node can be reached, so also can the mail boxes of those users at the node who wish to be reached. Hence the problem of linking the nodes for email purposes is solved. Notice that just as in current systems one needs to know a person's email address in order to send mail, in SPEEDOS one needs to know his node and his local name at the node.

9.2 Retrieving Emails from other Nodes

I pointed out in section 5.4 that a full email in the local email system can be retrieved simply by using the *copy* routine of the local Container Manager. This is not the case if the email is held on a different node from the recipient's node. But that is scarcely a problem. Instead of using the Container Manager's copy facility one simply uses the *download* routine of the Container Manager (see chapter 28 section 10.1). In this case the *download* access right must be set in the capability. The difference between a local copy operation and a download operation could easily be hidden from the user in a more sophisticated version of the email system described above.

9.3 An Advantage of the Above Design

I emphasize that the design as presented if fairly rudimentary and does not contain all the facilities found in current systems. But that is not a shortcoming. Individual designers can extend the design to make it more user-friendly. But I would like to emphasize that I would be very sad if it turns out that some designers were to make their email systems as non-transparent as they have designed current email systems. In particular they should not turn the email system into a world of its own, which, for example, completely hides the operating system directories. My reason for this is that emails thus become a "special" system. Emails are rarely isolated in practice from other work of users, yet to store an email in a normal directory along with other files as part of a project is currently very difficult. As I have defined them, emails ARE just files like any other and should be treated as such. Apart from the greater convenience for users, large amounts of software in current email systems would simply become redundant.

Chapter 32 Command Languages, Name Management and Graphical Interfaces

So far nothing has been said about the interface between the operating system and its users. The purpose of this chapter is to fill that gap, at least in broad outline. The first issue is that users need a way to communicate with the system. The first section briefly describes the ad hoc command languages which were developed in the early days of computing to activate user programs, followed by a more systematic command language approach based on a unified technique for invoking operating system operations and user programs.

The second aspect of the user interface issue is that, as so far described, low level names (typically consisting of integers) have been used when describing various objects (e.g. node numbers, module numbers). Because of its efficiency advantages this technique is appropriate in addressing the kernel and low level operating system features, but it would be dehumanising to expect normal users to work with such numerical names[185]. Humans expect to be able to use symbolic names, such a 'MyFile' or 'Temp'. The consequence is that an operating system needs to be able to translate between the two, and for this reason it needs a name management system. The second and third sections briefly describe how such a system might be organised for a SPEEDOS operating system.

Text-based command languages have largely been superseded in more modern operating systems by a user-friendly approach based on the extensive use of graphics and pointer devices. Suggestions for how this approach might be implemented in SPEEDOS are made in the fourth section.

185 cf . chapter 2 section 2.1.

1 Command Languages

We begin by briefly describing the early development of ad hoc command languages. We then consider a more systematic approach.

1.1 Ad Hoc Commands

In early systems computer card readers, paper tape readers and/or simple telex devices were used to communicate with the operating system. Individual commands were devised to allow users to activate operating system features and the programs which they had developed. Gradually standardised syntaxes arose, usually consisting of characters (such as //) which allowed the computer to recognise that what follows is to be understood as a command for the operating system to execute. Then followed the name of the operating system command, e.g. // copy. This was in turn followed by further information needed to indicate what the command had to achieve, e.g.

```
// copy MyFile NewFile
```

which might mean that a file called 'MyFile' should be copied and the resulting copy should be called 'NewFile'.

That was sufficient to invoke standard operating system commands, but users also wanted to start application programs which were not part of the operating system. It seemed obvious in those early days that this should be achieved by having a further operating system command which could start user programs, e.g.

```
// start MyProg
```

If 'MyProg' needed further information then the program itself (after it had started) had to read this information into its memory. The program would usually expect to find this information on further cards or at the following paper tape positions or on a further line typed into the telex device, e.g.

```
// start PayrollProg
file = EmployeeFile, printer = Printer1
```

In this example the format of the second line is fully determined by the individual program, which could have chosen a quite different format and quite different separators, e.g.

```
// start PayrollProg
EmployeeFile; Printer1
```

or it might have been defined by the program that different items should be placed on different lines, e.g.

```
// start PayrollProg
EmployeeFile
Printer1
```

The consequence of this is that users had to learn (from information supplied with each program which they used) how the input was formatted, which could differ considerably from program to program, even for different programs which achieve the same thing.

In this case the file name and the printer name are best understood as input parameters for the program 'PayrollProg', just as 'MyFile' and 'NewFile' are input parameters for the operating system command 'Copy'. Hence it is thinkable that the problem of different formats could have been solved by using the same syntax for invoking user programs and for calling operating system commands. In other words instead of writing

```
// start PayrollProg
file = EmployeeFile, printer = Printer1
```

or

```
// start PayrollProg
EmployeeFile; Printer1
```

it could have been written (using the same format that was used above for the copy command)

```
// start PayrollProg EmployeeFile Printer1
```

This would have regularised command formats (and considerably reduced the amount of learning required to call different programs).

The problem in conventional systems was (and still is) that user level programs are defined by programming languages and by operating systems in such a way that they cannot have standardised parameters. Consequently in these systems there was and still is a shortcoming that user program interfaces cannot be regularised. Even with modern graphical interfaces, the interfaces for calling programs cannot be standardised. The result is that programs still today must create extra graphical interfaces (which vary from each other) to obtain their parameters.

1.2 The SPEEDOS Solution

SPEEDOS, following the MONADS approach [26], could in principle overcome this problem in that operating system modules and user modules are uniformly structured according to the information-hiding principle[186]. One consequence of this is that their semantic interface routines can in appropriate cases be regarded as commands which in principle can be invoked using the module name and the name of one of its semantic routines, followed by the routine's parameters. The preferred syntax for this is rather like an object-oriented method call, e.g.

```
containerMan.copy(MyFile, NewFile);
```

or

[186] see chapters 13 and 14.

```
 Payroll.calculate(EmployeeFile, Printer1);
```

However the actual syntax is not important, and could in SPEEDOS be determined by the programmer of a command language interpreter.

1.3 SPEEDOS Command Language Interpreters

In the SPEEDOS environment a command language interpreter (CLI) is not a special module nor does it have special privileges. It is simply a module which is designed to invoke other modules, normally within the persistent thread in which it is executing, though it may of course create and activate subthreads.

In order to function correctly it needs access to a directory module of the user who owns the thread in which it is executing. This directory module serves as a starting point for the CLI to find the commands which it is required to execute, i.e. the modules and their semantic routines to be called. As we saw in chapter 30 directories can contain capabilities for other directories so that a hierarchical (or network) directory structure can be created.

Chapter 31 section 3.2 described how each user has a private directory structure which is initially set up when a new user is introduced. The user is represented by a persistent process and its initial thread starts the first activity of the user. Except in special cases the first module which the user will normally call is a SPEEDOS CLI. This can gain access to the standard input and output modules (via the mechanism described in chapter 19 section 5) which allow it to communicate with the user. From there on it can receive commands (including commands which allow the user to move from one directory to another) based on the modules and their semantic routines, as described above.

But there is one element missing in this description. The modules and their semantic routines which the user finds in his directories use numbers as names, yet the normal user wants to be able to work with commands which use symbolic names. We now consider how this gap can be bridged.

2 Translating Numbers into Symbolic Names

At the low level, executing a command (i.e. calling a semantic routine of a module) consists of a module capability, the number of a semantic routine and parameter values for the semantic routine. We consider these in turn.

2.1 The Module Capability

Symbolic names associated with a capability are normally held in directories which hold a potentially large number of details relating to the capability, including a symbolic name which was chosen by the user when he entered the capability into the directory. This is the name which the user will use when he wants to call a routine of the module addressed by the capability. In this sense there is no problem in associating symbolic names with capabilities. Nevertheless several users can have separate capabilities for the same module, and they can provide different symbolic names for the same module[187]. Each directory will have a semantic routine which takes as one of its input parameters the symbolic name of a capability and which returns a copy of the capability. This routine can be used by a CLI to translate the module name part of a command and thus obtain a capability. Hence the first part of the translation is solved in a very straightforward manner.

Of course the user might first want to obtain a list of the directory entries, in order to decide which entry is relevant for him. This too can be obtained in exactly the same way by the CLI, using a semantic routine of the directory module which lists its entries. And in a similar manner the user can use the CLI to make new entries, delete entries, etc. in a directory for which it has access.

But this raises a further question. How can the CLI, and more generally users who know the symbolic name of a module entry, discover the symbolic names which the user can use for the *semantic routines* of modules in a directory?

2.2 The Names of Semantic Routines

Normally, semantic routines have symbolic names. These are given to them in programs created by high level programming languages. Consequently it is reasonable to expect that when a compiler compiles a module it can also create a "partner" module which lists the names and maps them onto the numbers by which they are known at the kernel level. If this list is available to a CLI then it is in a position to select the appropriate routine number when the user issues a command in which it provides not only a module name but also a semantic routine name.

The question then arises where this list should be kept. If it were held with or integrated into the compiled version of the module (e.g. as symbolic names in entry point lists), it would not be easily accessible to CLIs, and it would be awkward and not very flexible to expect that inter-module calls could pass symbolic routine names.

A better alternative is that the compiler should produce a separate module, which we call a *template manager*, containing the mapping between routine names and entry point numbers. However, it is not immediately obvious how the CLI gains access to a capability for the template manager associated with a particular module to be called, since in many cases the module to be called is a file

[187] This possibility of having different symbolic names for the same module can of course be used to confuse hackers.

module, the code for which is implicitly (rather than explicitly) associated with the module's data container. From the viewpoint of the CLI the most convenient place to store a capability for a module would be in the same directory entry as it obtains the capability for the module to be called. But how does it get there?

When a directory entry is created, one of its parameters indicates whether the module can be used as a command for a CLI. If so the directory module executes the kernel instruction `getTemplate`, passing to it the capability for the module to be called. This returns a read-only capability for the template manager (which it obtains by examining the Co-Module Table for the module, see chapter 19 section 7). The directory module then places this capability in a field of the entry which it is creating, so making it accessible to the CLI. Since the information is not security sensitive and the capability provides only read access, this instruction is unprivileged.

2.3 Passing Parameters to Semantic Routines

In addition to listing symbolic names for the semantic routines of a module, a template manager also lists the names and types of the parameters for each routine, together with further information such as the errors which it can cause. This information is useful to a CLI not only in terms of error checking but it can also be used for example to prompt a user by providing the names of the parameters, using these as keywords.

A template manager might also contain further information which could be helpful for the system software or for the user, e.g. a symbolic name for the code module (e.g. 'PDF').

2.4 Alternative Template Managers

A template manager is an information-hiding module like any other, so that not only compilers can use its routines to create templates for the CLI. Users who would prefer to use other names than those which were taken from the source code of modules can thus use its routines to create their own templates for the CLI. In order to use such templates the user simply needs to use the directory manager routines to add or overwrite the field in the appropriate directory entry.

2.5 The CLI as a Module Tester

When a programmer writes any new program he will of course want to test the correctness of his code. A CLI as described above provides a useful basis for doing this, by allowing him to call the individual routines of the module and check the results.

3 Other Naming Modules

The user normally wants to use symbolic names not only for command language interpretation but also for other purposes where the kernel uses numbers as names. One example of this which we have seen already is the names of users themselves, when they need to identify themselves and their persistent threads/processes as part of the procedure for logging out and in (see chapter 22 section 11). Where appropriate the "Current Login Name" shown in Figure 22.3 could also be used as a thread/process name.

4 Graphical User Interfaces

Users today expect to use graphical interfaces rather than old-fashioned CLIs. However, I have deliberately emphasised the "classical" interface style as found in most systems before graphical devices became popular. There are three reasons for this.

First, operating systems which support graphical interfaces should not be seen as an alternative to the classical style, but rather as an extension. The basic functions of the kernel and an operating system remain substantially the same regardless of the interface which is offered to users. It is therefore important for a kernel (and an operating system) to provide a clear and secure basic design independent of the style of the user interface.

Second, it is important that the basic design and implementation of the system are thoroughly tested before the complexities of a graphical interface are added. This can best be achieved by testing the kernel, its security sensitive co-modules and related service modules (such as the spooling system which will be presented in the next chapter) without being concerned about the final user interface. This can be achieved by using a CLI designed along the lines which we have outlined earlier in the chapter. I thorough recommend that path is followed when a SPEEDOS system is built.

Third, I also recommend that certain systems, which can easily dispense with the complexity and the additional risks which a graphical interface represents for security, should do so. Here I have in mind mainly process control systems such as those used in power stations, in aircraft, in weapons systems, in automobiles and some (but not all) hospital applications, to name a just few, as most of these can easily be driven by a CLI style interface. Furthermore the consequences of security mishaps can be very serious indeed in such systems. Graphical interfaces are nice but the complexity of the software to drive them may put such systems unnecessarily at risk.

I must confess that I have little experience with graphical design, and I have little time to learn in detail about such systems[188]. Nevertheless I will now offer some comments on how the basic SPEEDOS architecture might potentially be enhanced by a graphical interface.

4.1 The Graphical Devices

I briefly consider three graphics-related devices: the visual display screen(s), the mouse and the keyboard. I regard these as a related set of devices which together provide the graphical interface. However, I do not discuss the hardware and software of these devices as such, since there is an overwhelming number of approaches and techniques. I simply assume that appropriate hardware and driver level software are available. I further assume that in a system which supports many graphical devices, each graphical unit has its own processor.

4.1.1 The Visual Display Screen Set

I refer to a 'set' because a main screen is sometimes extended by one or more auxiliary screens. Each might have its own device driver, and it is a function of the operating system (not the kernel) to provide the user with what he (with a slight stretch of the imagination) can regard as a single screen.

As was indicated in chapter 22 section 10.2, device drivers load a segment register via the kernel instruction `load_devSR` and thereafter control what is happening to their device using (pseudo-)memory addresses. In the present case I assume that appropriate mechanisms are available (in hardware, driver software and/or library software) for drawing graphical objects and writing characters in positions, sizes and colours, i.e. we do not concern ourselves with the pixel level as such. If multiple screens are to be treated as a single logical screen, this can be managed by a software module which directly accesses the device driver/interface module (see chapter 33 section 1 and section 4.1).

4.1.2 The Pointer Device

I assume the existence of a pointer device[189], which I will refer to as a mouse, as this is the most popular of the available pointer devices. It can be used to guide the movement of the pointer on the screen, and can take on different pointer forms, depending on where on the screen it is positioned (e.g. arrows, text cursor, eggtimer wait symbol, etc.). The pointer can be dragged across the screen and its movement can be tracked via interrupts which report its position as xy screen coordinates. It also typically has two or more buttons which can be used to cause single or double "clicks" which cause interrupts that can be interpreted

[188] At the time of writing I am 82 years old, and I have firm plans to write several further books on other subjects.

[189] This might even be a finger on some devices, such as touchscreen devices.

by operating system software.

4.1.3 The Keyboard

The keyboard, which on tablets and smartphones may be a logical keyboard, basically allows character input, which not only allows letters, numbers and punctuation marks to be input but also has further modifier keys (e.g. to signal upper and lower case, to input special symbols such as currency signs, percentage, ampersand and other symbols, to provide special functions, to delete characters already input, to terminate a line, and much more). The keyboard device may include a buffer which allows a text sequence as a whole to be passed to the main processor via an interrupt[190]. In systems which do not support graphics the keyboard may provide the only way of interacting with a system.

4.2 Graphical Libraries

The first prerequisite for adding graphics to the SPEEDOS design is to build a good graphics library, which can be integrated into SPEEDOS via the latter's library module mechanism (see chapter 18 section 6 and chapter 20 section 9.2). This should support all the usual mechanisms needed for a graphical user interface (e.g. windowing, menu creation, icons, drawing boxes and shapes with and without text, selecting colours, and much more).

4.3 A Possible SPEEDOS Graphical Interface

In this section I provide some general remarks which might assist the designer of a graphical interface to map his work onto the SPEEDOS architecture.

4.3.1 Gaining Access to the Graphics Devices

When a user logs in, his modules (in so far as they have appropriate privileges) can obtain capabilities to access the appropriate devices (e.g. screen, keyboard, mouse), as described in chapter 19 section 5 and chapter 26 section 2. Chapter 22 section 11 describes how these might be handled when a persistent thread is logged out/in.

4.3.2 When Should a Desktop for a New User be Set Up?

Most graphical operating system interfaces provide a desktop (which is actually a "start" directory) that is displayed over the main surface of a visual display monitor. The main items on the screen are directories and modules which are reachable from the desktop. To consider how this desktop is set up we begin

[190] In a simple system each character input may cause an interrupt which allows the kernel's input analysis routine to buffer the text until an end of line character is received (as in MONADS, see chapter 22 section 6.1). However, in a graphics based system it is important for users that they can see each character as it is typed.

with the situation in which a new user (and therefore his first desktop) is set up. Section 2 of chapter 31 describes how a new user is created – an existing user (the creating user), with the help of various methods of the Container Manager and other kernel co-modules,

a) creates a container for the new user;

b) transforms this into a process container;

c) registers the new user with the User Manager and the Login Service Module;

d) creates an initial thread in the process;

e) prepares this for running by providing a capability `startMod` (and a string name `startThread`, which is a symbolic name for the initial thread, registered with the User Manager), an integer `startRoutine` defining which of its semantic routines should be called, and a capability `rootMod` which is passed as a parameter when the routine is called and thus provides an environment for the new thread.

This might sound like a good time to prepare a desktop, which after all is part of the new environment that we need, but then we realise that all the activity so far described has taken place in the environment (and in one of the threads) of the *creating* user, and is therefore not under the control of the *newly created* user (especially if `startMod` is a standard module designed to introduce new users). Eventually the initial thread will call the process's Thread Control Manager, which will log the thread out (see chapter 31 section 4.1).

The new user first takes control when he first logs in to his new (previously logged out) thread. Not only is it appropriate to allow the user to create a desktop when he takes over control, but also because at this point he is sitting at a graphical device (which may be one of many in a multi-user situation) via which he can exercise this control. But how does this happen?

4.3.3 The Login Procedure Re-Visited

The following procedure is carried out when a logged out thread is to be reactivated, regardless whether the thread is for a completely new user logging in for the first time or for a thread which the user himself has logged out. The procedure is described in more detail in chapter 22 section 11.2, and only those details relevant to our task at hand (i.e. setting up a desktop) are described.

a) The user sits at a (in this context) graphical device with pointer device and keyboard. He indicates his presence by pressing a key at the keyboard, which causes an "unexpected" interrupt. This causes the kernel to create a login surrogate thread, which executes in the Login Service Module; this obtains access to the devices in order to establish the identity of the logging in user (see chapter 19 section 5). At this point it needs a mechanism for communicating with the user, i.e. a desktop!

b) When the surrogate thread has discovered from the user which thread should be activated it executes the kernel's `transfer_terminal` instruction, which transfers the device capabilities to the capability access area for the thread to be activated. It then activates the thread and kills itself.

c) At this point the re-activated thread (which is executing in the last module before it was logged out, i.e. the Thread Control Manager) carries out appropriate housekeeping tasks then calls the user's authentication module. This also needs a desktop to carry out its checks! When these are completed it returns to the TCM which in the case of a failed authentication logs the thread out again or if the authentication was successful it returns to the module which called it. This will also need a desktop!

4.3.4 Setting Up a Desktop

This brief overview of the login procedure shows that setting up a desktop is not a once only procedure but is needed several times. The best way to achieve this flexibly is to use an interface routine **`createDesktop`** of a graphical library module. This has the advantage (assuming that a capability for the appropriate graphical device is available, e.g. in the capability access area) that it can be called (using the kernel's library call `LC` instruction) from various different modules as necessary. Each use of this library routine will clear the current screen and set up a new screen. Parameters to the call will indicate what kind of desktop format is required. This might for example be a simple text screen with a selectable background or it might be a directory displaying the details of the objects in a directory.

4.3.5 Handling Multiple Activities

In the course of his activities at the computer a user might carry out a variety of activities in the same session, moving from one activity to another in an apparently random order. For example he might

- check his emails from time to time

while

- carrying out his primary activity (in my case writing a book),
- using the internet as a super encyclopaedia,
- writing an urgent letter,
- paying some bills which just arrived in the post via internet banking,

and so on.

One of the ideas behind the SPEEDOS notion of persistent processes and their threads is that these various, and to a large extent unrelated, activities can be managed quite separately and without interfering with each other. This is not always the case in current systems. Just to take a simple example, if I use a particular document editor in some current systems for more than one activity (concurrently) and switch between the different uses while leaving the documents open but off-screen, the likelihood is that not only do I activate the document which I want but the former document also reappears on the screen.

In SPEEDOS the intention is that different activities can (but need not) be organised in different processes, while the threads used in a user level process all contribute to the same activity. Hence when a user switches from one process to another the former activity (including its desktop) can be "cleared" temporarily by logging out the process, and a different activity (with a different desktop) can be cleanly retrieved by logging the process in[191]. Recall in this context that logged out processes and threads are simply saved in the persistent memory and can be retrieved by logging in to the process again (see chapter 22 section 11).

To simplify this, a graphics screen could provide a list of process icons e.g. at the bottom of a screen, so that in whichever process the user is active he can see his other processes. Simply by clicking on the image for a new process he can then start the logout procedure for his current process followed by the login procedure for a different process. However, this is simply a suggestion and should not be regarded as a mandatory aspect of a SPEEDOS system.

4.3.6 Other Desktop Windows

As the previous subsection implies, a desktop often consists not only of a main window (e.g. displaying the entries in the start directory), but also of other, usually less conspicuous windows, which can often be viewed as command bars (e.g. containing icons representing other processes of the same user and their threads, and/or continuous access to generally useful information such as the time and date, which printers are available, and so on). What actual windows are provided is a matter for the operating system design, not for the kernel.

If we assume that a command bar containing icons representing the currently logged out processes of the current user normally exists, this would make it easy for users to switch between their different logged out threads as they change to a different activity. A similar bar could be used to represent the currently active and the temporarily suspended threads of a user. These could be used, for example, to switch between threads. But once again these are matters

[191] It would also be possible simply to switch desktops temporarily to change the visible set of activity threads.

affecting the design of particular operating systems which need no further discussion here.

4.3.7 Composite Windows

Any window may be part of a collection of sub-windows which in some sense should be treated as a unity (e.g. in a main directory window there may be subsidiary windows which provide quick access to related windows). The entire collection of such windows can then be treated as a single entity for some purposes (e.g. for re-sizing and moving) while they may be treated separately for other purposes (e.g. for scrolling). While this is important for the user it provides no special problems for implementation, provided that appropriate data structures are developed which define the relationships between the parts. Therefore we do not discuss the issues involved, since they have more to do with application design than with kernel or operating system design.

Similarly some parts of a window (e.g. the frame, scroll bars, etc.) may appear to the user to be part of a single window, whereas they may be separate entities from the implementation viewpoint (often with the help of graphical library routines). Again we do not discuss the issues involved since they are scarcely impacted by the special aspects of the SPEEDOS design but may be relevant to the designers of the graphical libraries.

4.4 Some Technical Aspects

To discuss all the technical aspects of possible graphical interfaces for various SPEEDOS operating system designs is of course well beyond the scope of this book. However, it could be helpful if I provide a few general comments.

4.4.1 A Basic Approach

In a new SPEEDOS system (or in a multi-user SPEEDOS system when a new user is created) the user starts executing in the **`startMod`** module, receiving **`rootMod`** as a capability parameter. The **`startMod`** module could be a standard module and **`rootMod`** could, but need not, be a directory which holds further capabilities giving him access to system software and to software which the creating user wants to share with the new user.

As we saw above, after the new user has logged in and passed all the necessary authentication tests, the Thread Control Manager, which organised the tests, will return to its caller, which is probably the **`startMod`** module. At this point the new user will need a desktop. This can be organised by the **`startMod`** module making a library call to **`createDesktop`** and passing to it the capability **`rootMod`**, from which it creates a desktop which gives the user a starting point for carrying out his work, creating more processes, etc. If he does create new processes the procedure which it uses could follow the pattern which we have already established for the introduction of the new user, except that the process containers created will belong to the same user (unless he wants and has permission to create new users). Each of these new processes can have its own desktop, fashioned to suit the purpose of the process.

4.4.2 Displaying Directories and their Contents

When the content of a directory is displayed, the symbolic names associated with the capabilities can be shown along with appropriate icons. In contrast with most current systems the module names need not contain an indicator showing what type of module is represented, since in SPEEDOS the code associated with the module is fixed and cannot be directly modified[192]. However, since the syntax of a name certainly includes characters which can serve as separators (e.g. the full stop (period), colon, hyphen etc.), a user can optionally develop his own convention to indicate the type of the module (or can use a standard convention if this is made available by SPEEDOS OS designers[193]). Alternatively the type might be indicated by displaying an identifying icon along with the name. Another possibility is that the symbolic name for the code to which the module is bound (which could be made available via the template manager[194] that is stored in the directory) could be displayed if appropriate.

4.4.3 Interpreting Mouse Clicks

To access an item which is currently displayed on the desktop, the user selects and clicks on a desktop item. What happens then? I suggest that a click causes a desktop thread to receive a parameter from the system that provides the xy screen coordinates and indicates whether the click is a left click, a right click or a double click, etc. It must then determine the window in which the click is located and activate the item in some way, depending on the kind of click. This, and the following related activities, should be largely carried out in the graphical library routines.

4.4.3.1 Following the Pointer

When a pointer device is moved this movement is displayed on the desktop. The actual motion of the pointer should be independent of the window in which the movement occurs, since the pointer can traverse several windows. Nevertheless it is not entirely independent of the window (or sub-window) being traversed,

[192] The type of a module can be modified by using a conversion routine in association with a free capability.

[193] Unlike conventions in current systems, a possible SPEEDOS convention (which I consider to be superfluous) has no effect on the design of the system.

[194] see section 2.2 above.

since it is usual practice to change the display of the pointer itself to suit the kind of window being traversed. For example if the pointer is moving across a directory window it will probably appear as an arrow, but if the window is for typing text then its appearance may symbolise a cursor. This suggests that the pointer management must detect when a window boundary is crossed and change the pointer symbol accordingly. How can this be organised?

4.4.3.2 Managing the Desktop

A "map" of the current desktop, showing where the various windows are currently located, is needed in order to discover the window into which the pointer has moved when a window boundary is crossed and to change the pointer symbol. This map can be maintained by a standard Desktop Manager module which can be accessed by threads that create windows on the desktop.

Each entry in the map could hold at least the following information:

– the coordinates of the window's left top corner,

– a current width (horizontal) and a current depth (vertical) or alternatively the coordinates of the window's bottom right corner,

– a title (with font information) to be displayed at the top of the window,

– a pointer to the entry for the window in which this is nested (for the first level windows a pointer to the desktop entry),

– a list of pointers to entries which describe windows and sub-windows nested within this window,

– a symbol for the pointer to be used when the mouse is traversing the window,

– information about the borders of the window, e.g. what border style is used, how borders should be displayed (even if these are invisible),

– a thread capability for the thread which created and is responsible for the window, and

– any further information which the designer of the module considers appropriate.

This Desktop Manager module can activated by the initial start thread (since this has otherwise completed its essential functions), and the latter then becomes the 'desktop thread'.

Given such a map of the desktop, the coordinates can be used to establish the window in which the mouse is currently active.

4.4.3.3 Thread Activity and Mouse Movements

There are of course more kinds of pointer-based activity which affect the desktop than simply pointer movements and clicks, e.g. the dragging of objects from one position to another, the resizing of windows, as well as the effects of some of these activities on the appearance of other objects on the screen. The interesting question is which threads are active, and when, with respect to such events. Furthermore some of these activities have a side-effect on other (possibly unrelated) windows, e.g. when a window is moved in such a way that it reveals another window, which may need to be redrawn.

From the purely logical viewpoint it would seem appropriate to associate a separate thread with each window currently on the desktop. This would most appropriately be the thread which creates a window. To create a new window, a thread uses the appropriate graphical library routine **`createWindow`**, which inter alia creates a new subthread (see chapter 20 sections 5 and 8) to control each new window.

The desktop thread would initially be responsible for the desktop, and as new windows are created on the desktop it would remain the responsibility of the desktop thread to control those regions of the desktop which do not contain windows[195].

When the mouse pointer is at rest the desktop thread suspends waiting for a mouse interrupt. When such an interrupt causes the thread to be activated, it proceeds as follows:

- pointer position changes: the desktop thread moves the pointer to the new position.
- dragging objects held in a window: the desktop thread moves the dragged object to the new position, updates the map and activates the threads responsible for the initial position of the object and the final position of the object, providing details via a `put_message` kernel instruction described in chapter 22 section 10.2.3 in each case and redraws the (affected parts of the) desktop. Any consequences of the move (including cancelling it) can be taken by the application threads involved. If the drag action moves an object to a different position in the same window, the responsible thread is

[195] This might seem to imply that different threads should take responsibility for correctly moving the mouse across their part of the desktop, which would in turn imply that each time a user rapidly moves the pointer across the screen, several thread switches might be necessary in rapid succession, which could perhaps overload the system with thread switches. This would not be a very efficient way of using the CPU, especially when most of these thread switches would not achieve any effective purpose except to change the pointer symbol. Hence we suggest that all mouse *movements* are undertaken by the desktop thread. Since this executes in the Desktop Manager it is in a position to change the pointer symbol as this moves from one window to another.

advised of this. The recipient thread or threads can obtain the messages via the `get_message` kernel instruction.

- changing the position of, and resizing, a window: the desktop thread notes the change in the map and redraws the (affected part of the) desktop, and other windows affected by this[196].
- clicks (of any kind): it activates the thread associated with the current window, passing to it an indication of the kind of mouse click and the position of the pointer. These details are passed using the `put_message` kernel instruction.

The desktop thread then suspends awaiting the next pointer interrupt.

4.4.4 Application Windows

A click or double click can signify quite different things, depending on the window in which the click occurs, on the position of the click in the window and on the type of the click. As was indicated in section 4.3.5.3, all mouse interrupts are initially directed to the desktop thread. If the interrupt is for any sort of click (rather than just a pointer movement, window movement or drag action) the desktop thread, executing in the Desktop Manager module, redirects this to the thread for the window in which the click has occurred by activating the latter in the normal way, i.e. by calling an `activate` semantic routine of the thread's Thread Control Manager, which then passes this on to the User Thread Scheduler, after first transforming the coordinates into their relative positions within the window. The desktop thread first uses the `put_message` kernel instruction to enable an activated thread to discover why it has been activated.

When the suspended thread has been activated, it receives the interrupt information (location, click type) by executing the kernel's `get_message` instruction. It then interprets these to determine what action it must take. This depends entirely on the nature of the application.

4.5 An Example: Directory Windows

In conventional systems directory modules are part of the operating system software. However, as we saw in chapters 30 and 31 and in the earlier parts of the present chapter, they are simply application modules in SPEEDOS, with no special privileges. Hence the core actions of directory windows of the desktop provide a useful case study of how relatively simple application windows can be organised.

[196] These functions suggest that behind a window is a canvas which contains all the details displayable in the window. This can also be used to modify information in a window while it is not or is only partly visible.

4.5.1 A Question of Privacy and Security

In conventional systems it is quite normal that when a directory is displayed on the screen, further directories are automatically displayed in an associated window, which may appear to the user to be parts of the same window or window group, allowing him to navigate from one directory to others.

But herein lies a tricky problem for SPEEDOS, because it implicitly suggests that if a user has access to one directory, he automatically has access to the related directories and the files in them. Furthermore, in conventional systems directories serve as access control lists (ACLs) and therefore their contents (files and other directories) can be accessed (occasionally with restrictions) by any thread which has access to the directory (see chapter 2). But this assumption does not carry over to SPEEDOS, which is a capability-based system.

The basic rule of access in SPEEDOS is that to access an object one must have a capability for the object and is also restricted to accessing the object in terms of the semantic routines permitted by the access rights which it contains. The question therefore arises, how can directories be displayed in SPEEDOS? One quick answer might appear to be as follows. Allow related directories and their entries to be displayed in composite windows, as in conventional systems, but enforce the capability rule if the user attempts to access an object. However, this concept has a flaw, viz. that even if a user cannot access all objects which he can see in a window or on a screen, he can *see* them, and this alone can be regarded as a potential privacy and security violation!

The issue becomes clear when we look at the proposal in chapter 31 section 5 for defining a simple mail system, such that a user A, sending mail to another user B, can do this by inserting an entry in a mail box (directory) of B. We already described how A could use a special interface **`deleteMyEntry`** to delete the mail entry later. Thus User A has a capability allowing him to create an entry in B's mailbox and delete this entry later (if it still exists).

Clearly if user A could use this capability to view the entire mailbox, he would potentially see mails for B sent by other users. It is clear that such a violation of privacy should be prevented. The solution is similar to that for deletion. The directory needs two further semantic routines: **`view`** which allows the directory content to be viewed in full, and **`viewMyEntries`**, which restricts the holder of the capability to displaying and viewing only those entries which he has made. In this mail example the *recipient* of the mails needs to see *all* his mails (and therefore uses the **`view`** interface), while the senders of individual mails have no access to this semantic routine, but instead can only use the **`viewMyEntries`** interface.

4.5.2 Creating a New Module

When a new module (which might or might not be a directory) is to be created, the user might use a single click (e.g. a right click) on the area of the screen at which the new module is to be created. What happens then?

Recall that single clicks, in our example system, cause the desktop thread to activate a thread associated with the window. Since such a click can in some cases activate a long running program, during its execution the user may want to activate other modules. Hence the initial thread associated with the window can best achieve this parallel activity by creating a new thread to carry out the user-intended task. It can then suspend itself waiting for a further click. But it may carry out short activities itself.

Since different types of modules can be created, the activated thread must have access to all the necessary components to build the different kinds of modules which it offers, including for example a capability for the code of the new module, an option for associating with the new module a qualifier list and a module capability for the template and those required to create a container for the module (see chapter 19). Some of these are standard, while others can be determined via submenus. Finally it creates an entry for the new object in the directory.

4.5.3 Activating Modules

The most used facility of a directory module is probably the activation of modules which have entries in the directory. Typically the user double clicks on an entry to activate the corresponding module, and of course the desktop thread (acting in the desktop module) passes the double click to the thread responsible for the window. As was already described for the creation of new modules, the window associated with the directory will create a new thread, pass the "parameters" to it via the `put_message/get_message` mechanism and the new thread will then take the necessary steps to activate the module to be called.

This primarily involves making an inter-module call via the capability held in the directory, but before that can be done the semantic routine to be called must be selected by the user and the parameters for this routine must also be prepared for the call. By their nature these cannot be predefined in the directory entry. Hence the thread must obtain the necessary information from the user. He can do so by retrieving the template manager which is stored in the entry and use this to request the name of the semantic routine to be called (e.g. using a pull-down menu) as well as the appropriate parameters for this, perhaps using a pop-up menu (which could indicate the symbolic name of each parameter and the parameter type for each parameter in turn). The directory thread can check the parameters as they are typed in, convert them to internal format and, after creating the appropriate four parameter segments on the stack (see chapter 20 section 6.2), can use segment register 0 to enter them into the input parameter segment which it has created at the top of the kernel stack, then execute the inter-module call instruction. It will eventually return, perhaps passing back some return values of interest to the user. Meanwhile the user can carry on using the pointer to carry out further work.

4.5.4 Returning Results

When an inter-module call has been activated as described above, it will eventually return, and at this point it may have some results to report to the user. Since the thread might have executed over a long time span, it may arrive when the user is not expecting it, or is busy carrying out other activities. Hence the question arises how he can receive these results at his own convenience. A possible answer is that executing threads may be listed or symbolised by icons in a separate window on the screen (and perhaps coloured red while they are still running, or green when they have completed). In this case the user could double click on a green thread symbol and be shown the results returned by the module.

5 Concluding Remarks

A substantial part of this chapter has attempted to show how a basic SPEEDOS system can be enhanced by the use of graphical equipment, and to the extent that it does this it should be regarded more sceptically than those chapters which describe other parts of the SPEEDOS design. The reason for this is that I have in the past never been deeply involved in designing a graphical interface nor using graphical hardware or software. I hope that the chapter will nevertheless be helpful in so far as it makes suggestions about mapping a graphical interface onto the SPEEDOS architecture.

Chapter 33
I/O Devices and Spooling

Previously, input/output devices have only been mentioned with respect to SPEEDOS in Chapter 22, in connection with the handling of asynchronous interrupts. When an interrupt occurs, the kernel's interrupt analysis routine examines the interrupt and passes it on to the appropriate kernel process. This chapter discusses in more detail what happens when the interrupt is a general device interrupt (e.g. from a printer) and explains how print spooling can be managed in a secure way. Interrupts from discs are not included since these are directly handled in the kernel as part its virtual memory management function. Similarly communication with other nodes is initially handled within the kernel and is not considered in this chapter.

1 Device Drivers

The module which directly interfaces with a device is its device driver. However, in contrast with device drivers in most conventional systems there is very little that is special about a SPEEDOS device driver. It is implemented as a normal SPEEDOS module. It has appropriate semantic interface routines tailored to its own needs, and only other modules which have a capability for the device driver can call these routines and therefore use the device which it controls. Its only special privilege is that it has a kernel capability, which is typically stored in a read-only segment in its code; this allows it to execute the kernel instructions `load_devSR` and `wait_interrupt`.

A request by a module to activate an I/O device is invoked and handled as an in-process operation within the thread which requests the I/O operation. When the thread has invoked the device driver, the latter activates the device. In conventional computers there are two ways of achieving this, depending on the hardware design. One is a special hardware instruction (here called `start_device`), which can only be executed when the computer is currently in a privileged mode (corresponding to SPEEDOS kernel mode). Alternatively, the hardware might use memory mapped I/O[197]. In SPEEDOS memory mapped I/O is strongly preferred, because this gives the system much more control over the operation. In this case the only special privilege required is that a segment register is loaded to address the device memory. To achieve this, the device driver uses the kernel instruction `load_devSR`, specifying as operands the number of the segment register to be loaded and a device capability which defines the device[198].

Having activated the device, the driver (still executing in the application thread) executes the `wait_interrupt` instruction. The kernel then notes that an interrupt is expected from the device and which application thread should be activated when the interrupt arrives. It then calls an interface of the User Thread Scheduler, advising it to suspend the application thread.

When the device operation terminates it causes an interrupt, which leads to the activation of the appropriate kernel device processes (see chapter 22 section 10.2). Having examined the interrupt parameters and taken any further actions necessary, this then activates the Thread Scheduler to advise it that the interrupt has arrived, allowing the latter to put the waiting application thread into the ready state for eventual re-activation. The kernel process then suspends itself and waits for another interrupt.

The above description provides a general pattern, but does not exclude the possibility that further optimisations may be introduced in particular situations.

2 Device Allocation

In older computer systems the allocation of devices to application (and to some extent system) threads was quite a problem. Usually a central device allocator module existed which had the job not only of determining which users could use the system's I/O devices and when, but also whether particular users had a right to use particular devices and with what priority. But above all the problem of deadlocks (sometimes called *deadly embraces*) had to be solved.

2.1 Deadlocks

A deadlock (in the operating system sense) arises when applications have received permission to use more than one device or other resource (e.g. a semaphore) which they claim for concurrent use, with the result that there is no order

[197] For more detail see Chapter 6.

[198] Segment registers containing mapped device capabilities can neither be stored as pointers nor copied into other segment registers (including the parameter registers). Device capabilities can only be used at the node indicated in the capability. Usually the right to copy a device capability is unset.

of execution which allows all the threads to complete, because each is waiting for another to relinquish a resource. In the very simplest case this can involve two threads (say T_1 and T_2) and two resources (say R_1 and R_2), such that R_1 has been allocated to T_1 and R_2 to T_2. Then T_1 claims R_2 and waits for it to be allocated, but instead of releasing R_2, T_2 claims R_1 and waits for it. In this situation each is waiting for the other and neither can proceed. Real situations could be far more complex than this, for example involving circular waits by several threads.

The fundamental problem arises when threads can claim *exclusive* use of at least two resources. Two quite different strategies for solving this problem were developed by different system designers. The first approach was to *detect* the problem (after it was noticed that the applications were not making progress, or as a regularly run algorithm) and then to "solve" it (e.g. by deleting one of the offending threads). The second approach attempted to *prevent* the problem from occurring (for example by defining a set order in which devices can be claimed, e.g. if a thread has successfully claimed a printer, it may not then claim a card reader. If you are interested in this issue, you will find plenty of information about various strategies in older books on operating system design.

In modern systems deadlocks are far less a problem than a few decades ago. The main reason for this is that many I/O devices which need to be used exclusively (e.g. card readers, tape drives) have all but disappeared. The second reason is that some modern devices, although they generally need to be used exclusively, are allocated not by software modules, but rather by humans claiming them separately (e.g. keyboards and monitors which are general claimed by a user simply sitting at an unused device). The third is that many computer users now have their own personal computer system, which is used without competition from other users[199]. The fourth is that devices which are used exclusively have become cheaper and more plentiful, so that competition for their use has diminished (e.g. printers). Consequently a device allocation module will rarely be needed and is not illustrated here.

However, in a multi-user system the use of printers and other output devices (e.g. graph plotters) could in principle still be a problem. But there is a different way of solving that problem; this approach is known as *spooling*.

3 Spooling – The Basic Principle

Spooling (*simultaneous peripheral operations on-line*) is the name given to a method of interfacing programs and slow I/O devices in such a way that (a) us-

[199] This alone does not solve the problem, e.g. if a user organises the concurrent use of threads which claim the same resources.

ers are not inconvenienced by having to wait for the device to become available and (b) the device can be driven at full speed. This technique used to be important for devices such as card and paper tape readers and punches, but today its main application is for printing. It can also be used to achieve considerable cost savings (e.g. telephone connection charges) with output devices such as computer driven faxes.

To understand why print spooling is necessary, consider what would happen in a typical multi-user system without it. Such a system might have many users and few printers. A user at a terminal wishing to run an application program which produces print output – most applications produce results for printing – would therefore have to wait for a printer to become available before he could start his program, or before the program could start printing. This would be extremely inconvenient for the user, who in some older timesharing systems could have done nothing but wait, or tap the keys of his terminal impatiently. Then when the printer eventually became available his program would start using it.

Now suppose that when a printer at last becomes available to a user, he starts an application which does a lot of calculation and which every five minutes for the next hour prints a result consisting of one line. The other users waiting for access to the printer would be rather irate at such a wasteful way of using the device!

Spooling solves both problems simultaneously. The idea is simple. First the application is run, but instead of it using a printer directly, it outputs its results to a disc file. (Discs are shareable so there is no waiting problem for the user.) Then when the file is complete, it is passed (in conventional systems) to a system thread, called the spooler. The spooler runs continuously (whenever there is work to do) printing files one after another (for different users). With this technique a user never waits for the printer and, because the files to be printed are in their final form, the printer can be driven at full speed.

This is fine as far as efficiency and convenience is concerned, and it is in principle how nearly all multiprogramming systems organise the printing of user output (although it is amazing how some operating system designers can complicate even a simple idea like this!).

But from the security viewpoint it is far from ideal. The spooler program has the opportunity to see just about every interesting item of data in the system and make a secret copy of the juiciest bits for an unauthorised user. Furthermore, because the spooler has to be able to access the files of many users, it is sometimes given a protection status that allows it unrestricted access in the file system, which means that it can even access files which it has not been asked to print.

4 Spooling – The SPEEDOS Approach

How can these problems be avoided in a SPEEDOS system? The problem of the spooler having a special privileged status is avoided in a capability system because SPEEDOS has no privileged status of this kind. The only "privileges" are conferred by capabilities (including kernel capabilities) and if used properly these limit the privileges which they confer only to those needed to carry out a specific task. If a spooler is to print a file, it needs a capability for the file. Thus it needs a capability for each file to be printed, but not for others. That is a start, but it is not sufficient, because the spooler might retain the capability or make a secret copy of the file, or pass information in the file to a third party. So there is potentially both a revocation problem and a confinement problem.

In the remainder of this chapter a spooling system design will be presented which sets out to solve these and other security problems while retaining the efficiency and convenience aspects of spooling. We begin by considering some general issues associated with spooling.

4.1 Spooling Files, Interfaces and Drivers

Spooling files are normally created by application programs using a format which allows the same file to be printed later on many different kinds of printer and displayed on many different monitor types. A widely used example of such a format in conventional systems is Adobe PDF (which is a successor of PostScript, a format which was earlier in widespread use for printing).

At the other extreme, manufacturers of actual hardware devices produce printer drivers (and drivers for other devices such as monitors, graph plotters, etc.) which differ widely from each other (even those manufactured by a single company), and which are often quite complicated and messy to use directly. Consequently they usually provide software drivers with their hardware which accept a much simpler form of interface and do the messy things for the application.

Since it is impossible – and also unimportant in this context – to discuss all the possibilities in detail, a simplified scenario is now presented in which it is assumed that application programs create PDF files[200] and that these serve as print files which are accepted by the printer drivers in a system.

In order to shield spoolers and other software from the complexities of an actual driver interface, it is assumed that in a SPEEDOS context an interface module is implemented for each driver, which allows all other software to ac-

[200] Since 2008 there is an ISO standard for PDF files.

cess the printer in a standard form. This results in the situation shown in Figure 33.1, which illustrates the final stage of printing a file in a spooler thread.

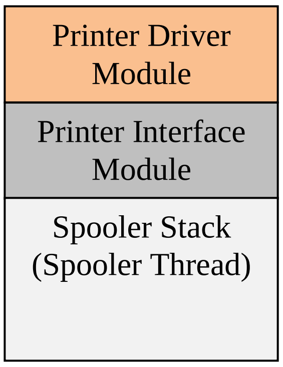


Figure 33.1: The Final Stage of Printing a File in a Spooler Thread

When called by the interface module, the driver takes the necessary device-dependent steps to carry out the actual printing (including a call or calls on the kernel's `wait_interrupt` instruction). On completion of the printing the driver returns to the interface module, which returns to the module that called it.

4.2 An In-Process Spooling Architecture

We now consider what modules and threads are needed in order to provide spooling in such a way that it is both efficient and secure. The normal starting point in an in-process architecture is that a task should be fully carried out in a single thread (or a collection of cooperating threads following the in-process principle), since this provides a framework which encourages both security (e.g. in the case of SPEEDOS via the thread security register) and efficiency. However, the spooling principle demands that the spooling activity should be carried out in a separate thread from that of the thread requesting the printing. In a conventional system this is a thread belonging to the system which accepts requests from all users. For each printer such a system thread is needed. Before rushing into such a solution as the obvious and only one, let us consider an unconventional alternative.

4.2.1 Each User provides his own Spooler Thread(s) for each Printer

The idea that the spooling activity should take place entirely in a thread or threads owned by the user is attractive for a number of reasons[201]. First and

[201] Some readers may fear that providing extra spooling threads for each user will be inefficient, e.g. because these will clog up the scheduler. However a SPEEDOS user thread is implemented as part of a user process, which is in fact more comparable to a file than a conventional process. In SPEEDOS user threads must only be made known to the User Thread Scheduler (UTS) when they become active and are then removed from the UTS

foremost, this allows the user to exercise some control over the security issues, e.g. via the thread security register. But also, since the owner of the thread is the user, the processor time, the disc and printer I/O operations, and any other resources used can be automatically clocked up to the user on whose behalf they are consumed, not to a system thread. Consequently spooling is not a special case with respect to accounting, logging, resource limits, etc.

In a single user system this architecture would not present any significant problems, but the question arises in a multi-user system how the different users could then share access to the same printer, and how the print requests of different users could be handled fairly. We tackle the fairness issue first.

4.2.2 How Can Print Requests be Handled Fairly?

It is by no means clear what *fairly* means when users are competing for the use of the same printer. Is it fair, for example, if a user requests in quick succession that three of his files should be printed which each take an hour or more to print, thus preventing users with much shorter print jobs from accessing the printer? To avoid this situation it would be possible to adopt a scheduling policy known as shortest job first, i.e. a printer scheduler gives preference to print job requests according to the (estimated) time they will need the printer.

Is it fair, for example, to give higher priority to some users over others? Should user jobs be selected for printing on the basis of their urgency? Who determines urgency and using what criterion? For example should print jobs for the Managing Director of a company get higher priority for his print jobs than those of his staff?

These issues make it clear that in a good system it is appropriate for an installation to have scheduling rules which are laid down in a scheduling algorithm suitable for that installation. From the present point of view it thus seems appropriate to have a scheduling module which determines when different print jobs (possibly from different users) should be started. At first sight this may seem to suggest that this should exist in a separate thread, which is used by all who have a right to use the printer. However, that would be a typical out-of-process solution for the problem.

4.2.3 Printer Scheduling Modules and User Spooler Threads

The in-process solution is that such a scheduling module (hereafter called the Print Scheduler Module) should exist, but not in a separate thread. Instead it is a module which is invoked in different user spooler threads as appropriate. Since

when they become inactive, i.e. they are not a permanent burden to the UTS. While they are inactive they behave more like a passive file.

this module determines which print request should be passed next to the printer interface module/printer driver module (see Figure 33.1), and since the latter is executed in a thread of the user whose print job has been selected, a switch must occur to a waiting spooler thread of the user whose print request has been selected as next for printing. How should this switch be coordinated?

The answer of course is via semaphores. The scheduler maintains a list of print requests from various users and sorts them into an order determined by the scheduling policy. When it receives a print request while the printer is occupied, it uses a P operation to suspend the current thread, which we assume is a user spooler thread (see below). When some other spooler thread (of the same or another user) frees the printer it returns to the Print Scheduler Module in the current thread, it then examines its scheduling list and selects the next appropriate spooler thread to execute, by issuing a V operation to wake up the thread[202], which will then call the Printer Interface Module to start the printing of its own print request.

The user spooler thread structure just described is illustrated in Figure 33.2.

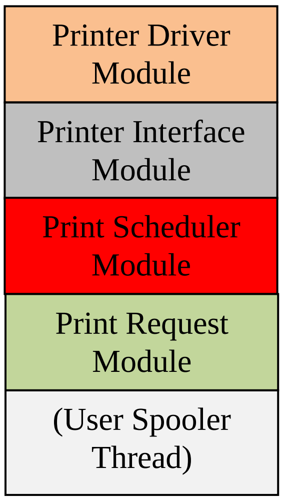


Figure 33.2: Printing a File in a Spooler Thread

4.3 The Print Request Module

When a user requests the printing of a file it will be executing in a normal application thread of the user (or in a Command Language Interpreter Thread, see Chapter 32), not in a spooler thread. This suggests that the PRM should be active not only in the user's spooler threads but also in any application thread which requests that a file be printed. In this way one of its semantic routines can be seen as a 'print' command, which is called in application threads that have a

[202] In order to wake up the selected thread it will need a thread capability for this thread, which is part of the print request information provided by the thread itself.

capability allowing this (see Figure 33.3).

This solution requires that the PRM can be invoked both in application threads requesting the printing of a file and in user spooler threads in order to pass print requests to the Printer Scheduler Module (for the printer on which the request is to be printed). How are these two "halves" of the module linked? In other words how is the print request passed from the application module to the spooler module?

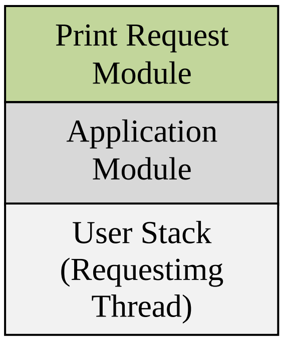


Figure 33.3: Receiving a Print Request

The simple answer is via a shared data structure, as is usual in in-process systems when a module is active in more than one thread. How this happens in detail depends on how user spooler threads are organised. The following suggestion attempts to avoid a number of problems which could otherwise arise in this scenario.

Although it was suggested above that there should be a user spooler for each request, I now propose that a single thread receives requests from applications of the *same* user; this allocates requests to different subthreads, which it creates and manages. These carry out the actual spooling activities. In the first stage, the PRM receives print requests for its user from application threads of that user via a shared data structure. This serves as a multiple producer/single consumer bounded buffer, in which different application threads of the same user can each produce print requests and enter them into a shared buffer while the PRM, acting in a separate user thread, consumes them in turn and allocates them to separate user spooler threads.

Why not instead simply also have multiple consumers (each being a user spooler thread)? That would imply that a sufficient number of user spooler threads for this use must always pre-exist. (If insufficient were to exist then a print request for the user would have to wait for another print request from the same user to complete, which would then adversely affect its chances at the Print Scheduler Module level, where requests from different users compete with each other.)

In the proposed solution the single consumer would receive requests in the order in which they had been submitted and ensure that each is immediately allocated to a user spooling subthread. The implementation then becomes a strategy decision: it would be possible to maintain a small pool of threads which might be sufficient for the normal flow of print requests, but if this turns out to be insufficient the Print Request Module can dynamically create new user spooler subthreads as required. Alternatively it might adopt a strategy of dynamically creating and deleting a user spooler thread for each request from its user. In either case we end up with a scenario as illustrated in Figure 33.4.

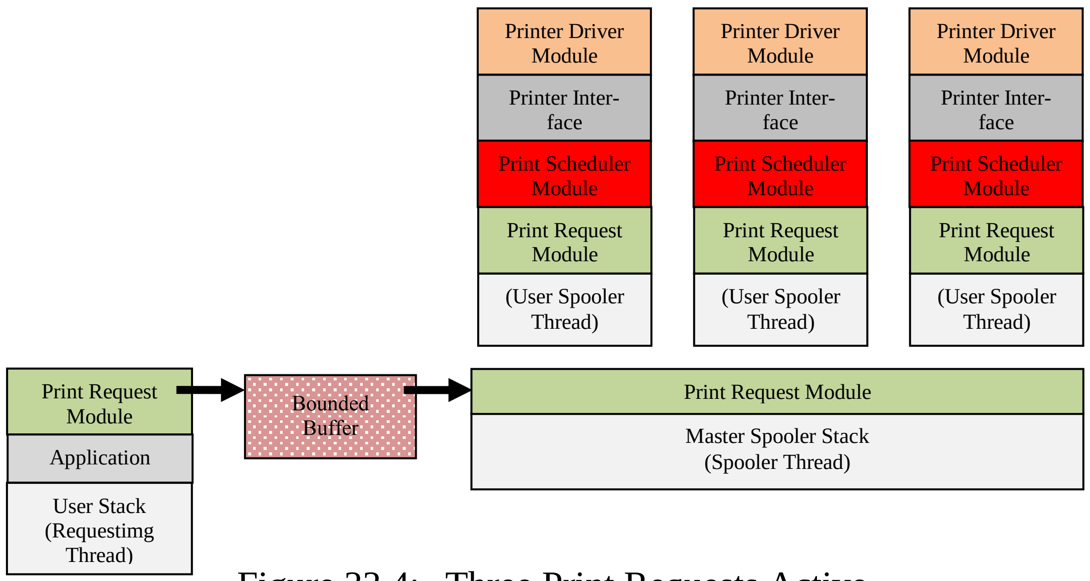


Figure 33.4: Three Print Requests Active in the Spooler Threads of a Single User

This figure suggests how the individual user spooler threads might be implemented, viz. as application subthreads of the Print Request Manager (see chapter 20 section 5). In this case the initial master spooler thread continuously waits (via the bounded buffer) for new print requests and then dynamically creates subthreads for them (and deletes them on completion).

4.4 After the Print File has been printed

When the printer driver has completed the printing of a file it returns to the printer interface module, which then returns to the Print Scheduler Module; this then selects a further request to be printed. However, it does not simply pass this to the Printer Interface Module/Printer Driver, but activates the thread which made the request and which is now in a wait state (see section 4.2.3). The best way to organise this is by each thread having a private semaphore (see chapter 8 section 12.3) in the Print Scheduler Module, with the current thread activating the thread whose print request has been selected. Once activated this thread calls the Printer Interface Module/Printer Driver to have its file printed.

Meanwhile the thread which activated it (i.e. the thread whose file has now been printed) continues by exiting from the Print Scheduler back to the Print Request Module, where it exits from the subthread (which causes its deletion). In this case there is no need to advise its creating thread of this.

4.5 Scheduling Equivalent Printers

In some cases several equivalent printers might exist (e.g. in the computing lab at a university). ("Equivalent printers" simply means a set of printers which are equally convenient for the user to use.) This requires a relatively trivial extension to the system as described above. In its last phase the print scheduler will normally have a binary semaphore which is used to signal that the printer has been claimed (via a P-operation) or released (via a V-operation). If there are multiple equivalent printers then this can simply be replaced by a resource set semaphore, thus allowing more than one printer to be in use concurrently. This is of course initialised to show the number of printers which are free.

4.6 The Print Request Module

A PRM's semantic routine `print` (i.e. the print command) expects three parameters:

i) a capability for the file to be printed,

ii) a name by which the user wishes to refer to the file (e.g. to appear at the head of the printed file, or to indicate which file if he decides to cancel the print request before it has reached the printing stage),

iii) the number of print copies which he requires.

However, when the print request module places this in the bounded buffer, it adds a fourth parameter:

iv) a thread capability for the thread in which the print request is made. This is needed to synchronise the bounded buffer with the spooler thread.

When the printer interface module prepares for the printing, it can establish the identity of the user requesting the printing from the fact that it is operating in a thread of that user (and can therefore use the kernel's environmental instruction `current_thread_owner`). Hence this need not be passed as a parameter[203].

4.7 Simplifications for Single-User Systems

Many desktop and laptop computers, as well as smartphones and other mobile devices with operating systems, are used exclusively by a single user with pri-

[203] If a symbolic user name is to appear in a heading of the print output, the interface module for the driver can provide this, if necessary by looking up the user's unique identifier in the user directory (see Chapter 31).

vate printers. For such systems the spooling mechanism can often be simplified in two ways.

First, the Printer Scheduling Module (PSM), which is designed to schedule requests from different users, can usually be eliminated. This simply involves providing the Spooler with a capability for the `PrinterInterface` module, rather than with a PSM) capability. However, if the user has equivalent printers, he can consider retaining a PSM to manage these.

The second possible simplification is that a single-user system will only need a single thread for each printer, because the PRM already supplies a queuing system which (in a single-user system) does not suffer from the fairness problem described earlier.

4.8 Additional User Facilities

The PSM would need some further routines to allow requesters prematurely to report on the current position of a request or to remove a request from the buffer. These are not described in detail.

5 Security Aspects of Spooling in SPEEDOS

As was noted in the introduction to the spooling section of this chapter, from the security viewpoint the conventional spooling technique is far from ideal, because spooler software has the opportunity to see just about every interesting item of data in the system and make a secret copy of the juiciest bits for an unauthorised user. Furthermore, because the spooler has to be able to access the files of many users, it is sometimes given a protection status that allows it unrestricted access in the file system, which means that it can even access files which it has not been asked to print.

5.1 Checking the Right to Print

The right to print a file is determined by the "metaright" `print` (see chapter 26 section 3.3.1). However, unlike the generic rights (to which categorisation it ideally belongs) it is not immediately clear how this can be controlled. In the case of the normal generic rights (see chapter 16 section 3.2) the implementation is carried out by the Container Manager – a kernel co-module–, which is normally responsible for checking the generic rights. As so far described in section 4 above, no module in the chain is a kernel co-module. How then can it be checked?

Ideally it would be good to have the necessary check carried out as soon as the print request is made. The Print Request Module would be the ideal module to carry out the check, but this module is a normal user module, which in principle could be different for each user. Hence, although it can carry out a check of the corresponding metaright in the capability for the file, this check could not be enforced.

The Print Scheduler Module is the next module to see the capability for the file to be printed. In a multi-users system this module is not directly under the control of a particular user. In principle there is a separate (instance of a) Print Scheduler Module for each printer or group of equivalent printers, which must be used by all the users wishing to print a file on the printer(s) which it controls. Its independence of a particular user makes it a good candidate for carrying out checks which apply to the files of each user. Hence when a Print Request Module passes a print request (including a capability for the file to be printed), the Print Scheduler Module should check whether the print right is set in the capability.

The Print Scheduler Module thus becomes a kernel co-module, which is in any case appropriate for a module which carries out a scheduling task that arbitrates between users. When a printer is installed in a system the installation manager must allocate for it a Print Scheduler Module, a Printer Interface Module and a Printer Driver Module. He must also provide and distribute capabilities for invoking these modules. (Without access to these capabilities a user cannot print files on the corresponding printer(s)).

5.2 Securing the User's Information

Except for the device driver, which will be considered shortly, no module in the SPEEDOS spooling system outlined above has any special privileges, e.g. kernel capabilities (not even the Print Scheduler Module despite its status as a kernel co-module). Spooling modules have access only to those files for which they receive a capability, and the only file capabilities made available to the spooling software are the individual files to be printed and capabilities allowing each module to call the next module in the spooling chain.

Although the scenario outlined in section 4 ensures that all spooling operations are carried out in spooler threads owned and controlled by the user of the file to be printed, this does not eliminate all dangers. In particular the code modules which are used in these threads are potentially untrustworthy, unless they can be controlled by SPEEDOS security mechanisms.

Hence there could be a danger that a module in the spooling system which contains secret code (e.g. a trojan horse) may attempt to copy or retain a print file capability for later use, or pass the capability or information in the file to a third party. So there is potentially both a capability revocation problem and an information confinement problem to be considered[204].

The first security decision which the user has to make is to secure the capabilities for

– his print files,
– the system modules discussed above,
– the parameters passed to the spooler module.

We now consider the security settings which should be applied in these capabilities.

5.3 Securing Print File Capabilities

When an application initially calls the print request module, it provides a copy capability (not the original) for the file it wishes to have printed. At this point the user has the opportunity to tailor the rights in this capability to secure it from attacks. Here are some of the measures which might be taken.

a) The Printer Interface and Printer Driver modules need direct access to the file's content. Hence the metaright `permit_free_cap` must be left set. However, no spooler module needs access to any of the semantic rights, so that the special setting 00 can be set in the semantic rights. This prevents any semantic routine from being invoked (see Chapter 26 section 3.1).

b) The following further metarights could be unset:

   i) `permit_duplicates:` if unset this results in each subsequent copy action becoming a destructive move, i.e. the source capability in each copy operation is invalidated, and therefore becomes unusable. This in effect revokes the capability at each stage of its transition to the driver.

   ii) `permit_read:` unsetting this prevents the software in the spooling system from examining the capability.

   iii) `permit_calls:` unsetting this ensures that the software cannot invoke any semantic routines of the print file module. This is possible because the driver accesses the file not via a semantic routine but via direct access using a free capability.

   These rights could be unset in all the metarights categories listed in chapter 26 section 3.1.1.

Notice that as the thread returns from the Driver, none of the modules to which the return is made have access to the capability, since the parameter segment in

[204] Other potential dangers are that capabilities passed to the spooling system for printing might be used to alter or destroy their contents. These can be avoided by unsetting the "write" bit in the semantic rights.

which it is held is inaccessible after return instructions have been executed.

5.4 Securing the Capability for the Print Request Module

If the owner of a thread finds the system's standard Print Request Module satisfactory then he will initially find this in the public software made available when his directories are created (see chapter 31, section 3.2.1). Alternatively he may choose to develop his own module or buy a module from a software vendor.

He can then make a copy of the capability for this and has the opportunity to reduce the rights in it. In particular he could reduce the semantic rights such that modules executing in the thread can only call the module's semantic routine **`print`** (and any associated rights allowing him to check the progress of, or to delete, a print request (see section 4.8)). He can also reduce the metarights as he finds appropriate, e.g. by unsetting the rights: `permit_file_copy`, `permit_in_param_copy`, `permit_free_cap`, `permit_read`, `permit_dir`, `permit_print`[205].

He can then distribute the capability for use by modules in his thread(s) by following the same pattern as the distribution of capabilities for other standard modules (see chapter 19 section 5), i.e. a user wishing to print can use a kernel instruction to obtain the capability for the Print Request Module which the thread owner has set up for the thread.

5.5 Securing the Capability for the Print Scheduler Module

The possession of a capability for a Print Scheduler Module, which contains a scheduling algorithm associated with a specific printer, indirectly gives a user access to that printer. Consequently the system administrator will provide specific users with the capability. The recipient of such a capability will need a copy of this capability for each thread which he creates as a spooler thread. In the architecture proposed such threads are actually subthreads which are dynamically created as demand dictates. This implies that the administrator cannot restrict the capability's metarights to *once only* use for the new owner. But if he allows unrestricted copying, he loses control over who can use the capability! To retain control, i.e. to prevent the recipient user from passing it on to other users, he can make a copy capability in which he unsets all the metarights for foreign owners[206] and foreign node owners, as well as the `read` and `print` rights for the same owner, while leaving the remaining permissions for the same owner metarights set, but also unsetting all the once only permissions for the same owner. This alone would not solve the problem because the administrator would

[205] This is not intended to be an exhaustive list.

[206] Note that in this context "owner" does not mean the owner of a file, but refers to the source and destination segments in move or copy operation, see chapter 2 sect. 3.3.1.

still need to pass the capability to a different user. He then places the capability in an output parameter segment (to pass to the new user) leaving the *once only* permission for `permit_out_param_copy` set in the foreign owner section of the metarights. This allows him to transfer the capability to the new user, thereafter leaving the `permit_out_param_copy` unset. Thereafter the new user can only copy the capability between his own segments.

The capability for the Printer Interface Module will be held in the constant segments of the Print Scheduler Module and will have been pre-secured. Similarly the capability for the Printer Driver Module will be held in the constant segments of the Printer Interface Module and will likewise have been secured, so that the normal user cannot reach these directly.

5.6 Securing the Confinement of Information and Preventing Unauthorised Access by the Spooler Software

Although the above metaright settings provide a strong measure of protection, they do not guarantee that either the capability for, or the information in, the file to be printed cannot escape to a third party. But unsetting confinement rights in the capability *for the file to be printed* cannot confine the spooling modules, because the effect of de-activating confinement rights specified in a capability only occurs when the capability is actually used either to make an inter-module call or as a free capability. But this implies that the *capabilities used to invoke the spooling modules* (rather than the file to be printed) can be used to restrict the confinement rights.

5.6.1 Confining the Print Scheduler Module

The Print Scheduler Module must be able to store the capability in its file data and pass it on to the Printer Interface Module. It requires no further facilities with respect to the capability. How might it illicitly make the capability available to a third party? Here are its only possibilities:

a) It might pass the capability as a parameter to an inter-module call, using a secret capability to make the call.

   This can be prevented by ensuring that it cannot use a capability to make such a call. Hence the module call confinement rights `permit_param_calls`, `permit_nonparam_calls` and `permit_comodule_calls` should be unset.

   The right `permit_const_calls` must remain set to allow the Print Scheduler Module call the Printer Interface Module. We consider below how it can be guaranteed that there are no illicit capabilities in the constant segments of the module.

b) A module executing in a different thread may call the Print Scheduler

Module (assuming the improbable situation that it can obtain a capability for this module), using a different (secret) semantic right which has a return parameter via which the print file capability is passed. This can be avoided by unsetting the Information Confinement Right `permit_return_cap`.

Then there is the question: How might the Print Scheduler Module illicitly make the content of the file available to a third party? In order to do that it must be able to access the content. Since the capability for the print file itself is protected by the unsetting of `permit_calls`, access to its semantic routines is unavailable. The only possibility would be via access as a free capability activity (which implies knowledge of the internal file structure). However, this is easily prevented by unsetting the right `permit_free_cap` (at all levels) in the capability used to call the Print Scheduler Module.

5.6.2 Confining the Printer Interface Module and the Printer Driver Module

Much of what was described in the previous subsection with respect to the Print Scheduler Module can be applied to the Printer Interface/Printer Driver. The fundamental difference is that these need access to the file content, so that the risk arises that these modules could leak the content of the print file.

This can to some extent be avoided by unsetting the Information Confinement Right `permit_file` (see Figure 25.1) in the capabilities used to invoke them. The effect of this is that they cannot have persistent data. In fact they do not need persistent data, because they have no need (according to their purpose) to store information persistently. The working space which they need can be provided in the temporary data that they can create and address via segment register 4 (see chapter 18 section 5.1). This has the effect that they cannot keep copies of the file content after they have returned.

5.6.3 How Can the Capabilities for the Spooler Modules Be Restricted?

We have described the restrictions which might be applied to the capabilities of the Printer Interface/Printer Driver Modules, but this raises the problem that the owner of a spooler thread does not have direct access to these, because they are held in constant segments of the code modules of the Print Scheduler and Printer Interface Modules! How can these settings then be applied?

The answer is straightforward. When a driver is introduced into the SPEEDOS system its owner is (or becomes) the system, and since it is in the interest of SPEEDOS to provide a very secure system the settings which we described above can be applied by the system to the capabilities for the Printer Interface and the Printer Driver Modules.

Furthermore such modules are thoroughly checked (e.g. using bracket routines) to provide a further guarantee that they are secure from information leakage. Thus although the individual user, as a non-owner, cannot bracket these modules, he can rely on the system to guarantee their safety.

5.7 A Concluding Note on Security Settings

In this chapter I have concentrated on describing the most important security settings with respect to the management of spooling, but as a glance at chapters 25 and 26 will show, I have by no means provided an exhaustive description of how all the security settings should be set in the relevant capabilities.

Furthermore no attempt was made to illustrate the use of all the security mechanisms available in SPEEDOS. For example the special facilities provided by the Thread Security Register and those provided by bracket routines were not needed to solve the spooling issues and were therefore left unmentioned. That should not be taken as an indication that these mechanisms are unimportant or superfluous. Both of these are important features of SPEEDOS which have a significant part to play in solving security problems in many environments.

6 Other Devices

In this chapter we have concentrated on the spooling issue because it is more complex than most other security problems associated with the use of input-output devices. This has allowed us to illustrate how security settings can be used effectively to prevent information leakage and other security problems. But at the same time it has allowed us to illustrate in a more positive sense how device schedulers and drivers can be organised at the operating system level.

Of course, not all devices have been illustrated, but a general pattern has been supplied to cover other devices which may be spooled, such as graph plotters and other drawing devices.

Simpler devices, such as built in cameras and sound devices, will also need device allocators and device drivers and it is important that their use is restricted by using capabilities to restrict their use to authorised users/modules, but in a secure environment such as SPEEDOS offers, they should present no special problems.

Chapter 34
A Secure Internet?

At the outset of this chapter it must be clearly stated that there is no such thing as an absolutely secure Internet! And this remains true however many precautions are taken. One reason for this will be explained in section 9. In this chapter we explore mechanisms whereby SPEEDOS computers can be made much safer than is often the case for users of the Internet.

That current systems frequently fall victim to attacks from Internet hackers need hardly be said. But why? The heart of the problem is very simple to state. In order for a hacker to break into a system, he needs some way of executing his programs on the computer system which he is attacking. Why is this even possible? To answer this it is first necessary to have a basic understanding of the nature of the Internet.

1 The Basic Functionality of the Internet

There are several layers of activity and protocols involved in sending messages across the Internet. We now briefly examine those which are relevant to SPEEDOS.

1.1 Transmission Control Protocol/Internet Protocol

The Internet is a vast collection of interlinked computers which can communicate with each other by sending messages. The basic mechanism for doing this is called TPC/IP (Transmission Control Protocol/Internet Protocol)[207], which consists of two layers. The TCP layer takes a message intended for the internet and breaks it down into individual *packets*, and when these arrive at their destination the TCP layer at the receiving computer reconstructs the message from the individual packets. When it receives a packet from the TCP layer for actual transmission, the IP layer is responsible for actually sending this to the correct desti-

[207] see https://www.hostingadvice.com/blog/tcpip-make-internet-work/

nation.

There are two kinds of computers on the Internet: routers and hosts. Host computers are end-user computers. Routers are responsible for passing communications from one to another until their destination can be reached. We need not be concerned with exactly how this works, except to say that every computer has its own address, known as an *IP address*. These are unique addresses, which for some computers are fixed, while others are dynamically allocated; they identify a computer's network interface and provide the location of the host on the network[208].

Each packet contains the destination IP address and the number of the packet within the message; in addition it contains the packet content and other relevant information such as the IP address of the sending computer. But this alone is not sufficient to tell the receiving computer which of the many internet protocols the sender is using or which of its processes knows how to deal with the message, since the latter may provide several services. This brings us to the subject of ports.

1.2 Ports

In order that a receiving computer knows what to do with an incoming message, the message contains a *port number*[209], which is a 16-bit unsigned number. The port number is used to identify a specific service on the receiving computer and when a message arrives for a specific port the corresponding service is activated.

There are three ranges of port numbers. The "well known" (or system) ports are numbered in the range 0 to 1023. Then there are "registered" ports, which are numbered from 1024 to 49151; these are registered for use by specific users (especially firms). The use of these ports is controlled by the Internet Assigned Numbers Authority (IANA)[210]. The remaining ports (49152 and greater) are known as dynamic or private ports[211], which can be used by any Internet user.

1.3 Secure Transfers

The most common way of attempting to provide security for internet messages is via the Transport Layer Security (TLS) protocols, which are the successor of the Secure Sockets Layer (SSL) protocols and are jointly referred to as TLS/

208 see https://en.wikipedia.org/wiki/IP_address
209 see https://en.wikipedia.org/wiki/Port_(computer_networking)
210 see https://www.iana.org/numbers
211 For an extensive list of port numbers see https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers.

SSL.[212] These protocols are encryption-based and also incorporate integrity checks (i.e. techniques such as checksums used to establish whether during transfer operations an attacker has modified a packet) in addition to a digital certificate. The latter, which contains the server's public encryption key, confirms via a 'certificate authority' that the server is secure.

A secure connection is created via a 'handshake' between the client and the server, using the server's public key to agree on a symmetric key, which can then be used to pass messages via the secure connection.

1.4 Email Protocols

Emails are normally transferred between computers using the protocols SMPT (Simple Mail Transfer Protocol) and either POP3 (Post Office Protocol version 3) or IMAP (Internet Message Access Protocol).[213]

SMTP differs from the others in that it is used to send emails from an email client program to an email server (a computer which accepts, transfers, holds and/or sends emails). Such servers are typically on-line the whole time, in contrast with the computers of casual email users, which are often turned off at night, for example. Hence a user typically has his incoming email sent to an email server with which he is registered, and when he goes on-line his email program calls up incoming emails which have arrived at his server and thereafter at regular intervals until he goes off line again. The SMTP protocol, which uses port 25 (or port 465[214]), is responsible for managing incoming email messages to the point where they reach the destination server. Thereafter there are separate (alternative) protocols (POP3 or IMAP) which are used to transfer emails from the destination server to the client user's email program on his own computer.

POP3 (Post Office Protocol version 3) allows a client to collect his emails from his email server usually on port 110 (or for SSL/TLS encrypted emails via port 995). It also deletes the emails from the server when they are downloaded; however, some versions allow a user can specify a period of time before the emails are deleted.

IMAP (Internet Message Access Protocol) also allows a client to retrieve his emails from his server. The main difference between this and POP3 is that IMAP does not implicitly delete emails from the server. This makes it better to use in situations where a user regularly accesses his emails from more than one device. IMAP also allows folders to be created on the server and to be searched,

212 see https://en.wikipedia.org/wiki/Transport_Layer_Security, which provides details of TLS/SLL mechanism.

213 see https://www.jscape.com/blog/smtp-vs-imap-vs-pop3-difference

214 Port 465 is a 'secure' port which uses encryption; it is sometimes known as SMTPS.

etc. It uses port 143 by default and for a secure version uses port 993.[215]

1.5 HyperText Transfer Protocol

There are many protocols for sending messages across the Internet, the most popular of which is probably HTTP (HyperText Transfer Protocol), which is best described as a request-response system[216]. The normal HTTP port is port 80, while HTTPS port 443 is used for secure (encrypted) HTTP messages. A host computer sends an HTTP request across the network, and this is delivered to the computer with the corresponding IP address. The recipient then deals with the request and responds to it by sending a reply message over a port which is specified in the original message.

1.6 Domain Name System

In order to simplify the management of IP addresses, which are 32-bit numbers, a database called Domain Name System (DNS) exists which translates ASCII strings into IP addresses. This is used for example to translate email addresses and web page addresses into IP destinations.

1.7 World Wide Web

Websites are domains on the World Wide Web (www) which are addressed by URLs (uniform resource locators) that can be thought of as the addresses of individual web pages. A URL normally consists of an Internet protocol name (usually, but not always, *http* or *https*) followed by the symbols :// and then a DNS name (of the host computer), optionally followed by a / separator and the name of the web page to be displayed (e.g. https://www.jlkeedy.net/biography.html). If a URL is used without specifying a web page, then the start page of the domain is assumed.

When a client computer wishes to connect to a web page the browser of the client computer first obtains the corresponding IP address for the website from the DNS data base then connects to the requested port (e.g. port 80 for http) of the website (the server). When a TCP connection has been made the client then requests the specific page, which the server then supplies and the connection is released. When the browser receives the page from the server website, it displays this at the client computer. It can then typically obtain a further URL from the web page and activates this in the same way, and so on.

[215] see https://www.jscape.com/blog/smtp-vs-imap-vs-pop3-difference

[216] see https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

1.8 Hypertext Markup Language

The Hypertext Markup Language (HTML) is a standard language used for creating web pages. It describes how a web page can be displayed as part of a website by following instructions contained in tags which describe structural elements of the page, such as a page's title and subtitles, images and videos to be displayed, etc.[217] These may involve downloading further information such as files, images and videos.

Programs can be embedded in the HTML. These are written in scripting languages[218] (often interpreted rather than compiled, such as JavaScript[219]). They can be used, for example, to allow the user to interact with the web page.

The presentation of documents can be separated from their contents by using a style sheet language[220] such as CSS (Cascading Style Sheets)[221]. Using CSS technology the same content of a web page can be presented in different ways, e.g. different layouts, colours and fonts, and/or for different display devices, e.g. visual display units, smartphones, tablets.

1.9 The Cloud

Perhaps the best way to understand "The Cloud" is simply to regard it as the Internet. "Cloud computing" means accessing your data or your programs over the Internet. Large companies have developed business models (e.g. Software-as-a-Service[222]) in which businesses subscribe to application programs over the Internet (and pay fees for this). There are lots of business models which in effect mean that you put your data, your programs, your internet bandwidth and your trust in the hands of large companies, which make you pay handsomely for it. If you want to know more I suggest that you begin by reading the Wikipedia article "Cloud Computing"[223] (especially the sections "Security and privacy" and "Limitations and disadvantages") and also the PC Mag article "What is Cloud Computing?"[224] (especially the section "Arguments Against the Cloud").

This (oversimplified) description of basic Internet mechanisms should be sufficient to understand how SPEEDOS deals with basic Internet issues. The list of protocols described in this section is not necessarily complete (from the

217 see https://en.wikipedia.org/wiki/HTML
218 see https://en.wikipedia.org/wiki/Scripting_language
219 see https://en.wikipedia.org/wiki/JavaScript
220 see https://en.wikipedia.org/wiki/Style_sheet_language
221 see https://en.wikipedia.org/wiki/Cascading_Style_Sheets
222 see https://en.wikipedia.org/wiki/Software_as_a_service
223 see https://en.wikipedia.org/wiki/Cloud_computing
224 see https://uk.pcmag.com/networking-communications-software/16824/what-is-cloud-computing

viewpoint of relevance to SPEEDOS). Other protocols, such as FTP[225], may also be relevant, and can easily be fitted into the architecture described in section 3 below.

2 Browsers

Browsers are standard programs which are notoriously insecure but which play an essential role in conventional systems. Their main functions are to assist the user

- to activate internet requests which the users specify and
- to display the results of these requests,

with the aim of shielding users from the low level details of accessing the Internet. These two related activities, together with email, represent the main modus operandi of Internet use for most users.

2.1 Browsers and Malware

The main problem with browsers for normal computer users occurs in connection with the browser's second function, displaying the results of requests. This corresponds to the response stage of HTTP requests, sent in reply to a request from the user (which normally takes the form of clicking on text on the web page). It might for example involve displaying a text file or other file, e.g. a PDF file, or it might involve playing an audio file, displaying a video file or a spreadsheet, etc. The heart of the problem appears to be that browsers need access to a variety of programs to allow these files to be displayed and to implement this need they typically allow *plug-ins*[226] and other browser extensions to be used to supplement the functionality of the browser; for example different plug-ins provide the functionality to display the various kinds of file on the user's screen.

While this may appear to be advantageous (by allowing the browser to be customised and extended), it also has a dark side, since users who use plug-ins to customise their browsers are in effect placing their trust in the code of these plug-ins, which often have access to "sensitive data, such as browsing history, and have the ability to alter some browser settings, add user interface items, or replace website content... There have also been cases of applications installing browser extensions in a sneaky manner, while making it hard for the user to uninstall the unwanted extension".[227]

The plug-ins may include mechanisms which display *adware*, i.e. software which generates advertisements that appear alongside the intended information

[225] see https://en.wikipedia.org/wiki/File_Transfer_Protocol
[226] see https://en.wikipedia.org/wiki/Plug-in_(computing)
[227] see https://en.m.wikipedia.org/wiki/Browser_extension

display. Sometimes the adware may appear in a window that cannot be closed, which is particularly annoying.[228]

Cookies[229] are small files which are placed *on the user's computer* when he visits a website. They are included in the discussion of browsers because it is browsers which provide the facilities for websites to store cookies. Like plug-ins, cookies can have a useful purpose (e.g. they can store information allowing hosts to log in to a website, they can store user preferences, they can allow web-sites to personalise their content), but they can equally be used for malicious purposes (e.g. some cookies, known as trackers, can be used to track the sites which hosts visit on the web)[230]. In the words of PC World, cookies can "hide in your computer so that your browser and websites can track your browsing sessions and save certain useful information, such as account names and passwords, for later retrieval. Although cookies may seem harmless overall, they can threaten your privacy if an attacker tries to use them maliciously."[231]

Supercookies and *Zombie Cookies*[232] are pernicious extensions of the cookies idea which use storage space outside of the normal cookie storage in browsers to store cookie-like information, especially tracking information, often in multiple locations, to ensure that the usual browser cookie deletion mechanism cannot delete them. If an advertiser discovers that his tracker information has been deleted from a location he can restore this by copying it from another location.

2.2 Browsers and SPEEDOS

In principle the first function of browsers (the activation of internet requests) should be entirely superfluous in SPEEDOS, since the standard ways of accessing other SPEEDOS nodes (i.e. Internet requests) are as described in chapter 28 (i.e. via remote inter-module calls) and in chapter 29 (via download and upload operations). However, at least until SPEEDOS becomes widely used, many users are unlikely to willingly forgo the use of the millions of websites which already exist in the World Wide Web. Therefore section 4 describes a SPEEDOS alternative to conventional browsers, which can access websites.

An important ancillary service provided by browsers, maintaining "bookmarks", should similarly be completely unnecessary since in SPEEDOS terms these should be nothing other than module capabilities via which remote inter-module calls can be activated. Since these can be stored in normal SPEEDOS

[228] https://en.wikipedia.org/wiki/Adware
[229] https://en.wikipedia.org/wiki/HTTP_cookie
[230] https://www.howtogeek.com/119458/htg-explains-whats-a-browser-cookie/
[231] htts://www.pcworld.com/article/242939/how_to_delete_cookies.html
[232] https://www.makeuseof.com/tag/what-are-supercookies-and-why-are-they-dangerous/

segments (and therefore directories), there should be no need for special lists with separate list management facilities as are found in browsers. In fact this is a more natural alternative for bookmarks, since the capabilities can be stored alongside other relevant information related to a remote site or to the user's work, rather than separately in a browser. Furthermore SPEEDOS capabilities have the advantage over bookmarks that they contain access and other usage rights which enhance privacy and security. Nevertheless we consider in section 3 how "bookmarks" can be maintained for websites in a SPEEDOS context.

We therefore conclude that browsers in the conventional sense should not be needed in a SPEEDOS environment. But of course that is not the same as saying that existing websites and otherwise useful aspects of the Internet can simply be ignored.

In the following two sections we consider how websites and email designed for a SPEEDOS environment can function. Then we consider how users of SPEEDOS systems might take advantage of existing Internet facilities. Thereafter we look at the implications of providing Internet support for the SPEEDOS kernel and review protection aspects of the proposed design.

3 Implementing SPEEDOS Websites

In chapter 28 a mechanism was described whereby a user sitting at his own computer (the client side) could activate a "call back" module which might then invoke a remote inter-module call (RIMC) via a capability that gives access to a remote (server side) module, e.g. a website which has been designed as a SPEEDOS website. This can in return use a call-back call (CBC) to a routine of the client side call-back module (which is waiting for information to display, e.g. a page from the website)[233]. The user might then choose to initiate some activity enabled by the webpage (e.g. select a file to download or activate another web page). In SPEEDOS the way in which this is done is determined by the page design, which might, but need not, offer facilities that resemble those used in conventional browsers. The decision on how the display works is not limited to browser-like facilities such as a typical website download mechanism or selection of a URL. Rather the design is entirely in the hands of the programmer of the call-back module and its remote partner.

When the call-back call exits, it returns the user's request back to the website call-back module at the server side. The website module then examines the returned information, completes the required action(s) and issues a further call-back call to provide and as appropriate display the results. This pattern of call-

[233] How a SPEEDOS user can obtain a call-back module and a capability for this is described in chapter 35.

back calls continues until the required action is to terminate the session.

Hence a SPEEDOS website designer has a choice between using HTML (with or without JavaScript) or using a quite different technique.

SPEEDOS websites are discussed in more detail in chapter 35.

4 Email in SPEEDOS

Chapter 31 sections 5-9 described how a primitive email system could be developed in SPEEDOS environments simply by using SPEEDOS directories and capabilities. The basic principle is that *capabilities* for messages can be passed from a sending user to a designated mailbox directory of a receiving user. If he chooses, the receiving user can then use the capability to view (or copy/download) the message. For emails between users at the same node the mechanism is straightforward. If sender and recipient are at different nodes the sender of an email can acquire a capability for the public mailbox directory (PMD) of the remote node and thence the mailbox of the intended recipient, which he can then store for future use.

To simplify the discussion emails in current systems are referred to simply as "emails", whilst emails in a SPEEDOS system are designated as "S-mails".

4.1 Delivering S-Mail

Provided that a receiving user's node is on-line at the time an S-mail is sent to him, delivering the S-mail is not problematic and simply involves an inter-module call (local or remote) to the receiver's mailbox, but if it is not on-line the receiver's mailbox cannot be accessed. The solution in current systems is to use mail servers, which are permanently on-line (see section 1.4).

An alternative solution might be to delay the sending of the email until the node comes on-line. Is this viable? In some cases this would be acceptable, but not always. For example if a user in England is trying to send a message to a contact in New Zealand, there is a time difference of 13 hours, so that normal users will rarely be active at the same time in both countries. This would make the delivery and receipt of S-mails an irritating and uncertain activity.

But delaying the sending of messages is not the only alternative. Another possibility is to send the message in stages via intermediate SPEEDOS nodes. How might this look in practice? The answer is: very difficult, because the S-mail (a capability) is inserted into the mailbox via an (in this case remote) inter-module call, rather than being transferred as a file!

In fact the last point makes clear that S-mail, as so far described, is quite different from email in its mode of delivery. We now consider this in more detail.

4.2 S-mail by Remote Inter-Module Call or by Content

Current systems deliver mail by sending the content of the email, whereas S-mail, as described in chapter 31, uses remote inter-module calls to directories. But as we just saw, that can lead to a problem when attempting to deliver email to a node which is not on-line.

In fact this issue applies in principle not only to sending S-mail but also to the uploading of files (as described in chapter 29 section 3.2); otherwise the solution could have theoretically been to upload S-mail in such circumstances. But whereas under normal circumstances the uploading of files is unproblematic, because it usually takes place in the context of website activity (where both nodes are normally on-line at the time of an upload request), this is not the case with S-mail.

Chapter 28 section 8 introduced the idea of permanently on-line SPEEDOS *directory nodes* to allow a node's kernel network process to locate other nodes. Such nodes can also function as S-mail servers for the case that an S-mail destination node is not online. To achieve this efficiently, the sending node uploads a container which holds the content of an S-mail to the directory node associated with the destination node of the S-mail. If necessary this then further uploads it to another directory node, etc., until the directory node of the recipient has received it. When the destination node then comes on-line it not only advises its directory node that it is on-line, but also asks how many S-mails it is holding for the node. It then requests these to be uploaded one by one to the destination node, with details of the intended recipient. An email client node can also contact its directory node from time to time to check for the arrival of new emails (automatically in an enhanced system).

4.3 Appearance of S-Mails

Current email schemes produce an email with standard information and text. No such scheme was suggested for S-mails. However, the scheme outlined in chapter 31 is intended only as an outline proposal. If operating system designers feel it appropriate or necessary to imitate the standard scheme or to produce an alternative, this path remains open for them. However, it would be very regrettable if the basic idea of using ordinary SPEEDOS directories as the basis of an email scheme were to be abandoned, since this brings with it considerable simplicity in the implementation, and in addition it allows standard SPEEDOS utilities such as search facilities (which, as pure application programs, are not presented in this book but which are of course necessary) to be used on all directories, whether or not they contain S-mails).

4.4 S-Mail Security

Emails in current systems are not very secure. One issue is that by default they are not encrypted, though it is possible to use secure ports. This is not the case in SPEEDOS, since all SPEEDOS messages sent across the Internet (including the initiation of remote inter-module calls needed to place S-mail in remote directories, and the uploading of S-mails to SPEEDOS directory nodes) are encrypted.

A further problem with SMTP transfers is that there is no check on the legitimacy of the senders of emails. The result is that it is easy for spammers to send unwanted emails, some of which may be malicious. With the directory-based S-mail approach the basic mechanisms used not only always uniquely identify the sender of an S-mail but also make it possible for mutually suspicious users to be sure about the sender and the receiver of a message, as is described in chapter 31 section 7.

In order that this level of security is maintained where S-mail is sent via SPEEDOS directory nodes the message must be accompanied by the unique identifiers of both sender and recipient of the message.

4.5 S-mail Attachments

In a conventional email system one of the features often used is the ability to attach files to an email. Chapter 31 did not explicitly describe how this can be achieved in S-mail systems, but there are two obvious ways to do this.

First, each attachment could be entered as a separate entry into the receiver's mailbox, if appropriate giving it a name such as "attachment 1 to email x".

Second, the directory structure described in chapter 30 serves as an example, but should not be regarded as definitive. Any user (at least in a system in which the access rights are discretionary[234]) can define a directory system which suits his purposes (because he can define segments which include capabilities). This raises the possibility that a directory structure can be defined in which each entry has as an additional possibility for storing several capabilities (either in a fixed number of slots or as a linked list), the first of which is intended as the actual S-mail and the remaining for attachments.

5 A SPEEDOS Architecture for Managing Conventional Internet Activities

At this point we turn to the situation in which SPEEDOS users would like to take advantage of Internet facilities which are only available in a non-SPEEDOS environment. This approach is based on two principles. First, the kernel should have only a minimal involvement in the activity, and second, measures should

[234] see chapter 2 section 3.6 and chapter 36.

be adopted to ensure that a high level of protection is provided to safeguard both the rest of the SPEEDOS system and the module/thread which carries out the internet activity itself.

The basic idea is based on the normal SPEEDOS in-process/inter-module call mechanism. Some user threads are authorised to access certain non-SPEEDOS protocols. The permission to access a particular protocol or group of protocols is based on the possession of a capability authorising the user to access one (or more) of a restricted set of *System Internet* modules, which serve as a protected environment, in Internet jargon a "sandbox"[235] (see Figure 34.1).

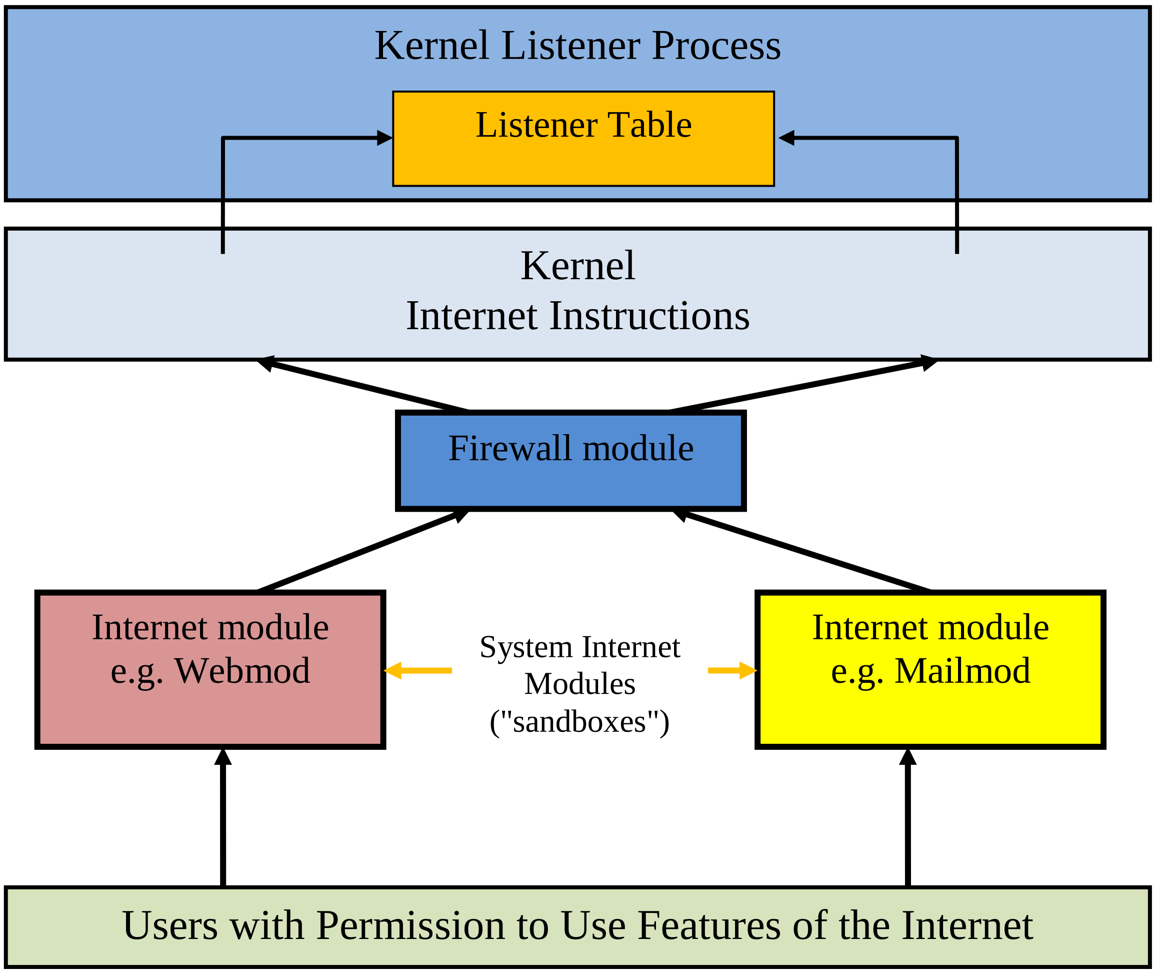


Figure 34.1: A SPEEDOS Internet Architecture

These have all the SPEEDOS protection mechanisms at their disposal to isolate them from the rest of the SPEEDOS system. Such modules can for example be severely restricted by bracket routines, as will be described below. These requests are then passed on to a **`firewall`** module, which has a separate interface routine for carrying out separate checks on requests for the various protocol groups supported. These routines also examine and if necessary block the

[235] see https://www.computerhope.com/jargon/s/sandbox.htm

results from the Internet when a response to a request is returned back through the firewall.

Figure 34.1 illustrates this arrangement by two example modules, **`webmod`** and **`mailmod`**, which are described in more detail below, allowing users to access non-SPEEDOS websites or mail servers. Such modules will always be singleton file modules which register details of users, including their unique 192-bit SPEEDOS identifiers. These modules can also note users' preferences, maintain other relevant security information and prepare (e.g. by calling the DNS database) and carry out individual user Internet requests.

If all is well, the firewall uses one of the kernel instructions provided to support the different protocol groups. The kernel issues the appropriate Internet request, causes the user thread to be suspended and passes a message to the kernel's listener process to listen for a reply.

When a reply arrives, the listener passes this back to the suspended thread and causes the latter to be re-activated. The reply then gets passed back to the **`firewall`** module, which carries out its checks; if all is well it then returns the reply back to its caller (e.g. **`webmod`** or **`mailmod`**) which may carry out further (e.g. user related or protocol specific) checks and where appropriate displays the results (e.g. by interpreting HTML) on the user's monitor. In this sense the System Internet modules serve a function similar to that of browsers and/or mail programs.

This general architecture can be used to define any approved Internet accesses, while the details may vary from case to case. In the following we first explain the two examples (websites and email). Then we provide further details of the kernel mechanisms and consider the protection aspects in more detail.

6 Accessing non-SPEEDOS Websites

In current systems websites work hand-in-hand with browsers, which display their results, returned in the form of HTML. This can introduce security risks. In particular it is possible to insert interpreted code into a web page, e.g. using Java Script. In the words of the 'Computer Hope' website, "Because JavaScript is downloaded from an unknown origin and executed on your computer, JavaScript could have the potential of being a virus or doing other malicious things to your computer."[236] This risk should clearly be avoided in SPEEDOS. The ideal situation, from a SPEEDOS viewpoint, would be for all websites to be designed as SPEEDOS websites. But that is obviously unrealistic.

The **`webmod`** module offers an interface routine which makes contact with a

[236] see https://www.computerhope.com/jargon/j/javascript.htm

conventional website. As its main parameter this has a character string in the form of a URL (see section 1.7) such as *https://www.jlkeedy.net/biography.html*. Like other SPEEDOS modules this is called using a normal (local) IMC.

Once activated, **`webmod`** establishes the website's IP address by activating a request to the DNS database and subsequently uses this to request a web page, using the appropriate protocol (e.g. http or https) indicated in the URL.

When the webpage arrives, **`webmod`** obtains the device capability for the user's screen in the usual way and then creates a new window by calling the graphical library routine **`createWindow`** (see chapter 32 section 4.4.3.3). It then reads the HTML from its buffer, using a reliable HTML interpreter to display the results on the user's new web page window. Thereafter it reacts to mouse clicks until the user activates a further URL embedded in the HTML. In this case the **`webmod`** module services this call in the same way as the initial URL. When the user closes its last window, the **`webmod`** module exits.

6.1 Cookies

In principle a cookie is nothing more than a small file. The risks arise because unscrupulous websites can use them in an uncontrolled way and because they are not normally visible to the user on the computer which hosts them. They can for example store personal information about a user, garnered from the use of their own website and sometimes from other websites. They can also be used to introduced viruses and other malware (e.g. spyware) into the host's computer.

Why then are cookies tolerated? The primary answer[237] is that websites need to store information during a user session to help them to relate requests to each other. For this reason cookies hold a "session id" and are passed backwards and forwards between user and website server with every HTTP request over the course of a session. This is necessary because the Internet as such (and HTTP) is stateless. The length of a cookie is limited to 4 KB to keep the level of traffic on the Internet reasonably low. A domain can have up to 20 cookies, which can be read and modified at both the server side (i.e. the website software) and the client side (i.e. the browser).

Cookies are managed at the client side by JavaScript (in conjunction with the local browser). Each cookie has an expiry date, which can be set in the HTML. If no date is set then it is deleted when the browser closes. But cookies can outlive browser sessions; this means that a website can leave data on the client side, which might for example contain registration details for the website.

These reasons for cookies make good sense, and to allow SPEEDOS users

[237] see https://flaviocopes.com/cookies/

to take advantage of non-SPEEDOS websites they should be supported by the **`webmod`** module. But the problem is that in conventional systems there is no control over what is stored at the client side, and the owner of the client node cannot examine this, although his browser will probably delete cookies on request.

It is therefore suggested that cookies be maintained in **`webmod`** as small SPEEDOS files associated with the **`webmod`** registry, and that transparency is achieved for end users in that an interface routine is supported which allows the registered user to read (but not write) cookies, providing him with a capability for the cookie with the access rights set for searching, for reading and for deleting the cookie. However he should not be given write access to the cookie, since this would enable him to cheat websites. It is of course part of the job of **`webmod`** to attach the content of a cookie to each user's webpage request and to update the cookie when a response is received. The system can set Thread Security Register restrictions and add call-out bracket routines to ensure that cookies are not used as platforms for launching malware.

7 Email Programs

Emails have become a standard mechanism for communicating between users of the Internet and must of course be provided in SPEEDOS systems.

7.1 Current Email Programs in Current Systems

In current systems email has developed into a quite sophisticated system, organised by application programs which deliver the email to users and assist them in receiving email, sorting it into email folders, etc. In my view such programs are unsatisfactory in the sense that they duplicate many functions of an operating system and thus make it difficult to integrate emails (which are nothing other than files uploaded and downloaded between users) into the general work of the user. Put simply a user normally places all the (non-email) documents relating to a particular event or task, etc. into a single directory (possibly with further subdirectories), but because email is organised as a separate activity with its own filing system one ends up with parallel but formally unrelated directory systems. Furthermore general purpose utilities, such as search programs, often cannot be used in the context of the data associated with email programs.

7.2 Using Conventional Email in SPEEDOS Systems

Presumably SPEEDOS users will want to communicate by email with non-SPEEDOS users. This can be implemented as an option, but those who use the mechanism which I now present should be aware that in doing so they open themselves up to risks which do not arise normally in SPEEDOS systems and therefore that wherever possible they should use S-mail.

Rather than using a conventional style email program, the intention is to provide a simple mechanism which simply allows conventional mail to be sent and to be received by SPEEDOS users. Its aim is *not* to provide a separate management system for mails by providing its own folder system[238], etc.

7.2.1 A SPEEDOS System Internet Mail Module

A SPEEDOS mail module, **`mailmod`**, needs three basic semantic routines.

a) A **`mailRegister`** semantic routine, which allows a user to apply for registration with **`mailmod`**. This accepts as parameters:

   – a string containing the email name (i.e. the part preceding the @ symbol) to be used by the sender; and

   – a capability for a directory into which arriving emails should be saved.

   After carrying out appropriate checks (e.g. that the thread security register permits external emails or, in a mandatory system, that the email name conforms with conventional standards, that the user meets criteria set by the system manager), the identifier of the calling thread's owner is established by calling the kernel instruction `current_thread_owner` (see chapter 26 section 1). The email name and the unique identifier of the user (i.e. the value returned from the kernel call) are entered into a list for use when a user attempts to send or receive emails. A user can register multiple email names.

b) The **`sendMail`** semantic routine accepts as parameters

   – a text file containing the body of the mail;

   – a list of capabilities for attachments to the mail;

   – a string containing the email name used by the sender; and

   – a list of strings containing the email addresses of the recipients.

   It creates an email in standard conventional form from this information and sends it to the recipient, using the SMTP protocol.

c) The **`receiveMail`** semantic routine accepts as parameters

   – a return parameter indicating the number of attachments received;

   – an indication how many further emails are waiting for the user.

   It accepts emails in standard form from a normal email server (using POP3 or IMAP) and converts each email which it has received into a SPEEDOS file, then places a capability for this in a directory provided (as a capability)

[238] When conventional email systems provide their own folder management system it becomes difficult for users to integrate their emails with related work and it is often also impossible for users to use other software (e.g. search applications) for their emails. It is also wasteful to provide parallel folder management systems.

by the user when he registers his email name with the mail program.

8 Kernel Mechanisms for Accessing the Internet

This section describes how the kernel handles Internet requests (including those from the network process, from the **`mailmod`** module and from the **`webmod`** module).

8.1 Handling Requests from the Network Process

The kernel network process at a SPEEDOS node (see chapter 28) sends and receives encrypted messages via the Internet to and from other SPEEDOS nodes. For this purpose it uses two as yet undefined ports, but the encryption is carried out not as part of the port definition but as described in chapter 28. It uses asymmetric encryption to allow 2 nodes to agree on a common symmetric encryption key, or for short messages it simply uses the asymmetric keys, using the TCP/IP transfer protocol.

8.2 Listening for Messages from another SPEEDOS Node

At each node a kernel Listener process listens continuously for messages from other SPEEDOS nodes, and when it detects one, it checks the authenticity of the message. Authentic messages received on the SPEEDOS receiving port are transferred by the listener process to the network process, using the normal inter-communication mechanism provided by the SPEEDOS kernel's process scheduling mechanism (see chapter 22 section 7).

8.3 The Listener Mechanism

The Listener process is the lowest priority kernel process, i.e. the process which in chapter 22 was called the 'idle' process. This listens continuously on the incoming SPEEDOS port for the arrival of SPEEDOS messages but also on a round robin basis for other messages which it is expecting. The latter are listed in its Listener Table, which contains a list of all the expected message arrivals.

Entries in the Listener Table hold the number of the port on which the message is expected, a further identifier to allow it to be distinguished from other messages arriving on the same port and a thread capability for the waiting thread. The latter is used to re-activate the waiting thread when a reply is received.

All other ports are kept closed, since the SPEEDOS Internet facilities only *request* information from other Internet sites (except other SPEEDOS nodes).

9 Protecting SPEEDOS and Its Users from the Internet

In this section we describe some precautions provided to help protect SPEEDOS and its users from the usual Internet problems.

9.1 Kernel Instructions

To communicate with the Internet the **`webmod`**, for example, executes the privileged kernel instruction `web_request` (or other correspondingly named kernel instruction). Its constant segments hold a kernel capability which allows it to use this instruction. (By providing a separate kernel instruction for each protocol group, more protection is achieved, and individual instructions are kept simple.) In the case of `web_request` this takes as operands the IP address, the port number and the requested webpage. This instruction causes the thread to be suspended awaiting a response from the website. When the response arrives, the listener process passes the HTML page back to the requesting thread and causes the thread to be reactivated.

The module **`mailmod`** uses two similar kernel instructions `mail_request` and `mail_send` in order to request and to send emails, which are also protected by two (separate) kernel capabilities. In both cases the calling program must supply an IP address and the port number. In the case of `mail_send` the kernel expects as a further operand the mail to be sent. And in both cases the calling thread is suspended. When the corresponding replies are received the kernel makes these addressable and causes the waiting thread to be activated.

Kernel capabilities for kernel Internet instructions are only issued (in the constant segments of the program code) to the respective modules. Notice that in a multi-user system each user could be provided with a separate instance of the file module, but the mail *registration* module should a singleton module (because the list contains entries for multiple users), a capability for which could be embedded in the individual file instances of the users.

The kernel instruction `dns_request` can be used by these modules to access the Domain Name System database.

9.2 Managing the Lengths of Messages Received over the Internet

The length of the information returned from an Internet request is variable and might be quite long; the actual length is returned to the user at a fixed position in the segment addressed by segment register 15, which holds the operands for kernel instructions (see chapter 17 section 6). To avoid the inefficiency of transferring long messages as return parameters from the kernel or from one inter-module return to another the following mechanism can be used.

For all returns from calls which are Internet-related (e.g. calls from modules such as **`webmod`**, **`mailmod`** and **`firewall`**) and for related kernel instructions (e.g. `web_request, mail_request`, `dns_request`) segment register 14 is used to address a large segment (created by **`webmod`**, **`mailmod`** and similar) which is used as a buffer by the listener to accept Internet messages for the thread concerned. The access rights of segment register 14 are set to *read-only* by the listener after reading in the message. Segment register 14 is stored and reloaded as with normal thread switches, but on inter-module calls and returns this register remains unchanged by calls, returns and kernel instructions if a special code is set at the bottom of the thread's stack. There is a protected kernel instruction `fix_sr_14`, which sets or unsets a code (depending on the setting of a boolean parameter for the kernel instruction). The code is checked as part of calls, returns and kernel instructions to determine whether segment register 14 should be invalidated or not (in contrast with the normal IMC mechanism which invalidates all segment registers except registers 0 and 1 on inter-module calls (see chapter 20 section 8.1).

9.3 Security Measures

All the SPEEDOS protection mechanisms (e.g. module capabilities with restricted access rights, the thread control register, environmental control instructions and bracket routines) can be used to safeguard the use of the Internet.

But above all, the listener keeps only the necessary ports open for accessing mail and websites and special (protected) kernel instructions are used to access the Internet from outside the kernel. It ignores any messages on other ports.

We now look at some specific examples of how security measures can be taken.

9.3.1 Protecting Access to the Internet

Leaving aside the kernel processes themselves, users can only gain access to the Internet via System Internet modules such as **`mailmod`** and **`webmod`**. Access to these programs in each case requires an appropriate capability, which is initially under the control of the administrator of a node[239]. He can determine which other users, if any, can obtain a copy, and these must all call the **`firewall`** module to access the Internet. They obtain a copy of the necessary capability for calling **`firewall`** from within their own constant segments. These capabilities have individual access rights to different routines of **`firewall`**, tailored according to the needs of their own functionality. Only **`firewall`** has a kernel capability (embedded in its constant segments) which gives it the right to execute the specific kernel Internet instructions.

The code capabilities for the System Internet modules can be used as a basis for creating "file" instances (see chapter 19 section 7), which can contain persistent data that records information such as registration details of users. (In reality only a singleton module is required for some Internet applications, in

[239] At a single user node this is of course the user who owns the node.

which case normal users only need a capability for the appropriate singleton module.)

9.3.2 Thread Security Register Settings

To cover the use of the Internet the TSR includes thread control rights allowing its use (see chapter 26 section 4.1). These are separate rights for calling websites, mail, FTP and other Internet operations), which are checked by the kernel on inter-module calls but also can be checked by the registration module and by the kernel instructions. *There is one special rule*: if a System Internet module is called and the thread is neither already registered with its Internet activity group nor is currently being registered, that System Internet module unsets the corresponding rights in the TSR. Thereafter that group of Internet operations cannot be used by the thread.

The effect of this rule is that only registered threads can access the Internet.

9.3.3 Bracket Routines

Bracket routines (which from the standpoint of Internet activity can be thought of as private firewalls for individual modules) can be used by the system for a number of purposes, e.g. as a revocation list, which holds the unique identifiers whose right to use the Internet have been revoked.

Another use of bracket lists could be to hold a list of website URLs; it could contain a list of websites etc. which have been disallowed because they are known to be dangerous[240]. This would be applied as a set of call-in bracket routines which check the parameters supplied to **`mailmod`** or **`webmod`**, etc.

Possibly the most important form of bracketing would be the use of call-out brackets on the System Internet modules and **`firewall`** to ensure that if these modules are in some way penetrated, the attacker could not activate new modules of its own devising or penetrate other SPEEDOS modules. The output returned to a caller of these modules should also be checked by call-in brackets to ensure that capabilities, for example, could not be passed.

There are certainly several other uses to which bracket routines could be put, but there is one problem which they cannot directly solve, i.e. checking the results which are returned from the Internet, because this involves executing kernel instructions rather than making inter-module calls. We now consider this issue.

[240] A mechanism could be devised to allow SPEEDOS nodes to share information about dangerous websites, etc.

9.3.4 Checking Information from the Internet

Information from a website comes in 2 basic forms: as HTML or as a downloaded file. It is of course important to check that such information does not have dangerous content, such as viruses and the like. It is not the function of the kernel to carry out such checks. Rather, this is primarily the responsibility of the **`firewall`** module.

9.3.4.1 The Firewall Module

The **`firewall`** is responsible for checking all internet content which arrives from the normal Internet (excluding transfers between SPEEDOS nodes), including cookies.

The interface routines used by the System Internet modules to call the **`firewall`** module correspond to the kernel's Internet instructions, i.e. there is an interface routine **`webrequest`** (corresponding to the kernel instruction `web_request`), an interface routine **`mailrequest`** (corresponding to the kernel instruction `mail_request`), an interface routine **`mailsend`** (corresponding to the kernel instruction `mail_send`), etc. These routines accept the corresponding Internet requests, carry out checks on them and if these checks are successful, each routine uses the corresponding kernel instruction to activate the Internet. When it returns, it uses segment register 14 to address the response from the Internet and to check its content. If all is well it returns to its caller in the normal way. However, if it detects a serious problem it raises an error exception.

Perhaps the most serious risk is that JavaScript could be used in an attempt to attack the rest of the system. It is therefore especially important that checks should be included in the firewall module, but also in the modules which interpret JavaScript to ensure that any such attack can be contained, e.g. by the use of call-in and call-out bracket routines.

One reason for placing the firewall outside the kernel is that from time to time it will need to be updated. One possibility for simplifying the update activity would be to put the updates into a file which it accesses as part of its work. To reduce the risk that this file is broken into, a capability for the file should be embedded in its own segments (e.g. a constant segment) rather than being held in a more accessible location. Another capability could be held at a site responsible for the update, and via this changes could be made to the file. (Such an arrangement would of course have to be synchronised.)

10 Search Machines and Similar

The issue of search machines has been left to last, because the issue of accessing search machines appears to be quite straightforward. A search machine is simply a website which returns information about other websites. Consequently the website mechanism proposed in section 6.1 (i.e. using **`webmod`**) can also be used to gain information on the existing Internet about other websites which a SPEEDOS user might wish to access. Similar considerations apply also to other commonly used websites, e.g. for on-line shopping, for accessing social networks, etc. The next chapter discusses search machines for SPEEDOS.

11 Concluding Remarks

As was stated at the beginning of the chapter, there can be no guarantee of ever reaching complete security when accessing the Internet. The reason for this is that there is no way of preventing *denial of service* attacks. A denial of service attack occurs when an attacker deliberately tries to overwhelm an Internet node by transmitting huge numbers of Internet requests to that node, beyond its capacity to deal with them[241]. What is particularly pernicious about some such attacks, especially if they are initiated by Internet bots[242], is that the attacker may target several IP addresses which in effect by sheer volume of messages then clog up the same or related routes to the target nodes; this can in the end affect not only the target(s) but other 'innocent' nodes which use the related IP address ranges.

Can SPEEDOS withstand such attacks? In the end the best that it can hope to do is in effect to turn off the Internet by closing all Internet ports. One way the listener process could do this is by observing the amount of incoming traffic and when it reaches a suspicious level (or a higher rate than it can cope with) and closing down the ports, and only reopening these when the attack has subsided. Fortunately the servicing of Internet calls by the listener is the lowest priority task of the kernel processes, so that this will only affect those processes which are dependent on the Internet, while the rest of the kernel should continue to function normally.

Finally, this chapter should be regarded more sceptically than those chapters which describe other parts of the SPEEDOS design. The reason for this is that I have in the past never been deeply involved in designing Internet software. Hence I have ignored certain more advanced features, such as virtual private networks (VPN) and remote desktop (team viewing). How these can be safely integrated into the SPEEDOS architecture would make good topics for PhD students.

I hope that the suggestions about mapping Internet interactions onto the SPEEDOS architecture will nevertheless prove to be helpful.

[241] see https://en.wikipedia.org/wiki/Denial-of-service_attack

[242] see https://en.wikipedia.org/wiki/Internet_bot

Chapter 35
Secure Website Applications

The previous chapter outlined how SPEEDOS users can access the conventional Internet, hopefully in a more secure way than is possible using current operating systems. However, the techniques presented there are intended as an intermediate solution which will allow users to convert conveniently to SPEEDOS systems. In the longer term it should become possible for SPEEDOS users to confine their activities to applications which are based entirely on SPEEDOS concepts and remain within its protection bounds, without the need to access mechanisms and software not designed especially for SPEEDOS. This chapter describes in further detail how SPEEDOS can support websites which are designed especially to use its protection mechanisms rather than the rather insecure mechanism available to conventional Internet users.

1 The Basic SPEEDOS Networking Mechanisms

The SPEEDOS kernel supports four basic networking mechanisms:

a) The remote inter-module call (RIMC), which allows a user to activate a module that is located on a different node[243].

b) A call-back call (CBC), which allows a remotely active module to "call back" an entry point routine of the module which activated it (i.e. at the original node)[244].

c) The downloading of a module from a remote node (which is managed by the Container Manager with the assistance of the kernel)[245].

d) The uploading of a module to a remote node (which is managed by the Container Manager with the assistance of the kernel)[246].

The most significant of these for supporting SPEEDOS websites are the RIMC

243 see chapter 28 section 3.
244 see chapter 28 section 7.
245 see chapter 29 section 3.1.
246 see chapter 29 section 3.2.

and the CBC, but the websites can of course offer the downloading and uploading facilities to their users, as was described in chapter 29 section 3.

2 How a SPEEDOS Website Operates

SPEEDOS websites rely on the existence of a related call-back module at the computers of their users. This plays a role similar to that of browsers in conventional systems. However, as was discussed in chapter 34, browsers are the cause of much insecurity in current systems. An ideal alternative is to dispense with browsers entirely and replace them with SPEEDOS call-back modules, as was suggested in chapter 34 section 4.1. We begin by outlining this approach.

2.1 Using Custom-Built SPEEDOS Call Back Modules

In this scenario a user of a website, at the client node A, activates a thread T1 which has a capability for a *website call-back* module at the *client* node. This makes a normal local IMC to activate the call-back module (at node A). The latter has an embedded capability which allows it to make a remote IMC to the website module at node B.

In the course of initialising the call-back module at node A, thread T1 obtains and stores capabilities for the user's monitor, mouse and keyboard. It may then display a standard start page. At this point the user might request a further page. In this case the call-back module issues a remote inter-module call to the website at node B, using a capability which is embedded in its own segments. This activates a surrogate RIMC partner thread T2 in the website module.

At the remote node B the partner thread carries out any preliminary tasks such as ensuring that the caller is registered. It then prepares the requested web page information and uses a call-back call (CBC) to return this to the call-back module at node A, causing the initial thread at node A to be reactivated.

The call-back module will then display the new information on the user's screen. The user might then activate a further request, in which case the call-back module requests the appropriate information in its return parameters and exits back to the website RIMC thread.

In a further call-back call the website module provides the necessary information to fulfil the user's request and the call-back module displays it. This activity pattern is repeated until the user signifies that he wishes to close the session. The call-back module then exits from the call-back with a return parameter indicating that the session is to be closed. The RIMC then makes a normal return back to the call-back module and the surrogate thread is deleted in the normal way. The thread in the call-back module at node A then also exits and the session is closed. Normally the thread will then log out until it is needed at some future time. This activity pattern is illustrated in Figure 35.1.

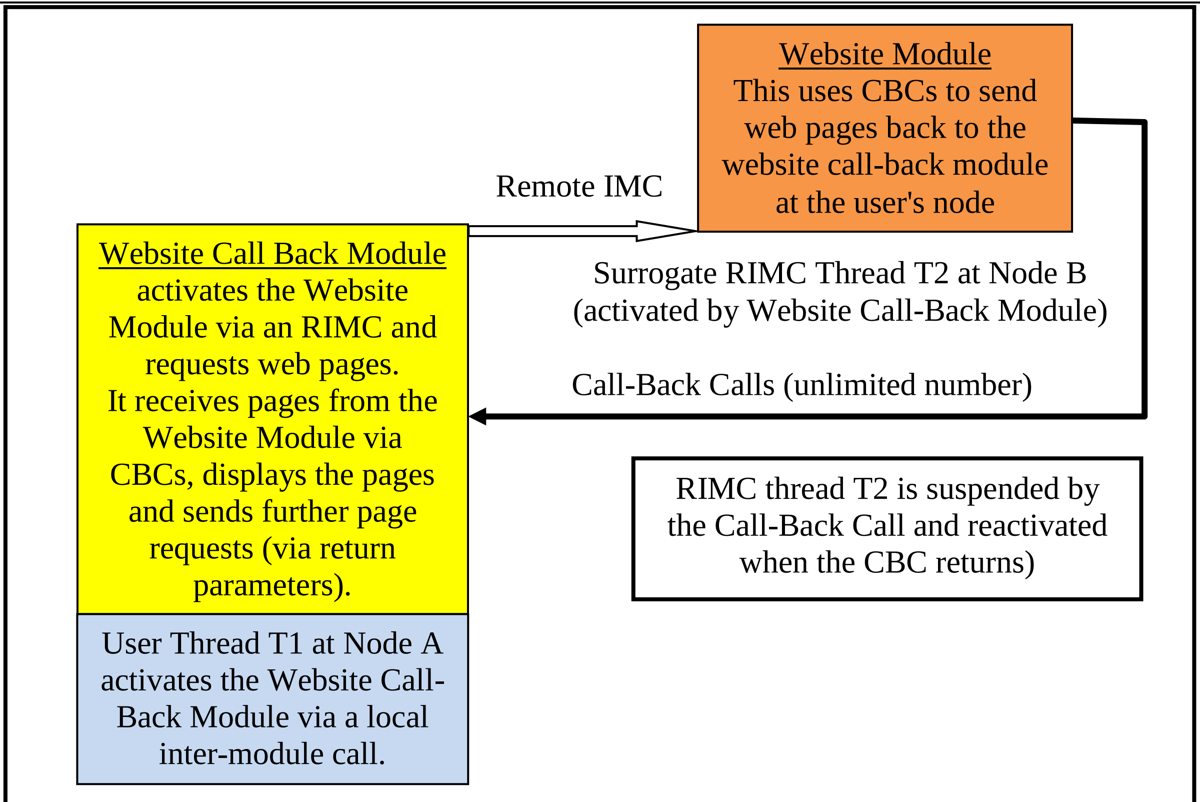

Website Module
This uses CBCs to send web pages back to the website call-back module at the user's node
Remote IMC
Website Call Back Module
activates the Website Module via an RIMC and requests web pages.
It receives pages from the Website Module via CBCs, displays the pages and sends further page requests (via return parameters).
Surrogate RIMC Thread T2 at Node B (activated by Website Call-Back Module)
Call-Back Calls (unlimited number)
RIMC thread T2 is suspended by the Call-Back Call and reactivated when the CBC returns)
User Thread T1 at Node A activates the Website Call-Back Module via a local inter-module call.

Figure 35.1: Custom-Built Call Back Modules

The organisation behind the displaying of web pages is an internal matter determined by the programmer(s) of the website module and of the call-back module. SPEEDOS does not define how this works, leaving the programmer of the website free to determine, for example, whether HTML is used or whether some other technique is applied, e.g. using preformatted pages held in the call-back module.

This arrangement is particularly suitable for websites which a user will frequently visit (e.g. an internet banking facility, the employer's website, a bookshop website, a shopping website, etc.). It presupposes that before the website can be accessed the user must not only have a capability for the call-back module but also a copy of the module itself. Otherwise he must somehow obtain these. This can occur in several ways.

One possibility is that he buys the call-back software or obtains it free. In this case he might use a CD or a memory stick supplied by the website owner or bought from a software shop, which he then uses to install the call-back module.

Alternatively he might download the software from the website. But how he might obtain a capability for this? Before answering this question we consider how more casual visits to a website might be organised.

2.2 Using Standard Call-Back Modules with Library Routines

The architecture described in section 2.1 is suitable for accessing websites which are frequently visited by a user, but not for accessing websites which are used as "quick lookup" websites, such as encyclopaedias, dictionaries, calendar and international time zones, etc. These are the websites which one normally expects to access quickly but infrequently.

For such websites each SPEEDOS node can offer a number of standard, pre-prepared library routines which use standard display strategies such as HTML[247]. Unlike browsers the SPEEDOS library module does not offer the dangerous possibility of plugging in new software.

The library routines offer functionality which can display and download (SPEEDOS conform) PDF files, play videos, etc. Like all SPEEDOS modules and library routines these are rigorously tested in advance to ensure that they perform the correct functionality required of the individual module according to their specification (and nothing more!)[248].

To find and quickly access other SPEEDOS websites a search machine module is needed. This might crawl websites, just as in conventional systems, but an alternative is suggested in section 2.4. The search machine itself should be developed with its own pre-installed call-back module to help users with their searches. When the search machine displays a page, this will offer a menu of webpages which attempt to meet the search criteria. Associated with each webpage listed there will be a webpage address (in an as yet undefined SPEEDOS format[249]) and a *website capability*/page number pair.

If the user selects a page to be displayed from the search machine's menu, the search machine module prepares parameters for a new subthread, including the page number and a capability for the website's main module[250]. It then creates a subthread[251] by calling the Thread Manager.

When activated the subthread locates its parameters and calls the selected standard call-back module, which then makes an RIMC call to the website module. This in turn locates the requested page and prepares it for display, if necessary using the library routines. It then makes a call-back call to the standard call-back module, which displays the page in a separate website window. The user can then use this page to select further pages of the same website, which the call-back routine passes to the main website module, etc. This procedure contin-

247 The HTML call-back should not be confused with **webmod**, which was described in the previous chapter, although the two could use common library routines, e.g. for interpreting HTML and JavaScript.

248 see chapter 26 section 6.1, section 6.22 and chapter 38 section 3.

249 perhaps a node number and module number?

250 see chapter 31 section 2.6 for passing parameters to subthreads.

251 see chapter 20 section 5 and chapter 31 sections 2.5. The reason why subthreads are used is to allow the search machine to manage multiple requests.

ues until the user closes the website window, in which case the call-back routine signals the search machine call-back module and exits, causing the subthread to be deleted.

Note that the user may leave windows open while he searches for other websites. When all its windows have been closed the search machine itself can exit. This solution is illustrated in Figure 35.2.

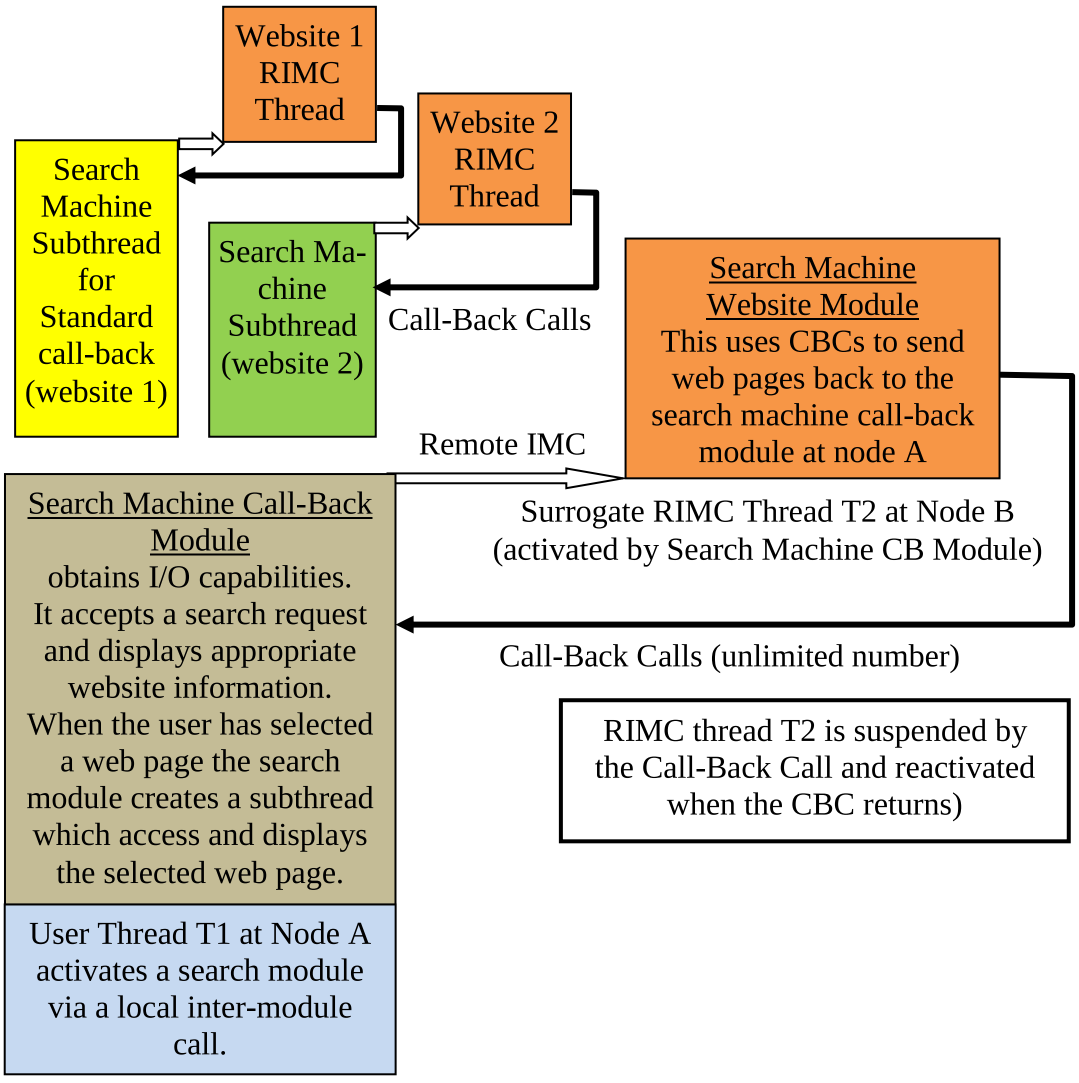

Website 1 RIMC Thread
Website 2 RIMC Thread
Search Machine Subthread for Standard call-back (website 1)
Search Machine Subthread (website 2)
Call-Back Calls
Search Machine Website Module
This uses CBCs to send web pages back to the search machine call-back module at node A
Remote IMC
Search Machine Call-Back Module
obtains I/O capabilities. It accepts a search request and displays appropriate website information. When the user has selected a web page the search module creates a subthread which access and displays the selected web page.
Surrogate RIMC Thread T2 at Node B (activated by Search Machine CB Module)
Call-Back Calls (unlimited number)
RIMC thread T2 is suspended by the Call-Back Call and reactivated when the CBC returns)
User Thread T1 at Node A activates a search module via a local inter-module call.

Figure 35.2: A Search Machine Environment

In the unlikely case[252] that the SPEEDOS library routines cannot display or otherwise process a request, a message will be displayed to this effect.

2.3 SPEEDOS Bookmarks

A SPEEDOS website bookmark is simply a capability for a SPEEDOS website or for its call-back module. Hence bookmark software is in principle simply di-

[252] This should not happen because we are considering only SPEEDOS websites in this chapter. The designers of SPEEDOS websites will be aware of the constraints imposed by the library.

rectory software. If it is felt necessary to develop special software for bookmarks this simply involves extending a normal SPEEDOS directory.

When a user activates a SPEEDOS search machine he passes a capability for a directory module as an input parameter. The search machine can then record capabilities and names for all the websites which it visits. In this way when an Internet session completes the user can access the directory (for which he has retained a capability) and distribute appropriate entries to other files and directories. In this way Internet access can be fully integrated with the user's other work.

This approach to bookmarks also answers the question which was posed at the end of section 2.1, i.e. he can obtain the first capability for a custom-built website (e.g. for his favourite bookshop) by using the mechanism described in section 2.2. In this way he can then take advantage of an offer from the website to upload the necessary call-back module.

2.4 Must a Search Machine crawl?

Because we are concerned in this chapter only with SPEEDOS-conform websites, it would be possible for new SPEEDOS websites, when they want to go on-line, to provide the necessary information to a distributed SPEEDOS database of information and to provide other SPEEDOS nodes with access to this information on request, including providing a capability giving appropriate access to the semantic routines of the website. This could be organised in conjunction with the idea of "directory" nodes briefly described in chapter 28 section 8.

3 Conclusion

The proposals in this chapter not only describe how SPEEDOS-conform website activity might be implemented but they also by implication explain how many of the dangers of the conventional Internet might be eliminated, in particular those associated with browsers, including for example how all the commercial espionage software, such as the tracking of users' use of the Internet, can be avoided.

As in the previous chapter the present chapter should be regarded more sceptically than those chapters which describe other parts of the SPEEDOS design. The reason for this is that I have in the past never been deeply involved in designing Internet software. I hope that the chapter will nevertheless be helpful.

Chapter 36
Mandatory Access, Rule Based Systems and Computer Administration

As was explained in Volume 1 part 1, there are two quite different approaches to viewing security in a computer system: the libertarian view (which some might perhaps see as the anarchist view) and the authoritarian view (which others might perhaps see as the dictatorial view).

In the variously coloured security criteria which have appeared since the Orange Book in the early 1980s (see chapter 1), these two viewpoints are usually described as "discretionary" and "mandatory" access controls. Not surprisingly the coloured books all favour the mandatory view. I say "not surprisingly" for two reasons. First, these documents are the work of military and/or government departments. Second, the original formulations of the security criteria stem from the age of mainframe computers, computers which were (and still are) so expensive to buy and to run that they can only be purchased by the military, by government departments and by business and commerce.

But since their heyday the computing scene has radically changed. Today mainframe computers are increasingly thought of as the dinosaurs of the computer revolution. In the last three decades the microprocessor has made it possible for virtually every household to possess its own personal computer(s), tablets and smartphones, and these have become as indispensable as the home telephone or television.

When personal computers first appeared, it seemed that security problems were not a serious worry for them. However, it has become increasingly evident that this is not the case. As PCs grow in power and storage capacity, they are already individually more powerful than many mainframes of the past. This means that they are being increasingly used to hold sensitive data and are therefore increasingly targets for penetration by hackers. And in the last two decades the Internet has taken off in a spectacular way. It is now normal that virtually every personal computer, tablet and smartphone can be reached via the Internet, thus making inter-computer communication and therefore illicit penetration by hackers a major problem.

But PCs and smartphones on the Internet do not fit the mould of mandatory access controls. They do not have the paraphernalia of mainframes, with operators and system administrators and so on. People own their own devices and expect to make their own decisions about how they can use them. This, together with the lingering death throes of the mainframe industry, makes it clear that the coloured books no longer provide a balanced view of computer security. Discretionary controls have undoubtedly become far more important than the role given to them in those documents.

Hence this book had been written primarily with discretionary access controls in mind. However, the purpose of the present chapter is to demonstrate that implementing both rule-based systems and mandatory access systems (even side by side) is also straightforward in SPEEDOS, beginning with a description of how the Bell-LaPadula rules, and by implication the Biba rules, and most rules based on the Access Rule Model (see chapter 3), can be implemented.

1 A Bell-LaPadula System

Chapter 3 describes the rules of the Bell-LaPadula security model. The aim of these rules is to permit information flow only to trustworthy objects, and thus to solve a special case of the confinement problem for systems, as viewed through the eyes of the military and similar organizations. Each subject in the system is given a *clearance* which reflects his hierarchical role in the organization and each object (e.g. file) in the system has a similarly hierarchical *classification* which determines how subjects may access it. A set of *projects* is also associated with each subject; these define his permitted sphere of activity in terms of access to the files, and files are likewise associated with specific projects.

1.1 The Bell-LaPadula Rules

The rules of the Bell-LaPadula model are as follows:

a) Reading of Objects (simple security property):

$\text{(clearance (subject)} \geq \text{classification (object))} \cap \text{(projects (subject)} \supseteq \text{projects (object))}$

b) Writing of Objects (*-property):

$\text{(clearance (subject)} \leq \text{classification (object))} \cap \text{(projects (subject)} \subseteq \text{projects (object))}$

c) Creation of subjects:

$\text{Subject}_s \text{ creates Subject}_t \Rightarrow \text{(projects (Subject}_t) \subseteq \text{projects (Subject}_s)) \cap \text{(clearance (Subject}_t) \leq \text{clearance (Subject}_s))$.

To implement the Bell-LaPadula rules two modules could be used in SPEEDOS.

1.2 The Subjects File

The first, a file module (see Figure 36.1), maintains a list of subjects together with their clearance and associated projects. It supports the following basic semantic operations:

- for creating new subjects (identified by their unique SPEEDOS identifier), with a clearance level and a list of projects;
- for changing the subject details (e.g. with a new clearance level and/or changed list of projects).

All routines assume that the users already exist in the SPEEDOS system and therefore have unique identifiers. The first routine (creating new subjects) executes the kernel instruction `current_thread_owner`, and uses the result to establish (from the module's list of users) whether the caller is authorised to create a new subject at the proposed clearance level and with the nominated projects, using the Bell-LaPadula subject creation rule. If so it enters the new subject into its list with the appropriate details.

The remaining operations carry out similar checks and if the caller is authorised to do so the changes are entered.

Further semantic routines can provide an authorised caller with information (the clearance and projects) about another user.

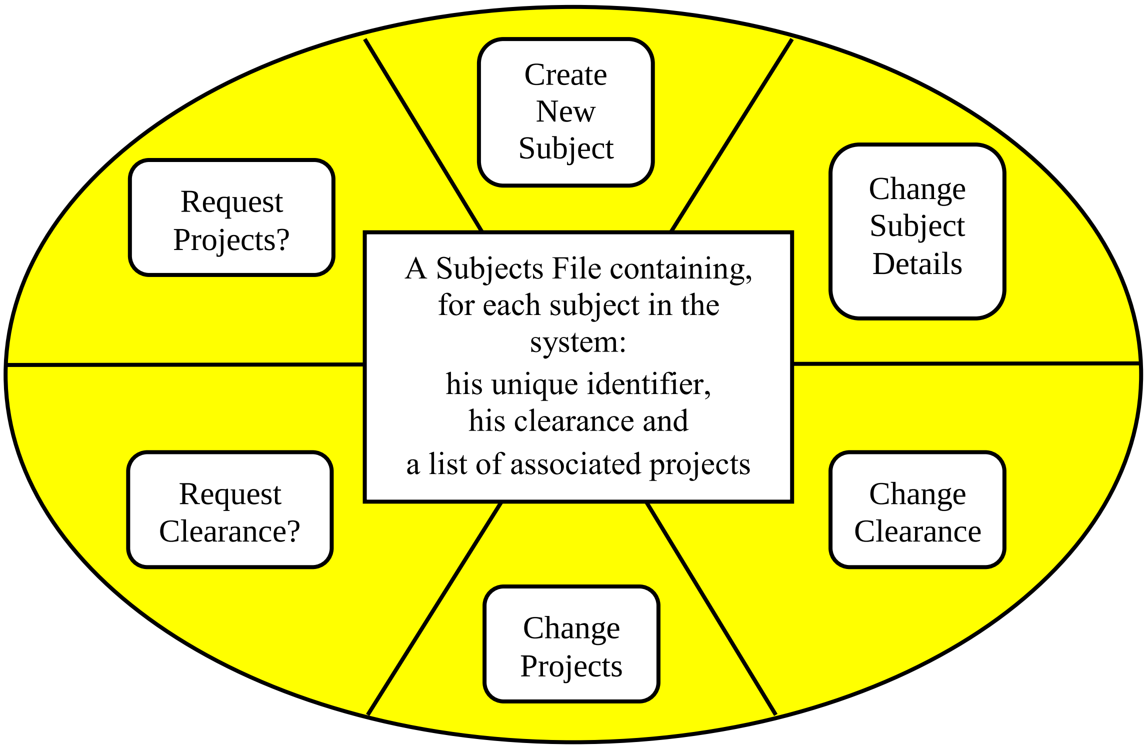
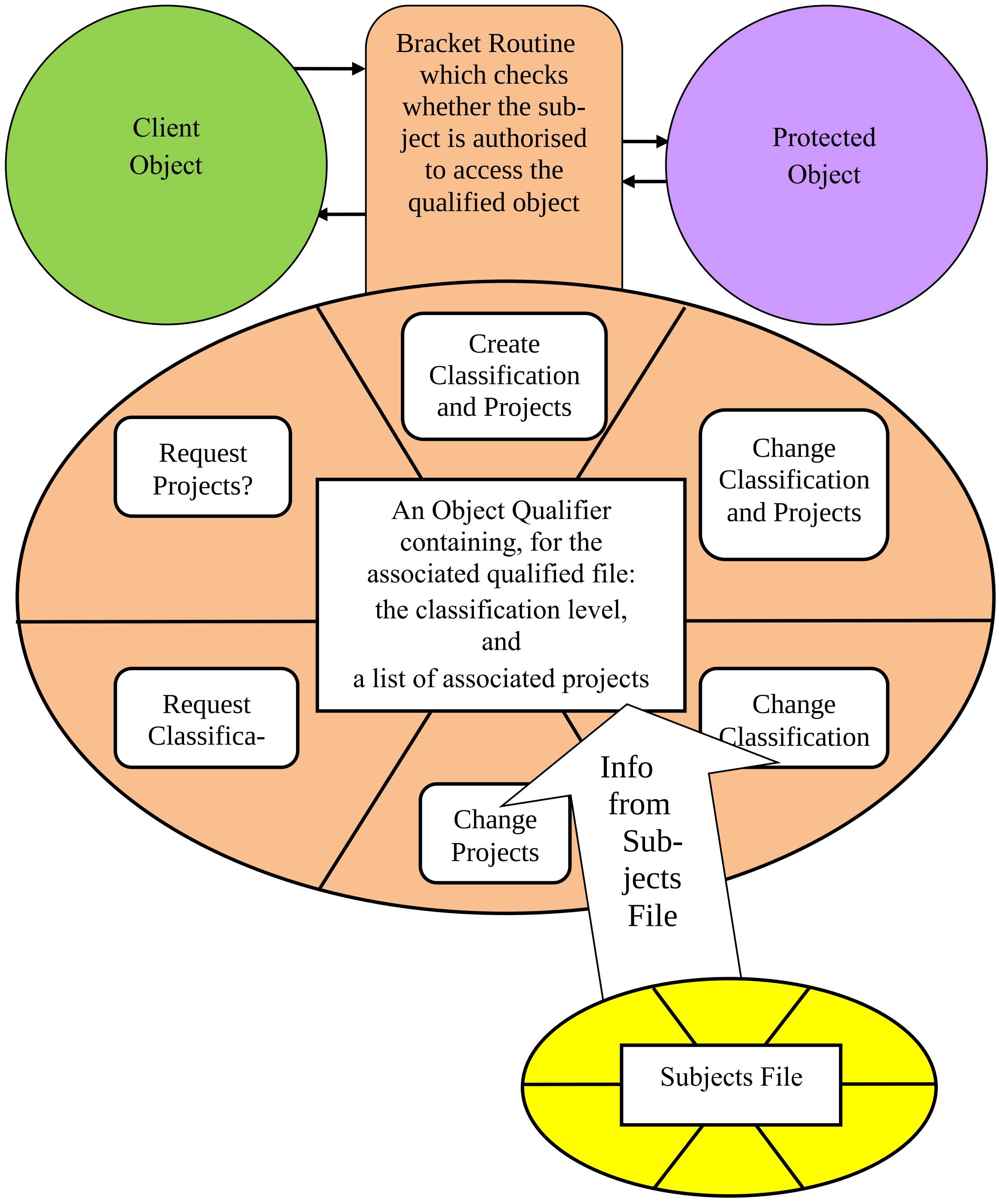


Figure 36.1: A Subjects File with Semantic Opera-

1.3 The Objects Qualifier Modules

The second module is a qualifying module which contains protection information (classification, associated projects) about a Bell-LaPadula object to be protected by its brackets routine (see Figure 36.2).



Client Object
Bracket Routine which checks whether the subject is authorised to access the qualified object
Protected Object
Create Classification and Projects
Request Projects?
Change Classification and Projects
An Object Qualifier containing, for the associated qualified file: the classification level, and a list of associated projects
Request Classifica-
Change Classification
Change Projects
Info from Subjects File
Subjects File

Figure 36.2: A Bell-LaPadula Rule Controller

It has interface routines that can be used by the system administrator to define and redefine a classification and the associated projects for the file, and to make enquiries about this information. One of the routines (callable by the system administrator) provides a capability for the subjects file. The system administrator also has a capability allowing it to set up and redefine the protection properties of each file to which the qualifier routines are to be attached, including the classification level and associated projects; he also indicates which semantic routines are subject to the Bell-LaPadula 'reading' rule and which are subject to the 'writing' rule.

Such a qualifier is associated with each protected file (i.e. each object) in the system and is set by the administrator to suit the individual security properties of the file. The qualifier has a call-in bracket routine which is activated for all calls to the protected file. When this is activated it first executes the kernel's `current_thread_owner` instruction to establish the identity of the caller. It then uses the capability for the subjects file to request details of the clearance level and projects of the caller. Next it determines the semantic routine number of the call by executing the kernel instruction `calling_ep` (see chapter 26 section 1.1) to establish whether it is a read or write routine. It then invokes an appropriate subroutine to carry out checks needed for the current routine number, using the read or the write rule as appropriate. If it discovers an error it raises an exception which disallows the call and then writes the error into a log.

1.4 Conclusion

We have now seen how the rules for creating new subjects and for controlling read and write access to protected files can be achieved in a SPEEDOS based Bell-LaPadula system.

It is interesting to note that more stringent rules could easily be applied. For example, since SPEEDOS can easily detect when a new file is being created (because the entry point number 0 signals that a constructor is being called) it could implement a further rule defining which subjects can *create* new objects. Similarly it could check that a user has opened the file before he calls the normal routines. It could also be supplemented by a call-out bracket which ensures that no information is released by malicious code.

It will be obvious to the reader that similar access control rules, e.g. those of the Biba model described in Chapter 3, can be implemented in a similar way. In fact SPEEDOS can use a similar pattern to implement most access rules that can be expressed using the Access Rule Model.

Finally, as we already observed in chapter 3, the Bell-LaPadula model does not guarantee the integrity of objects, because it permits subjects with a lower clearance to write to objects with a higher classification. Similar problems arise in Biba, which simply reverses the Bell-LaPadula model.

2 Retaining Control of a System

We now turn to a more realistic issue. Systems which involve multiple users normally need a controller (e.g. a system administrator or a superuser) who is responsible for the correct, secure and orderly management of the system. How can such a person exercise control and carry out his functions safely?

2.1 Retaining Control in a Business System

How can a business system administrator ensure that he can maintain control over the rights which he needs? The basic answer is straightforward. He achieves this by *not providing users with a capability containing the right to carry out sensitive actions* (e.g. by preventing them from calling the Container Manager's **`createContainer`** routine and other related routines, e.g. **`copyContainer`**, **`download`**) if they are not permitted to create files. This is easily organised, since when a system is first initialised by the system administrator, the basic capabilities needed to use the system are handed over by the system software to the thread which carries out the initialisation, in this case the system administrator's thread. It is then a question for the system administrator to determine which capabilities (with what access rights) he hands over to other users when he creates them (see chapter 31).

There are probably cases where other staff members may need to create files, for example in the business's software development department. The best way to do this is undoubtedly on separate computers which themselves are separate nodes, with carefully scheduled tests on the main computer carried out under expert supervision. But if that is not considered necessary, how can such staff be prevented from accessing the active business files? The answer is of course that they may be permitted to create new files using the Container Manager, but they do not receive capabilities for the business files which are currently in use. And just to be sure, the sensitive files can be bracketed to indicate which users of the system can – and/or cannot – access the files. In other words the brackets can implement both access control lists and revocation lists.

To ensure that problems do not occur when the system administrator is on leave or ill, etc., his deputy or deputies may need to take over the system administrator's duties. This could be arranged, for example, by the deputy being provided with the knowledge to allow him to log in as system administrator. Alternatively the administrator could nominate two or more deputies, who are each provided with half the information needed to log in as system administrator (just as in a bank two keys might be needed to open the vault door). But of course such arrangements should be determined by the business itself, and not fixed by SPEEDOS. For this purpose lots of mechanisms are available within SPEEDOS to provide the needed level of security. Inventive minds can use capabilities, access rights, kernel enquiries, bracket routines, passwords, the thread security register, etc. to devise good solutions.

2.2 Retaining Control in a Multi-User Discretionary System

Whereas in business systems the business staff may not be allowed to create new files, in a discretionary multi-user system such as a computer system at a university, the users will want to create files, etc. Nevertheless the superuser may wish to retain the power to delete student files (e.g. when students leave the university). He may also want to ensure that students do not exceed limits which he sets on the use of resources), etc. How can such requirements be organised?

Consider first the *delete* issue. Only one way of deleting a file has so far been described, i.e. by presenting a capability for the file with the owner right or the delete right set. To expect a student (or other user) voluntarily to provide the superuser with a copy capability in which the delete right is set is certainly not a guaranteed solution! One alternative is for the superuser to obtain the owner capability before it gets into the hands of the user. This solution could in fact be made to work, as is shown in Figure 36.3.

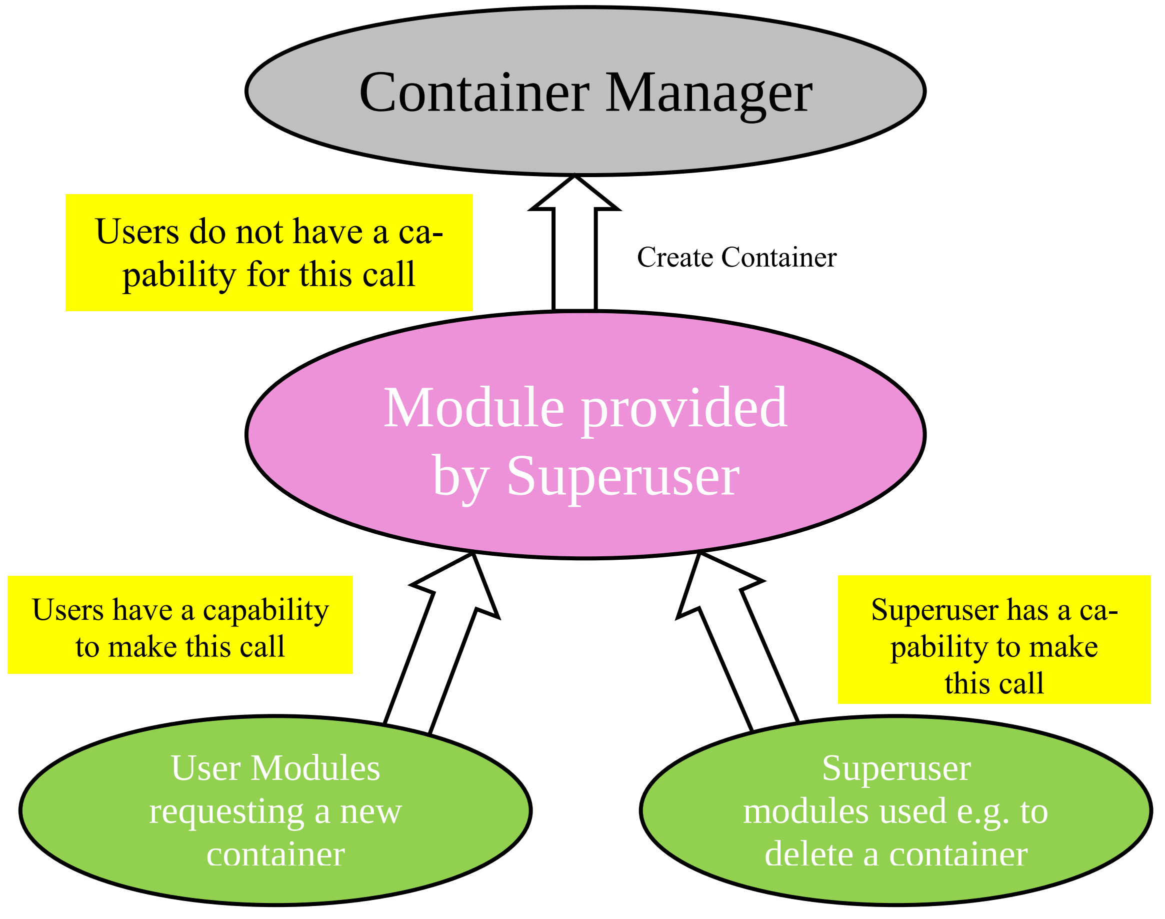

Container Manager
Users do not have a capability for this call
Create Container
Module provided by Superuser
Users have a capability to make this call
Superuser has a capability to make this call
User Modules requesting a new container
Superuser modules used e.g. to delete a container

Figure 36.3: Superuser Control over Capabilities

As in the business situation described in section 2.1, the superuser does not provide other users with a copy of the Container Manager's **`createContainer`** routine and other related routines (e.g. **`copyContainer`**, **`download`**). But in contrast with the business scenario, users in a discretionary system expect to be able to create their own files at will, so that can only be part of the solution. The superuser (as a person) does not expect to be directly involved in the creation of user files, but he can do so indirectly by providing users with a capability for a module that creates files for him. This module would retain the owner capabilities for all the containers which it creates, and thus the superuser has complete control over all the containers in the system. This is the solution which I proposed in an earlier (unpublished) version of this book, and I have presented it here to show the flexibility of the SPEEDOS system. Nevertheless I now reject the solution because

- it involves considerable overheads in managing the system;
- it gives far too much control to a superuser, who can easily misuse the mechanism to violate other users' privacy; and
- above all, this solution would obviously offer a perfect target for hackers!

Instead I now recommend a much simpler alternative. All that is required to solve the problem is a "superuser" capability which the superuser can retain for himself when the system is first initialised. In fact this is a special *kernel* capability which allows the superuser to delete (but not examine) all the containers owned by a specified user before he left (or was removed from) the system. An implementation of the mechanism could be as follows.

A kernel `delete_users` instruction takes as its parameter a list of users. These are specified as full 192-bit user identifiers. (Recall that a user identifier is the full container number of his first container, and that this is stored in the red tape of each container which he creates.) The kernel then scans all the containers at the node (or perhaps on a specified disc if the user can only create files on a single disc) searching for containers for which the users listed are marked as the owners (in the red tape at the base of the container, see chapter 19 section 2). The kernel deletes each such container.

This is a time consuming activity, especially if used for each student individually. For this reason it would be sensible to take the following steps to help optimise the deletion procedure.

- Include as many users as possible in the search. This is why a parameter has been included to list multiple users. In a student situation this makes sense, because at the end of an academic year many students leave the university at the same time.
- To reduce the scope of the search it would be sensible to place the files of all the students (or all the students in a single annual intake) on a single disc or disc partition.

How does the superuser know the identifiers of the relevant users? The answer is of course that he initially creates the users and in the first part of this procedure a new container, which is used as the new user's first container (and therefore his unique identifier), for which an owner capability is returned to the creating user (see chapter 31). This can then be held in a list of new users which the superuser can also use to store other details, such as his name.

2.3 Managing Forgotten Passwords

One service which a superuser or business administrator often provides is to help a user to log in if he has forgotten his password. This solution is not feasible in SPEEDOS, where logging in is carried out in an unconventional way and may not involve passwords (see chapter 22 section 11). The central control of passwords is also undesirable in a secure system, not only because it gives a superuser inordinate powers, but also because a central password system offers a tempting target for hackers.

But the problem is easily solved in SPEEDOS. Normally a user can create multiple processes/threads with different login procedures, so that if he forgets how to log in for one thread (thread A), he should still be able to log in to another thread (thread B). Provided that he has had the foresight to provide thread B with access to a capability for the authentication module of thread A, he could call this directly in thread B to call a semantic routine which temporarily turns off the tests and allows the user to log in to thread A. Alternatively he could use a semantic routine of the Thread Control Manager to change the login authentication module.

3 Resource Management and Exceeding Rations

In the general discussion of the kernel the issue of logging and controlling the use of resources was not discussed, because to do so would have taken the focus off the main issues. But also, it is not always clear what logging requirements and limits are necessary. It would therefore be appropriate, as far as possible, to leave such decisions to the user. One difficulty in achieving this is that the use of some resources is entirely under the control of the kernel. This is particularly true of the use of discs and other mass storage devices.

3.1 Disc Usage

Here at least two kinds of measurements can be made, e.g. on a user basis: (a) the amount of space used, and (b) the number of disc accesses made.

3.1.1 Counting the Pages Used

Measuring the amount of space involves counting the number of pages as they are allocated and reducing the count when a page is deleted, noting the identifier of the user on whose behalf this occurs and checking if a limit has been reached.

Counting the pages as such is relatively straightforward, but assigning their use to a particular user is not quite so simple. This cannot simply be achieved by establishing which user thread was last active, because the kernel may be servicing several requests.

The first page of a container is allocated as a result of executing the kernel's `new_container` instruction (see chapter 23 section 6.1). This accesses the appropriate Disc Directory Manager to obtain a new page. Additional pages for the container are requested by the Segment Manager (see chapter 23 section 5.1). In both cases a kernel virtual memory block is used. This contains the number of the thread issuing the request, and so a note of the user and of the count of new pages can be returned to the Container Manager or the Segment Manager; these can then access a log file to which the new pages can be added to the count for the user in question.

Deleting pages can follow the same pattern, more or less in reverse. Appropriate measures must also be taken with respect to the copying, downloading and uploading of containers.

3.1.2 Counting the Number of Disc Accesses

Since the discs are used to resolve virtual memory page faults such actions are not visible outside the kernel. This implies that counting must take place internally within the kernel. Yet it would be undesirable to clog up the kernel space with large amounts of data. For this reason an implementation is best sought by using shared co-module data, which also had the advantage that it is persistent (see chapter 17 section 3). Such a solution must also respect the fact that the kernel should not be delayed by too much processing activity.

The Container Manager can provide an appropriate data structure (preferably as its first persistent data structure, so that the kernel can find it easily via the state data pointer in the Container Manager's co-module table (see Figure 19.5). This can be a simple list of disc numbers and user thread numbers, each representing a disc access for the thread in question. The individual disc processes (see chapter 23 section 4.7) add such entries to the list as each access is made. They are aware of the user thread number from the virtual memory block which requests the access. The Container Manager has a thread which reads and clears the list at regular intervals. It counts the individual entries and writes them into a log file. To avoid the kernel's disc processes from searching for the list each time, they find this once when the system is initialised or a new disc added, and store its address in their local data.

3.2 CPU Time Usage

This is simply organised by the User Thread Scheduler, which uses the start time and stop time for each user thread which it schedules to calculate the total time which it uses. It provides a semantic routine which allows a Container Manager thread to obtain the details at regular intervals and to add these to a log file.

3.3 Printer Usage

Since the control of printers is organised outside the kernel, it is a straightforward procedure to count the number of files and/or pages printed by each user at a particular printer and to check that this is within the limits imposed by the system administrator or superuser. The appropriate module in which this occurs is the Printer Interface Module (see Figure 33.4).

3.4 Internet Usage

Statistics of Internet usage are best gathered by the Listener process (see chapter 34, section 8), since all Internet traffic passes through this kernel process. It should record details of all Internet requests and the unique identifiers of the user threads (from which the responsible user can also be identifier) and all responses to user thread requests, all external requests (which should not occur, except for requests from other SPEEDOS nodes) and record these in Container Manager data in a similar way to disc accesses. It should also record SPEEDOS kernel requests and replies to and from other (normally SPEEDOS) nodes, including the amount of bytes transferred.

3.5 Remote Inter-Module Call Resource Usage

The same statistics are of course gathered for resource usage at a remote node as a result of a thread being transferred following a remote inter-module call (see chapter 28). If the calling node indicates that it requires such statistics (not all will) it indicates this on the caller's stack when it makes the RIMC, and when the RIMC exits, the statistics are also provided on the stack with the return information. The kernel at the remote node obtains the necessary information by activating a surrogate stack, which obtains the information from its Container Manager.

3.6 Charging for Resource Usage

Whether charges are made for the use of resources is clearly a matter for individual systems. To obtain the details of what resources have actually been used by a user is made available to the system administrator or superuser by calling a semantic routine of the Container Manager (which has an overview as provided in the previous subsections).

3.7 Run-Time Monitoring

A system administrator should be provided with a run-time monitoring tool which provides him with an overview of the kinds of activities discussed in the previous section, e.g. a list of active threads, an overview of disc activity and of internet activity. In contrast with some conventional systems this should be transparent, showing exactly which users (by their unique identifiers) are active.

4 Initialising a New System

The processor of a new SPEEDOS node has an in-built read-only memory address which is directly accessible only to the kernel. This contains its world-wide unique node number (see chapter 16 section 2, which describes how this can be world-wide unique). Using the technique designed originally for the MONADS-PC system [24], the new node is securely booted and the kernel and related software are loaded from the system disc. The kernel then initialises itself and its co-modules, in so far as this is necessary. When this is completed, control is passed to an initial thread in a standard user level process of the user (in a single PC system), of the superuser or of the system administrator. This can then be used to create users and initialise files, etc. In the course of this initialisation the thread can invoke modules and pass parameters necessary to customise the system, initialise external discs, etc. It can use subthreads to allow longer tasks to be processed in parallel. When the initialisation is complete the system can be freed for use by other users.

5 Closing Down and Restarting a System

In many cases a user or superuser may wish to close down his system (e.g. overnight) and reactive it at a later time (e.g. next morning). One might think that because the system is persistent no special action is necessary, but this is not quite accurate.

Although a SPEEDOS system is persistent it uses the main memory as a cache for the persistent memory devices. Consequently the least which must be undertaken before the system can be turned off is to write to disc all the pages currently in main memory which have changed since they were paged in. For this purpose the kernel provides a privileged instruction `close_down`, which can only be executed by the User Thread Scheduler (because this can confirm that there are no executing user level threads).

When the decision is made to close down a system, the initial thread (i.e. the thread which was used to initialise the system, see section 4) is reactivated and calls all the Thread Control Managers in turn to close down their threads. (This means that the threads are brought to a state in which they can continue after the next system restart.) When a Thread Control Manager has done this it returns to the initial thread, which then continues to the next to do likewise. When this phase is completed, the initial thread calls the User Thread Scheduler to check that there are no executing threads and uses the kernel's `close_down` instruction. The stack and the thread state for this thread are frozen by the kernel which then closes the kernel threads down and ensures that they have all run to completion. It then stops the CPU and dies. It does not store the kernel state as such (see chapter 17 section 2) but it ensures that the shared co-module state data is consistent.

In order to restart the system the kernel is re-bootstrapped and then sets itself up to a functional state and reactivates the initial thread to restart the system at the user level.

As a final note, we observe that by restarting the system in this way, the problems which might otherwise arise from the fact that the user can reconfigure the system while it is closed down (e.g. by removing external discs) are avoided. If a re-started thread attempts to use a disc or device which has been removed from the system during close-down it causes an error for the thread, but not for the system and is treated like any other error, e.g. in that the system requests that the disc is brought on-line or causes an exception condition for the thread.

6 Handling a System Crash

Systems can crash for a number of reasons, e.g. because of a loss of power, because of an unrecoverable processor or disc failure, because of an error in a key kernel process or in the User Thread Scheduler.

In the early days of computing, power failures were a major problem, but that should no longer be the case. In fact even most PCs have batteries which allow the PC to run for several minutes (or even hours) without external power, and warn users when the battery is beginning to get low, thus allowing users to either provide external power or to organise their system to run down in an orderly manner. Larger systems can be organised to have an uninterruptible power supply (UPS)[253] which will at least serve the same purpose or for a longer use of the system some form of emergency power generator.

In the very unusual situation that the processor or main memory has an unrecoverable error, the manual intervention will of course be necessary.

Whatever the cause of a system crash (assuming that the processor and the main memory are in order) both the reason for the crash and the extent of the damage must be established. On the assumption that power is available, at least for a reasonable amount of time, a special kernel process should be automatically activated which first saves the current values in the registers, then establishes which kernel process is active (or was the last to run), and if a user thread was currently active. From this information it should then be possible to establish the nature and extent of the damage.

[253] see https://en.wikipedia.org/wiki/Uninterruptible_power_supply.

The most likely cause of a system crash is a software failure. If an application contains a software error, this can normally be handled as described in chapter 22 section 9.2, and the SPEEDOS protection mechanisms will ensure that the effects of such a failure are limited.

Of course SPEEDOS systems should be regularly backed up to minimise the effect of failures. As in other systems there are several ways of doing this, e.g. at the end of every day take a full back-up, or use some form of on-line backup system. To allow a special backup thread to copy an entire disc it can be armed with a capability (the owner capability for the disc or a disc capability in which an 'archive' right is set) which allows the thread presenting the capability to have page access to the disc. Notice here that an archive of a disc must be marked as such, since it contains owner capabilities and possibly other items which should be unique in a running system. To convert an archive into a "live" disc a procedure must be devised which solves this issue.

Finally we also mention that there are ways of using duplicated hardware to keep a critical system running. There is no reason why all such techniques should not be used with SPEEDOS, but they are beyond the scope of this book.

Chapter 37
An Example – Online Banking

Banks, but not only banks, have a long history of adapting their computer systems to changing circumstances, improved technology and new ideas. Consequently it is not irrelevant to begin by asking how easy it is for them to carry out this process of modifying their software. This is relevant in the context of security because computer programming is not an easy discipline to master, so that each time changes are made it is easy to make (and introduce new) mistakes, and mistakes in programs are one of the main causes which can give rise to security and privacy problems. The hackers' evil handiwork is of course made easier if they can discover mistakes in programs which they can use to their own advantage.

Added to this is the fact that the CEOs of companies are themselves often under pressure from their investors to introduce improvements as quickly as possible and to complete the necessary changes to their software (and of course other) products before their competition does. This pressure is passed down to those who are responsible for the new products and those who are involved in making the changes necessary to bring the new product to market as soon as possible. But pressure only leads to hastily carried out and therefore often imperfect work.

1 Software Structures

As was explained in volume 1 chapter 13, one of the reasons why software modifications are often badly carried out is because the tools available to programmers are quite inadequate. There the fundamental structural problem of conventional systems was discussed in a general way, drawing attention to the many problems that are created by what I called "flow of control modules". Using this standard technique, which is forced on programmers by standard programming languages and operating systems, the fundamental data structures in a system are separated from the modules which contain the program algorithms that access these. I strongly recommend that readers look again at chapter 13 and remind themselves of the difficulties which this "normal" technique brings (and bear in mind that this is the technique which still persists in conventional operating systems and programming language). Here is a summary of the shortcomings which were listed and explained there[254]:

(i) the specification of the system design is difficult;

(ii) communication between the implementers of separate modules is high;

(iii) inconsistent modules create difficult debugging problems;

(iv) verification is difficult;

(v) synchronisation problems easily arise;

(vi) maintenance of the system is difficult;

(vii) extension/adaptation of the system is difficult;

(viii) optimisation of the system is difficult.

It was suggested that the solution of these problems lies in rigorously enforcing the information principle and object orientation techniques at the operating system and programming language interface level, and qualifying types were added as a new technique.

These ideas have formed the basis of the SPEEDOS design, in two senses. First, the principles have been used as the basis for the SPEEDOS design itself.[255] Second, a primary aim of the SPEEDOS system is to free user applications from the straightjacket of conventional systems, allowing applications also to be designed according to the information hiding principle. Consequently future systems and applications which are based on SPEEDOS should benefit both from the support of a more reliable and secure operating system, but also from being able to apply the same principles directly to their own software systems.

With this in mind we now focus attention on developing in outline some of the main components of a banking system, showing first how a fictional banking system first designed in say the 1960s might have benefitted from the information hiding approach as it would have gone through various stages of development up to modern times. But before we do this we should briefly look at how a fictional system might have fared using conventional techniques.

[254] See volume 1 chapter 13 section 5.

[255] The kernel design itself does not directly use them, but this is because the job of the kernel is to implement them for the rest of the operating system. However, the kernel related co-modules do.

2 The Framework of a Conventional Design

At the heart of any banking system is the idea of bank accounts. Using conventional systems these are stored in files in the file system, and are accessed by programs which have embedded in them subroutines for carrying out the typical operations on the bank accounts, such as opening an account, closing an account, making deposits, making withdrawals, adding interest, setting an overdraft limit, enquiring about the current balance, listing recent transactions, etc.

Not all of these operations should be made available to all the employees in a bank, but should be restricted to those with a "need to know" and a right to carry them out. For example the bank teller should not have the authority to add interest to an account or to increase an overdraft limit. A bank branch manager will possibly also not have the authority to add interest, but a bank accountant might have, etc. In a conventional system the easiest way to give different authorisations to different staff is to provide these in different programs, such that a particular staff member (or employee group) can have a separate program in which the appropriate access is provided. This is reflected in Figure 37.1, which is based on Figure 14.2).

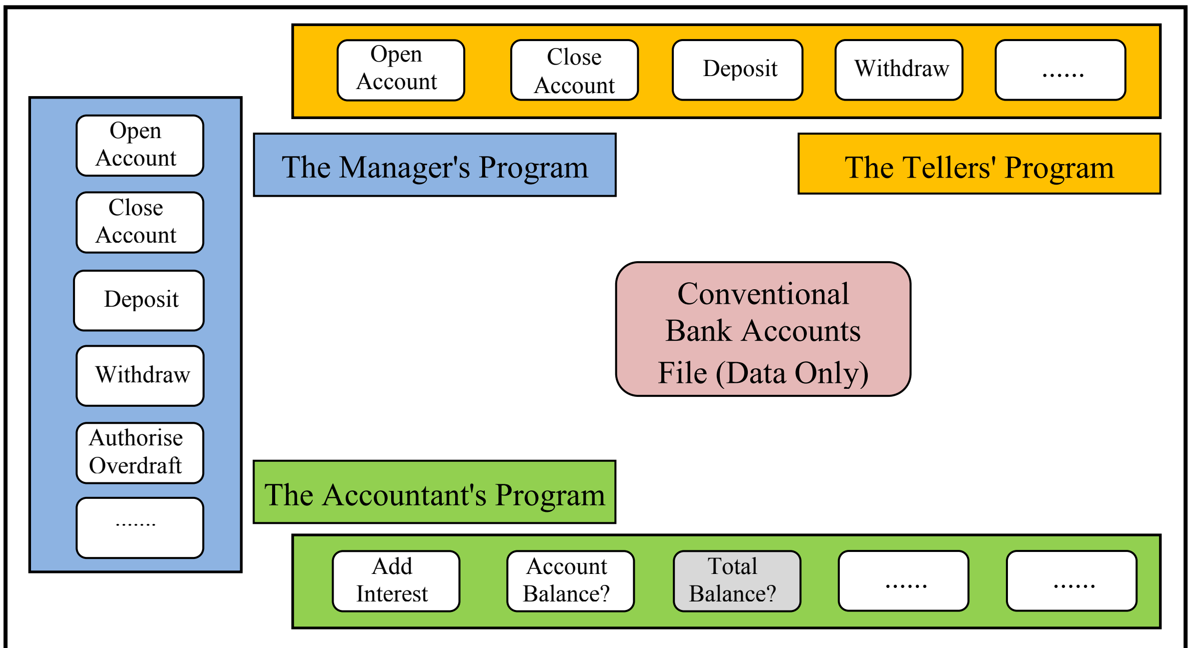


Figure 37.1: Accessing a Conventional Bank Accounts File

The reason for this is that protection works at the file system level, which associates rights with entire data and program files. The bank employees can be given access to programs and the programs can be given access to files.

One particularly significant drawback of this arrangement is that in effect the same operation may appear in different programs, and normally programs in a large operation are programmed by different programmers. Consequently not only might the same logical operation on the bank account file be programmed in several individual programs by different programmers, but the actual programs may be different. And such duplicated effort is not only costly (on programmer salaries) but can lead to problems if a change is needed. And above all, the problems listed in section 1 also possibly arise. One way of tackling such issues is to use database systems, but this adds another level of complexity and the possibility of yet more errors.

A further problem can arise with respect to access rights. If all the bank accounts are held in a single central file, the implication is that a bank teller at one branch can access to all the accounts of all customers, even possibly those at different branches.

3 The Effects of Technological Changes on the Conventional Approach

With the passage of time radical technological changes and improvements have taken place which have affected banking (and often other) systems. We now briefly look at the main developments and their effects on the bank programs.

3.1 Batch Processing Systems

Commercial computing systems in the 1960s were based on an arrangement called "batch processing". This was the age when computers were large monsters which stood in large computer rooms (and sometimes even in separate computer centres), when rotating discs still were physically large but with very small storage capacities. The main medium for storing files was magnetic tapes, which had the severe disadvantage that they could only be accessed sequentially. Furthermore data input to the system was achieved by punched card or punched paper tape devices. Figure 37.2 illustrates how such systems functioned.

In such a system the information about bank accounts was typically held on a magnetic tape, called the "master" file, in a fixed sequence (ordered for example by increasing bank account number). The day's banking transactions were collected together each evening, they were encoded onto punched cards or paper tape and then were read into the system. There the transactions were checked for consistency, reasonableness and so on by an "edit" program, and after that they were copied onto a magnetic tape and sorted into the same order as the master file. In the next step the master file update program read the transaction file and the main file together, and created from them a new master file on a different magnetic tape. This program included the code for processing the individual transactions and modifying the banking data, recording deposits, withdrawals and transfers, authorizing overdrafts, etc. In the final stage relevant information was printed about the day's transactions. On the next evening the transactions for that day were vetted, sorted and read against the master file, and yet another new master file was created.

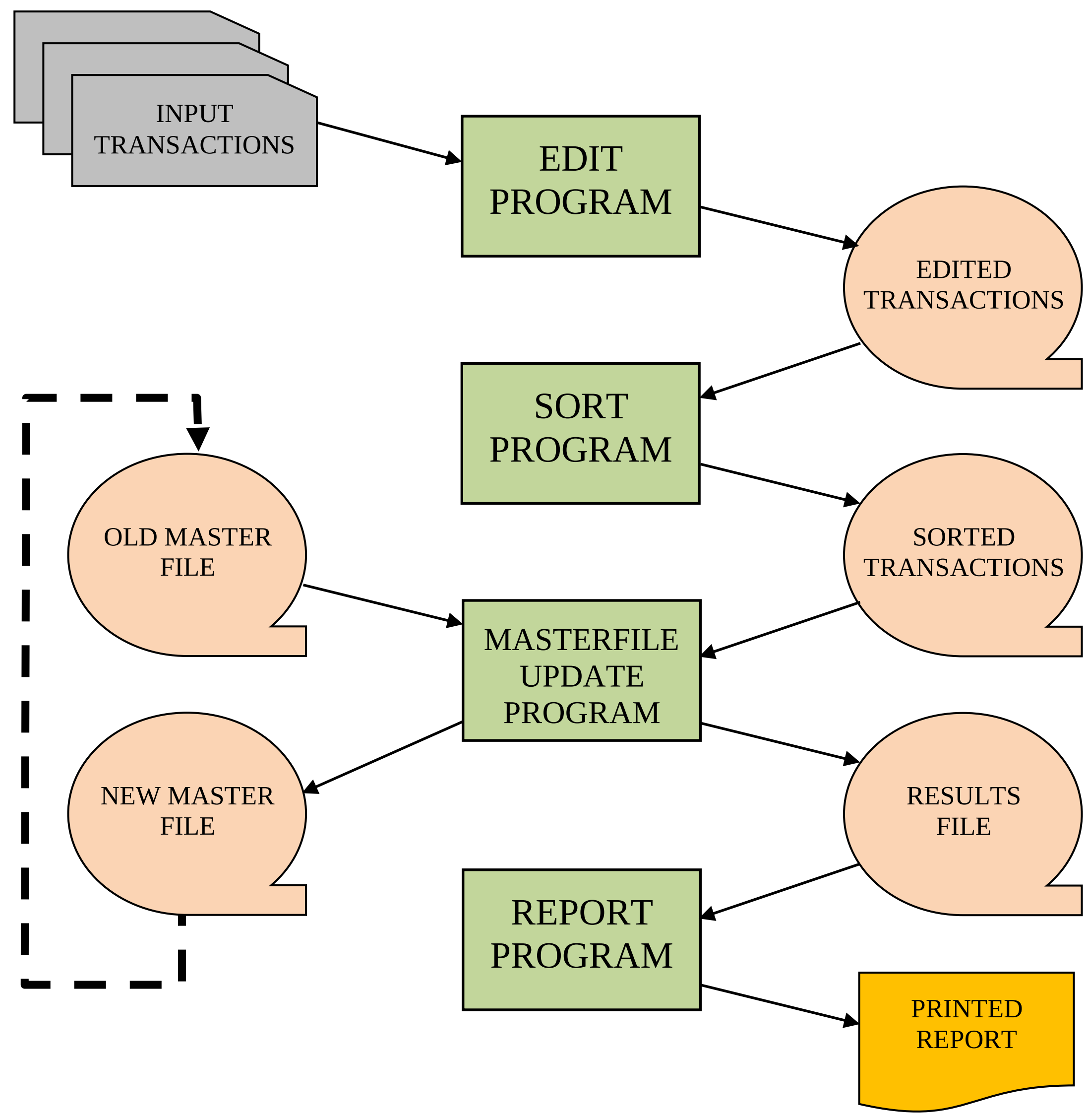


Figure 37.2: Bank Programs in a Batch Processing System

The control code in the master file update program consisted basically of a large loop in which the next transaction was read and the appropriate subroutine for the deposit, the withdrawal, etc. was invoked. It was in this program that the semantic routines were buried. Since they did not appear on the interfaces of the programs *they did not need to be specified in the design documentation*.

The whizzing tapes which you may sometimes have seen in computer rooms in old films are reminders of that era. These were eventually replaced by files on disc, but although disc accesses need not be sequential they were often used as if they were sequential tapes to minimize the changes to the system.

3.2 Online Terminals for Bank Staff

The next stage in the development of banking systems was the introduction of online terminals for the bank staff. For those banks adventurous enough to introduce online updating of the master files on disc, a transaction processing monitor program was needed. Bank staff had to input transactions into a transaction processing system which read the transactions from terminals, processed them and updated the master file – which was by this time a disc file in which the relevant accounts could be accessed directly. Different control routines were needed in the transaction processing monitor, but although the basic semantic file operations (deposit, withdraw, etc.) had not changed, new routines to implement them were needed in the transaction processing monitor.

3.3 Automatic Teller Machines

Later ATMs (automatic teller machines) were introduced, from which customers can directly initiate transactions. New programs were needed with new control routines to read in the customers' plastic cards, to check PINs (personal identification numbers) etc. And again the basic banking operations, although these had not changed, had to be incorporated into new programs, which typically meant that they also had to be rewritten.

3.4 Online Customer Banking

Later still online customer banking from home computers was introduced, once again requiring new programs to access the banking files. This time other protection requirements had to be built in, but although the basic banking operations did not change, these once again had to be incorporated into the new programs.

3.5 Online Banking from Smartphones

And of course further adaptations of the programs were needed to cope with the introduction of banking via smartphones.

3.6 The Fundamental Problem

Conventional systems (not only banking systems) suffer from the fact that they separate data structures (e.g. files) from programs. One of the results of this separation is that programmers, partly for protection reasons but also for structural reasons, constantly need to rewrite their programs as technology develops. The latter problem is exacerbated because programs need to serve two purposes in conventional systems. First they serve as "control" routines and second as "data management" routines, such that with technological advancements the control routines need to be updated or changed, whereas the data management routines can usually remain more constant, but since the conventional concept of a program forces programmers to mix these two things up in a single program structure new programs had to be developed.

You might of course want to argue with this analysis by pointing out that data management is separated in conventional systems by the separate existence of file systems. But the fact is that conventional file management systems cannot take into account the semantics of the file data; they only provide a few basic routines for organising files which have different semantics into patterns which simplify their organisation in the file system (e.g. as index sequential files, as sequential files, as hashed files, etc.). And as we discussed in chapter 2, the best that they can do is to provide a totally inadequate protection system, based not on the semantics of the data but simply on a choice between no access, read access and write access. And database systems, while offering an improvement over the simpler file systems, introduce more complicated mechanisms which also, in the final analysis, do not solve the protection problems adequately and are often cumbersome because they are often built on top of file systems.

4 Using the SPEEDOS Approach

As was explained in volume 1 chapter 13, SPEEDOS offers its users the possibility of organising their applications in such a way that the semantic routines associated with major structures (such as bank accounts), are tightly bound to the data itself in modules, while the control programs are held as separate modules. The most important effects of this different structural method are that:

a) when the control structures have to change in order to adapt to improved technology or new ideas there is no necessity to change the file modules containing the major data bases in the system (such as, in the present case, the bank accounts).

b) the semantic routines associated with the data (e.g. the deposit routine, the withdrawal routine) do not have to be repeated in multiple programs, which is both cost saving and reduces the potential for errors. This is not merely a software engineering advantage but also a security advantage.

c) protected access to a data structure such as a bank account can be based on the semantic routines themselves.

In other words, we have a win-win situation by providing a better software engineering mechanism and a much more flexible security system. We now look in a little more detail how some parts of a fictitious banking system might look.

4.1 A Bank Account File

We re-use Figure 2.7 as Figure 37.3 to remind readers how a bank account file and its semantic routines might look. Note that in a real bank account there would certainly be more routines than are listed here. These are only provided for illustrative purposes. Furthermore in the next few sections we make the obviously false assumption that a bank only provides one kind of account for all its customers. This assumption will be corrected in sections 4.6 and following.

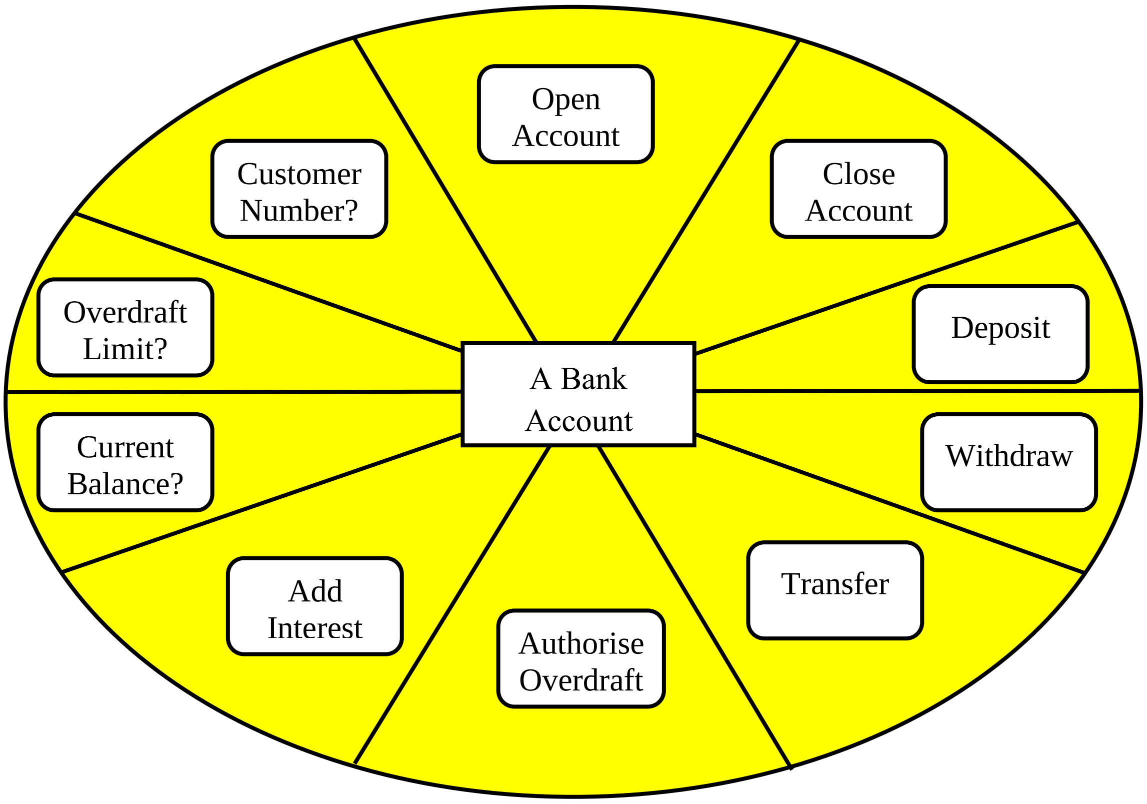


Figure 37.3: A Bank Account with Semantic Operations

These routines can be divided into two groups: *operations*, which manipulate and change the state of the bank account, and *enquiries*, which provide the subject with information about the account without changing its state[256]. Examples of operations include the routines 'deposit', 'withdraw', 'transfer', 'add interest' and 'authorize overdraft'. These can be viewed as differing kinds of write operations. The enquiries are distinguished in the diagram by a question mark. They include such routines as 'customer number?', 'overdraft limit?' and 'current balance?'. The enquiries can be viewed as different kinds of *read only* routines, which return specific information to the caller. Such read only routines are protected by SPEEDOS to ensure that they do not make modifications.

4.2 Protecting Access to the Semantic Routines

Because they are protected in SPEEDOS by module capabilities, users of the semantic routines can be provided with separate capabilities on a 'need to know' and a 'need to use' basis. Figure 37.4 (repeated from Figure 2.8) illustrates how the authorisations might look.

[256] The programming language Timor [7] allows programmers to distinguish between these two kinds of semantic routines, thus also allowing SPEEDOS to ensure, for example, that enquiries cannot modify the data, and thus add further protection to a system.

	Teller	Branch Manager	H.O. Accountant	H.O. Auditor
Open Account	√	√	x	x
Close Account	√	√	x	x
Deposit	√	√	x	x
Withdraw	√	√	x	x
Transfer	√	√	√	x
Add Interest	x	x	√	x
Authorise Overdraft	x	√	x	x
Customer Number	√	√	x	√
Overdraft Limit	√	√	√	√
Current Balance	√	√	√	√

Figure 37.4: Authorisations based on Semantic Routines

Notice that this diagram is of a different kind from those which were used to illustrate Lampson's Access Matrix in chapter 2. An entire column in Figure 37.4 corresponds to a single access rights field of Lampson's Matrix. What this means is that Figure 37.4 refers to the access rights for a single object (identified by the capability). In other words, it is not sufficient simply to define which operations of an object *class* a particular subject may invoke. Such a list of permitted operations only makes sense in conjunction with a particular object or list of objects. For example, I may have the right to withdraw money from my own bank account, but that should not automatically give me the right to withdraw money from yours!

4.3 How Many Bank Account Files?

In principle all the accounts of a particular type (e.g. savings accounts) could be kept in a single file module, but this would have the disadvantage that access rights associated with the routines can only be associated with all the accounts in the module as a single group[257]. Keeping each account in a separate file module has the advantage that different accounts at the same branch can, for example, be associated with different customer advisers. Separate accounts can of course share the same code module, because the protection is based on the access rights to the data files, not to code files (as is usually the case in current systems).

If each bank account has a separate file it has a separate owner capability;

[257] The implication of this approach would be that a further layer of protection, such as a password, would be needed.

additional capabilities can be created by copying them (subject to the metarights in the capability from which the copy is made, see chapter 26 section 3.3).

4.4 Collections of Bank Accounts

For some purposes the bank will need routines which are associated with a group of accounts, for example to maintain a cumulative balance of all the accounts of a particular type at a branch or to add interest to the accounts, etc. Such routines can be programmed in separate file modules, which again can share a single code module. Such a file module will have access to a stored capability (with appropriate access rights) for each account, giving it access to the individual details of the accounts files. The rights to the individual accounts in these collection modules will be restricted on a need to know basis. For example in a module designed to calculate a cumulative balance for all the accounts the module might only have the right to call the 'current balance' routine for each account.

4.5 Using the More Traditional Approach

Nothing in SPEEDOS (or Timor) forbids the more traditional approach of placing an entire set of bank accounts into a single file module and providing some additional routines which operate on all the accounts in the file to maintain cumulative balances, add annual interest to them, etc. This is possibly marginally more efficient and easier to organise, but it lacks the extra security provided by keeping each account in a different file.

4.6 Different Kinds of Bank Accounts

So far we have made the simplifying assumption that our fictitious bank only offers one kind of bank account. In reality banks have a whole range of account types, some but not all of which are interest bearing and some offer higher interest if the money in the account exceeds a certain amount and is left on deposit for a fixed period of time. Other accounts are loan accounts, or accounts associated with stocks and shares, etc. For such accounts different module types with different routines are needed[258]. But such accounts must be distinguished from each other, so that in practice each account associated with a particular customer will have not only a customer number but also a separate account number.

4.7 Customer Information

Another kind of module is needed, describing the customer and his accounts. Such a module might record details of the customer, e.g. name and address, date

[258] Timor offers facilities for code re-use which are independent of subtyping [8]. This makes it relatively straightforward to re-use some parts of the code for different kinds of account, without affecting the semantics of the objects.

of birth, marital status, nationalities and passport numbers, bank card numbers, tax identification numbers, the customer's unique SPEEDOS identification number and other relevant information (e.g. taxation information on his annual interest, details of authorised representatives). This customer information module will also hold a list of the customer's account numbers and capabilities for them. It will be set up when the customer registers as a customer and will be modified as his requirements change (e.g. when he opens a new account).

5 Online Banking

In conventional Internet banking systems it is standard that data is processed by the banking system on its own computers and data is returned to users and displayed on the user's screen, possibly with the help of HTML or similar. As past experience has shown, this technique, as used in conventional computers, has often led to security breaches and theft of funds.

5.1 A SPEEDOS Online Banking Architecture

The most obvious and certainly the best way to organise an online banking system using SPEEDOS is to take advantage of its in-process philosophy and the availability of remote inter-module calls with call-back calls (see chapters 28 and 35). In this case the interactions between the customer and the bank begin by the user at his computer activating one of his threads which invokes a call-back module at his own node.

5.2 The Call-Back Module

The call-back module will have previously been supplied by the bank and is a module which must be installed on the bank customer's computer[259]. In view of the very high level of security needed by the bank, this module must not contain information which could be problematic for the bank in the case of failure at the customer node. Hence all security relevant information must be stored on the bank's own computer(s), including the customer's registration details (see section 4.7). However, the call-back module can contain some initial security checks, which can be supplemented by further checks in the banking module on the bank's own computer.

5.3 The Relationship between Bank Modules and Call-Back Modules

In many conventional systems HTML serves as a medium between a remote

[259] The call-back module could previously have been installed on the user's computer via mechanism described in chapter 35 section 2.2, or it might for example have been made available on an installation CD (see chapter 27 section 2.5), etc. For smartphones it can be supplied to the customer as an "app", i.e. the bank's call-back module can be regarded as a smartphone application module if the smartphone has a SPEEDOS architecture.

computer and its displays on the user's local computer, HTML interpreters (in particular browsers) have often proved to open up major security loopholes. For this reason the SPEEDOS design has attempted, as far as possible, to avoid the use of browsers (see chapter 34 section 2). This applies especially to websites which are used in systems that need a very high level of security, such as banking systems. Hence we recommend that HTML is not used for online banking. Instead we encourage the use of the SPEEDOS call-back mechanism. In the case of highly secure systems we recommend that call backs are used in a particular way, applying a rule that the call-back modules do not persistently store secure information (such as registration information) on the user's computer. Instead, when such information is requested from the user it is transferred from the call-back module to the website module (in this case a module on the bank's computer) for checking and/or for storing persistently, and when sensitive information has to be supplied back to the user this is held persistently by a website module and only transferred (temporarily) to the call-back module on demand. In this way secure information is placed only in temporary segments except at the website, and is therefore less vulnerable to thieves and hackers.

5.4 Starting Online Banking

The call-back module is activated (on the customer's computer) via a local inter-module call to it in one of the customer's persistent threads.

When activated the call-back module displays the bank's website start page, which allows the user to select from a number of options, including a page showing information about the bank or a page which allows the user to log in to his accounts (see Figure 37.5).

In the following subsections we assume that the user selects the latter. This then uses a capability which the bank has stored in the call-back module to make a remote inter-module call to an appropriate login entry point of the customer information module described in section 4.7. In a perfect world no further checks would need to be made, since in theory the capability used to make the call is evidence of the right to call the customer information module. However, this capability might have been secretly obtained by a hacker and so in practice further checks are appropriate.

5.5 Implicit Checks

Some of these checks can be carried out implicitly, i.e. without the user being directly involved. The kernel's environmental checking instructions (see chapter 26 section 1) are especially useful for this purpose. For example, as an initial check the call-back module can use the kernel instruction `target_code_owner` in a call-out bracket routine to ensure that the capability used to make the remote inter-module call is actually the bank's software. If this check fails the call to the bank is abandoned.

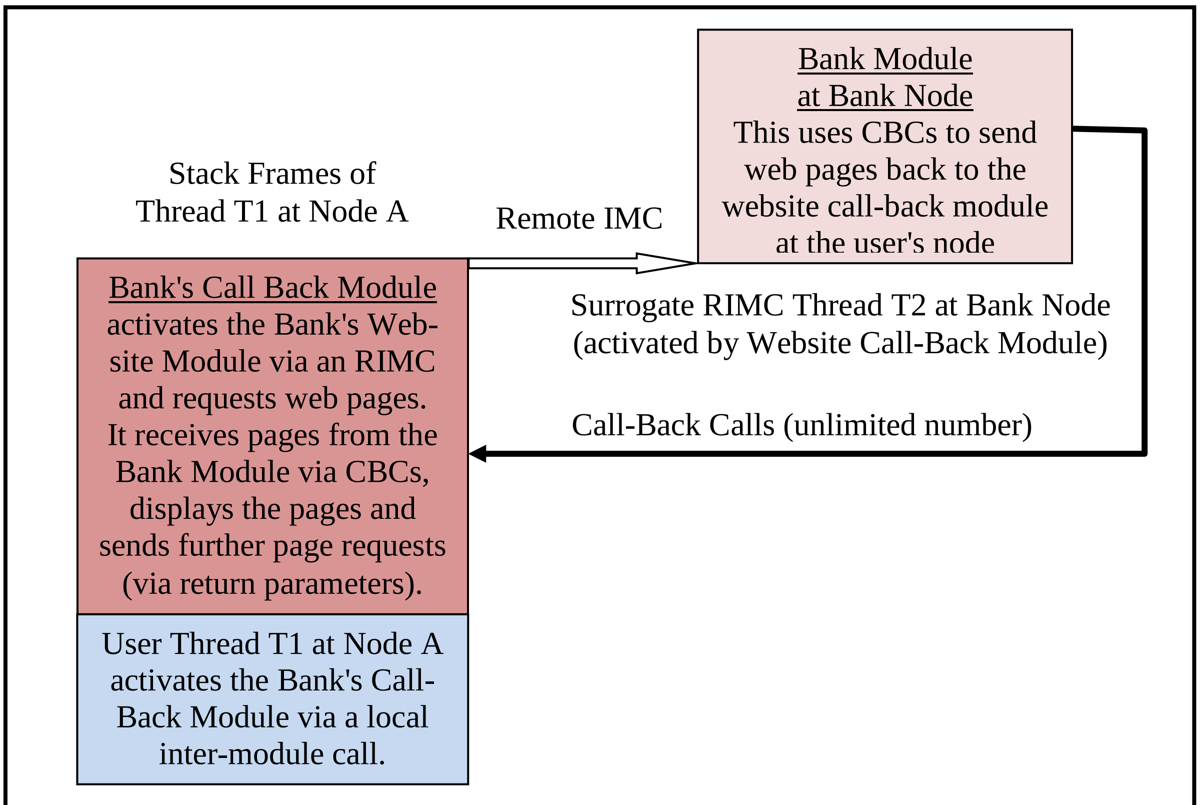

Bank Module
at Bank Node
This uses CBCs to send web pages back to the website call-back module at the user's node
Stack Frames of Thread T1 at Node A
Remote IMC
Bank's Call Back Module
activates the Bank's Web-site Module via an RIMC and requests web pages. It receives pages from the Bank Module via CBCs, displays the pages and sends further page requests (via return parameters).
Surrogate RIMC Thread T2 at Bank Node (activated by Website Call-Back Module)
Call-Back Calls (unlimited number)
User Thread T1 at Node A activates the Bank's Call-Back Module via a local inter-module call.

Figure 37.5: The Basic Architecture of SPEEDOS Online Banking

Similarly the bank module which is called by the call-back module can check (against the customer information stored at the bank) that the owner of the calling module is its own call-back module, using the kernel's `calling_code` instruction. It can also check that the `calling_file_owner` and the `current_thread_owner` are authorised to access the accounts (either as account owner or as authorised representative).

5.6 Explicit Checks: Identifying the Customer

In order to be sure that the online banking customer is really who he claims to be (e.g. rather than another person who has illicitly gained access to his computer), it is appropriate to adopt a logging in strategy in two steps. The initial remote IMC to the bank will be made before these checks are carried out and call-back calls to the bank's call-back module (on the customer computer) can be used to obtain the information from the user (after first using the implicit checks to ensure that the call-back module is genuine).

5.6.1 The Bank's Identification Procedure

In this first stage the bank carries out checks of its own devising, which have been agreed with the customer. These might, for example, include a password or follow a PhotoTAN procedure[260], etc. The aim is to convince the bank that it is dealing with a genuine customer entitled to access his own accounts[261].

5.6.2 The Customer Checks

If the first stage is successfully completed, the bank module then activates the second stage, the purpose of which is to convince the genuine customer that no-one is trying to impersonate him.

This stage uses an authentication module similar to that provided for users as they log their persistent threads into a SPEEDOS system (see vol. 1 chapter 15 section 3.4 and chapter 22 section 11.4). In this case the user's identification module should not be stored in the call-back module but rather on the bank's own computer. Such checks are carried out as defined in the user's authentication module, which challenges him to provide evidence that he is the valid user, as was previously defined by the user, not by the bank. A capability for this module will be stored at the bank as part of the customer information (section 4.7) and the bank will also store the module itself. This may, but need not, be a copy of the authentication module used to log the user into his current thread[262].

The bank computer activates this module (on the bank's computer) as its next step after carrying out its own checks. The bank's code can locate the authentication module by using the kernel's environmental instruction `current_thread_owner`. This allows it to locate the user's customer information and therefore obtain a capability for the authentication module. In order to carry out the identity checks it uses the call-back module to display the questions and to return the answers[263] (see Figure 37.6). The authentication module may make provision for an inter-module call to an "alarm" module in the bank, thus allowing a fraudulent user to think that he is logged in successfully, while the bank takes special precautions to prevent the fraudster from doing damage and perhaps attempts to locate/identify him.

If the final authentication step succeeds the call-back module then works in step with the bank module to display and carry out the user's transactions. When the customer has completed these he signals this via the call-back module, which then advises the bank module to log the user out of its system. This could include a provision to allow the user to indicate that the bank should use a dif-

[260] see for example https://www.youtube.com/watch?v=Ivuodu8plV0.

[261] Nothing is absolutely safe, since smartphones, PhotoTAN devices, etc., just like credit cards, can be stolen. For this reason it is best not to rely on the bank's tests alone.

[262] It may have to be provided in a standard source format and re-compiled at the bank if the computer instruction set used by the customer's computer differs from that used by the bank.

[263] Note that the authentication module can directly use the call-back mechanism, see chapter 28 section 7.2.2.

ferent authentication module when he next logs in online.

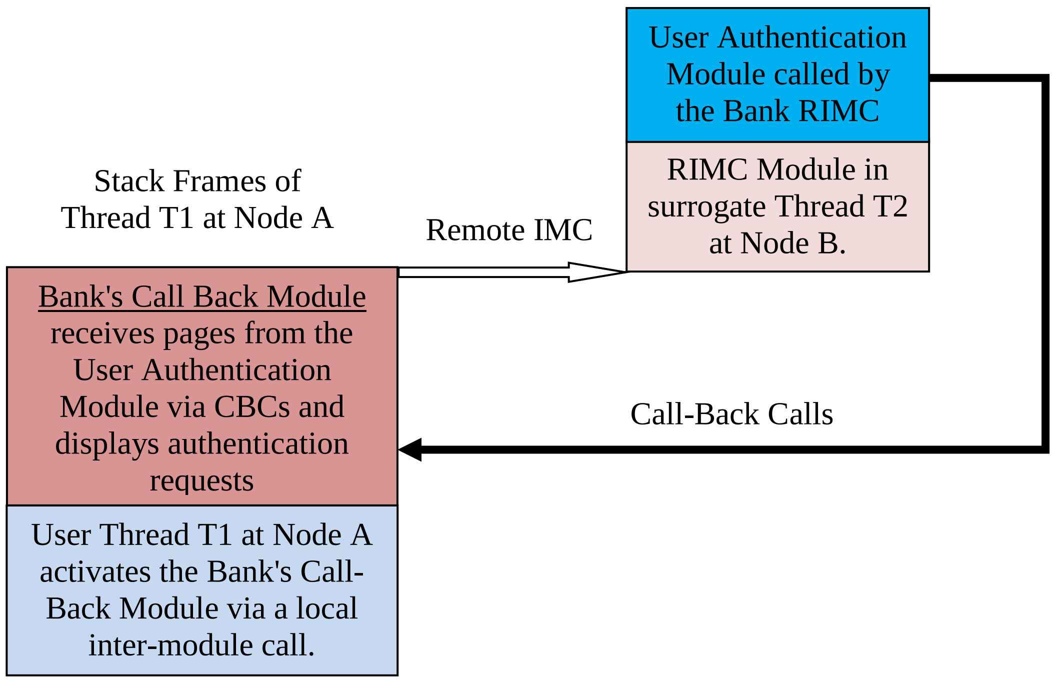

User Authentication Module called by the Bank RIMC
RIMC Module in surrogate Thread T2 at Node B.
Stack Frames of Thread T1 at Node A
Remote IMC
Bank's Call Back Module receives pages from the User Authentication Module via CBCs and displays authentication requests
Call-Back Calls
User Thread T1 at Node A activates the Bank's Call-Back Module via a local inter-module call.

Figure 37.6: Customer Identification via his Authentication Module

NOTE: The bank should of course treat the user's authentication module with suspicion and take steps to ensure that it is does not contain a trojan horse! To do this it could for example use

(a) the module call confinement rights (which it can unset in the capability used to call the authentication module and/or in the thread security register) and

(b) the thread control rights (which it can unset in the thread security register).

5.7 Displaying the Information

At first sight using the call-back arrangement over an entire banking session might appear to have the disadvantage that a potentially large amount of information might have to be transferred backwards and forwards over the Internet. But in fact this is not the case, assuming that the *structural* information of web pages is maintained at the call-back module, e.g. as formatted pages. For example in the case of banking the formats of the individual web pages are normally fixed. If the call-back module receives display requests from the website module in the form: <webpage number, parameter list>, all it needs to do is to insert the parameters (i.e. the actual values, e.g. of the user's account) into the pre-prepared web page formats and display the page. This is not only more efficient than sending the entire information over the Internet to display the page but it is more secure, since a hacker would have to understand the page numbers, the order and meaning of the parameters, etc. before he could use the information sensibly, and intercepting the call-back information would not help him to access the persistent information held at the website. The only exception to this might be the authentication page, since this might be user-defined. However, if the bank chooses, instead of offering complete freedom with respect to the use of authentication modules it might offer, say, ten standard alternatives.

When the user chooses to log out, the call-back module deletes the information held in its temporary segments then advises the bank module in the call-back call's return parameters. This results in the bank module finalising its activity and returning in the RIMC thread back to the call-back module and thence back to the user.

6 Online Shopping and Services

It has now become common practice to visit websites which offer goods and services for which the customer can immediately pay online. Payment methods can vary (and may involve risks, e.g. in the case of providing a credit card number[264]). However, in their morbid fascination for investment banking and their corresponding neglect of private customer banking, many banks have overseen the opportunities offered by online shopping and left the field to others. Here we show how the banks themselves could gain back some of the custom which they have lost.

6.1 The Basic Scenario

The scenario here envisaged is that a bank account holder goes online to shop for goods (or for services such as airline tickets). He begins by visiting the website of a business (the vendor) offering items of interest. To do this from his home computer (or from a smartphone) with a SPEEDOS system he activates a vendor call-back module on his own system (which he will have obtained in one of the ways described in chapter 35). The vendor call-back module, which has an embedded capability for its associated website, makes a remote inter-module call to the website, which then eventually makes a call-back call to the potential purchaser offering goods or services for sale. This process is repeated until the customer has selected an item or items which he wishes to purchase (see Figure 37.7). He signals this by pressing a "request payment details" button.

The call-back module then displays for him a price list. If the customer decides to purchase the item(s), he then presses a "pay" button. This causes the vendor's call-back module to display an invoice, which the customer can download if he wishes.

[264] see section 8 below.

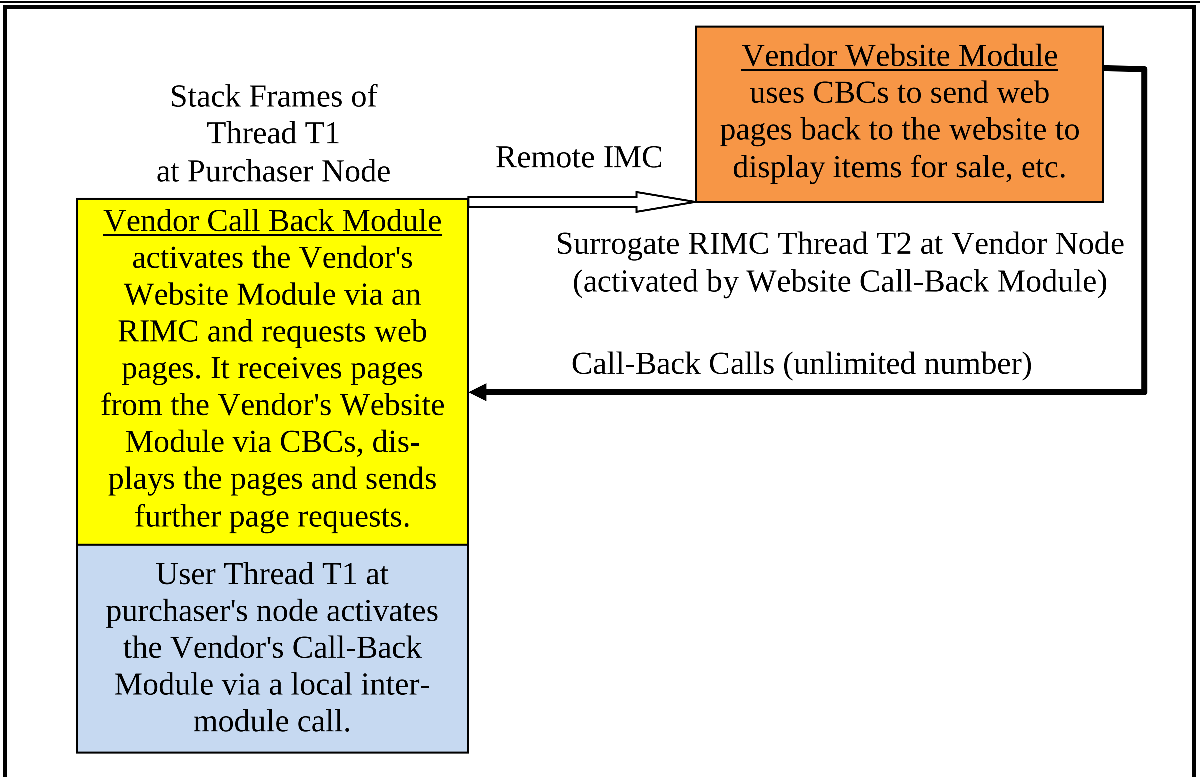

Stack Frames of Thread T1 at Purchaser Node
Remote IMC
Vendor Website Module uses CBCs to send web pages back to the website to display items for sale, etc.
Surrogate RIMC Thread T2 at Vendor Node (activated by Website Call-Back Module)
Call-Back Calls (unlimited number)
Vendor Call Back Module activates the Vendor's Website Module via an RIMC and requests web pages. It receives pages from the Vendor's Website Module via CBCs, displays the pages and sends further page requests.
User Thread T1 at purchaser's node activates the Vendor's Call-Back Module via a local inter-module call.

Figure 37.7: Online Interactions between Purchaser and Vendor

6.2 Activating the Bank's Online Purchase Mechanism

At this point it is appropriate for the vendor's website module, still acting in the customer's thread, to call the purchaser's bank to allow the payment to be authorised. But to do this the vendor needs access to a capability and the appropriate semantic routine number for the bank. This raises the question: how can the purchaser safely make an appropriate capability and semantic routine number available to the vendor?

On the one hand such a capability should be made available to the vendor only after the purchaser has decided to go ahead with the transaction, but on the other hand the flow of the transaction should not be disturbed. To achieve this, the purchaser activates a second thread (T3) which activates the *bank's* call-back module, using an **`online_purchase`** semantic routine, not the routine normally used for straightforward online banking.

Thread T3, executing in the bank's call-back module, opens a window which allows the user to select the capability for the vendor's call-back module from a directory in which he has previously stored it (e.g. as a bookmark). Using this (and a semantic routine number also provided in the window) the thread calls a **`payment`** interface routine of the vendor's call-back module, passing to it a capability for the bank and the number of the entry point in the bank module which handles payments (**`online_purchase`**), see Figure 37.8.

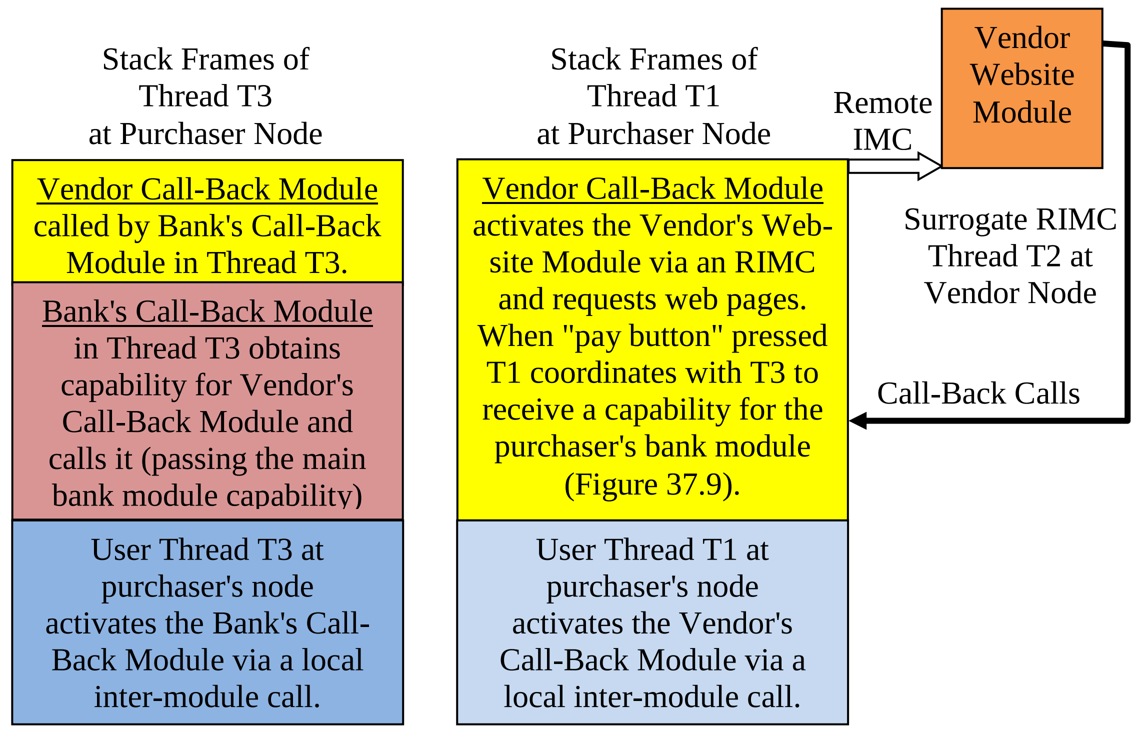

Stack Frames of
Thread T3
at Purchaser Node
Stack Frames of
Thread T1
at Purchaser Node
Remote
IMC
Vendor
Website
Module
Vendor Call-Back Module
called by Bank's Call-Back Module in Thread T3.
Vendor Call-Back Module
activates the Vendor's Website Module via an RIMC and requests web pages. When "pay button" pressed T1 coordinates with T3 to receive a capability for the purchaser's bank module (Figure 37.9).
Surrogate RIMC
Thread T2 at
Vendor Node
Bank's Call-Back Module
in Thread T3 obtains capability for Vendor's Call-Back Module and calls it (passing the main bank module capability)
Call-Back Calls
User Thread T3 at purchaser's node activates the Bank's Call-Back Module via a local inter-module call.
User Thread T1 at purchaser's node activates the Vendor's Call-Back Module via a local inter-module call.

Figure 37.8: Passing a Bank Module Capability to the Vendor (Step 1)

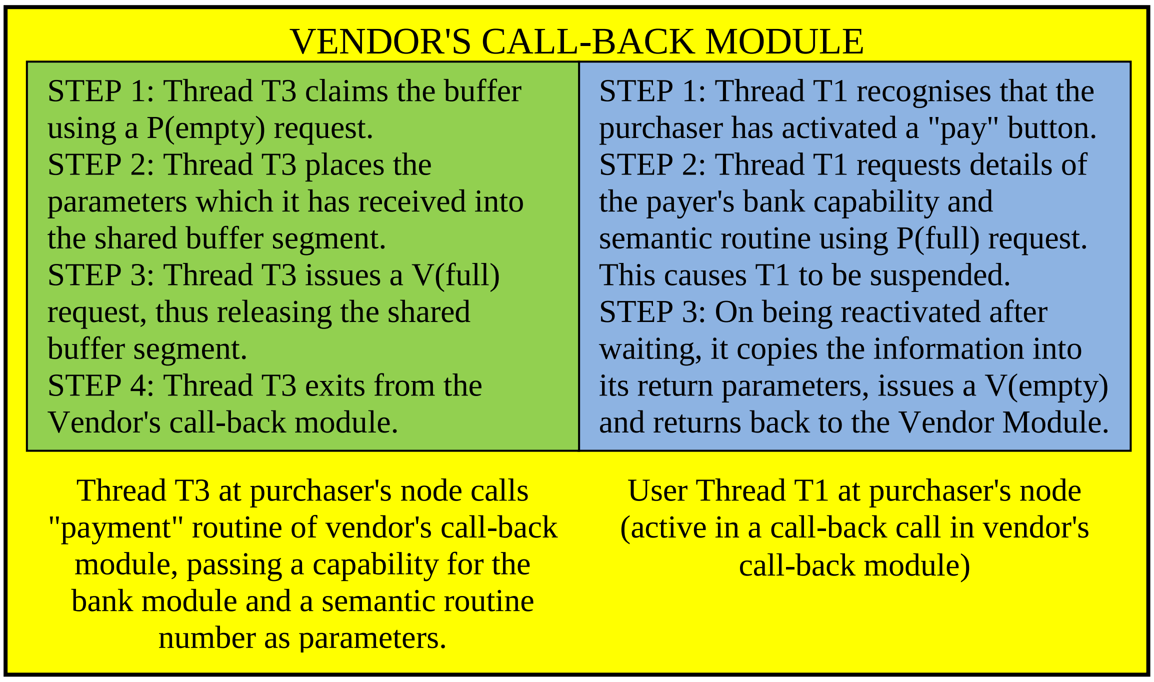
Now executing in the vendor's call-back module, thread T3 must somehow pass the bank capability and the entry point number to thread T1, which is currently suspended in the vendor's call-back module, so that this can return the bank's capability and routine number back to thread T2 in the vendor's website module, to enable it to call the **`online_purchase`** routine of the bank's main module.

The technique which I suggest is that the transfer is carried out using a simplified version of the producer-consumer algorithm[265], in which there is a single producer (T3) which "produces" a capability and a single consumer (T1) which is waiting to "consume" this.

This requires a single "buffer" (which in this case is simply a single slot in the capability partition of a segment of the call-back module and space for an integer in the data partition). The *empty* semaphore is initialised to 1 and the *full* semaphore to 0. T1 executes the P(full) protocol, which causes it to suspend until the buffer is full. When thread T3 arrives in the vendor's call-back module it executes the P(empty) protocol and moves the capability and integer into the buffer segment. The *nextfull* and *nextempty* variables are not needed. The V(empty) and V(full) operations are carried out as appropriate to ensure that both threads can continue. At this point T3 can exit from the vendor's call-back

265 see volume 2 chapter 8 section 12.1. Producer-consumer routines should be available in the privileged synchronisation library routines, see chapter 21.

module. Figure 37.9 illustrates these operations.



VENDOR'S CALL-BACK MODULE

STEP 1: Thread T3 claims the buffer using a P(empty) request.
STEP 2: Thread T3 places the parameters which it has received into the shared buffer segment.
STEP 3: Thread T3 issues a V(full) request, thus releasing the shared buffer segment.
STEP 4: Thread T3 exits from the Vendor's call-back module.

STEP 1: Thread T1 recognises that the purchaser has activated a "pay" button.
STEP 2: Thread T1 requests details of the payer's bank capability and semantic routine using P(full) request. This causes T1 to be suspended.
STEP 3: On being reactivated after waiting, it copies the information into its return parameters, issues a V(empty) and returns back to the Vendor Module.

Thread T3 at purchaser's node calls "payment" routine of vendor's call-back module, passing a capability for the bank module and a semantic routine number as parameters.

User Thread T1 at purchaser's node (active in a call-back call in vendor's call-back module)

Figure 37.9: Passing a Bank Module Capability to the Vendor (Step 2)

Thread T1, operating in the vendor's call-back module, is now in a position to pass the bank module capability and routine number as return parameters from the call-back call to the vendor's website module and await further call-back calls or a final return from the website module.

6.3 Making the Payment

On receiving the capability and routine number, the vendor website module, executing in the surrogate RIMC thread T2, uses this to make a remote inter-module call to the purchaser's bank module (see Figure 37.10).

In addition to using the capability and semantic routine number to make the call, it passes the following (or similar) information from the invoice as parameters:

a) the account details of the vendor (e.g. in Europe the IBAN and BIC numbers);

b) the name of the payee (i.e. the vendor);

c) invoice number (or similar) and date;

d) the name of the purchaser.

This takes it to the semantic routine `online_purchase` of the main bank module, which now receives the payment details that it requires to make the payment, but to go ahead the bank module still needs the authority of the purchaser (i.e. the bank customer).

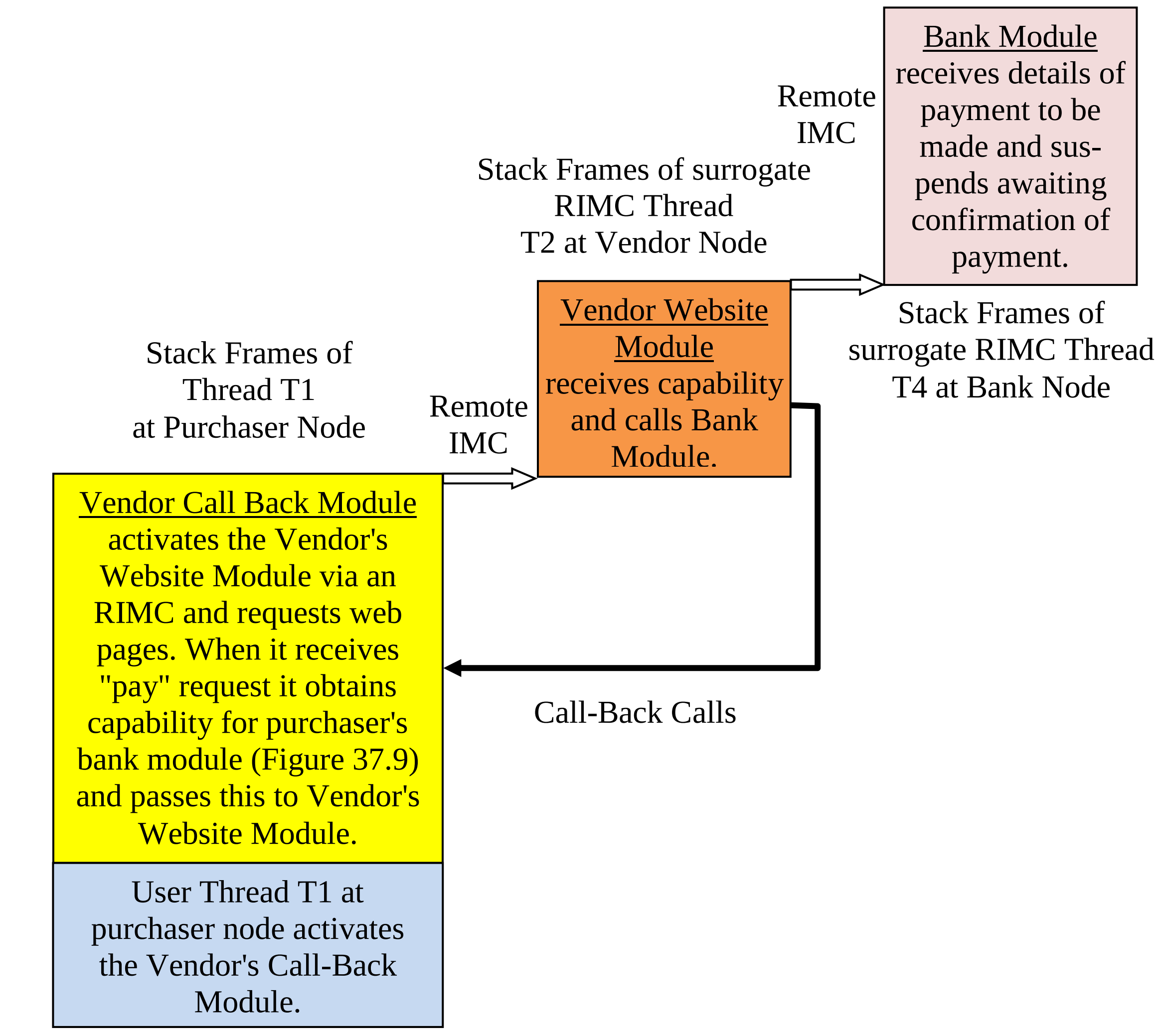

Bank Module receives details of payment to be made and suspends awaiting confirmation of payment.
Remote IMC
Stack Frames of surrogate RIMC Thread T2 at Vendor Node
Vendor Website Module receives capability and calls Bank Module.
Stack Frames of surrogate RIMC Thread T4 at Bank Node
Stack Frames of Thread T1 at Purchaser Node
Remote IMC
Vendor Call Back Module activates the Vendor's Website Module via an RIMC and requests web pages. When it receives "pay" request it obtains capability for purchaser's bank module (Figure 37.9) and passes this to Vendor's Website Module.
Call-Back Calls
User Thread T1 at purchaser node activates the Vendor's Call-Back Module.

Figure 37.10: The Vendor's Website Module calls the Bank Module

The simplest way to organise this is for the bank module to prepare a payment order and display this to the customer. This can be done via a call-back call to the bank's call-back module. However this must first contact the bank module, and the user must be authenticated.

6.3.1 Establishing Contact between the Bank and its Call-Back Module

We left thread T3 exiting from the vendor's call-back module, and returning to the bank's call-back module. This now makes a remote IMC to the bank module at the bank node (see Figure 37.11). The latter can then use call-backs to display information.

The resultant RIMC thread T5, executing in the bank module, can following similar authentication procedures to those described in sections 5.4 to 5.6. Once authenticated, the bank module can make a call-back call to its partner thread T3 in the bank call-back module, allowing the purchaser (i.e. the bank customer) to select an online purchasing page which displays the invoice to be paid. Assuming that this is correct, the user can then click a "confirm payment" button. T3 can then return from the call-back call, indicating whether the user has logged out from the bank session or wants to continue with other online banking.

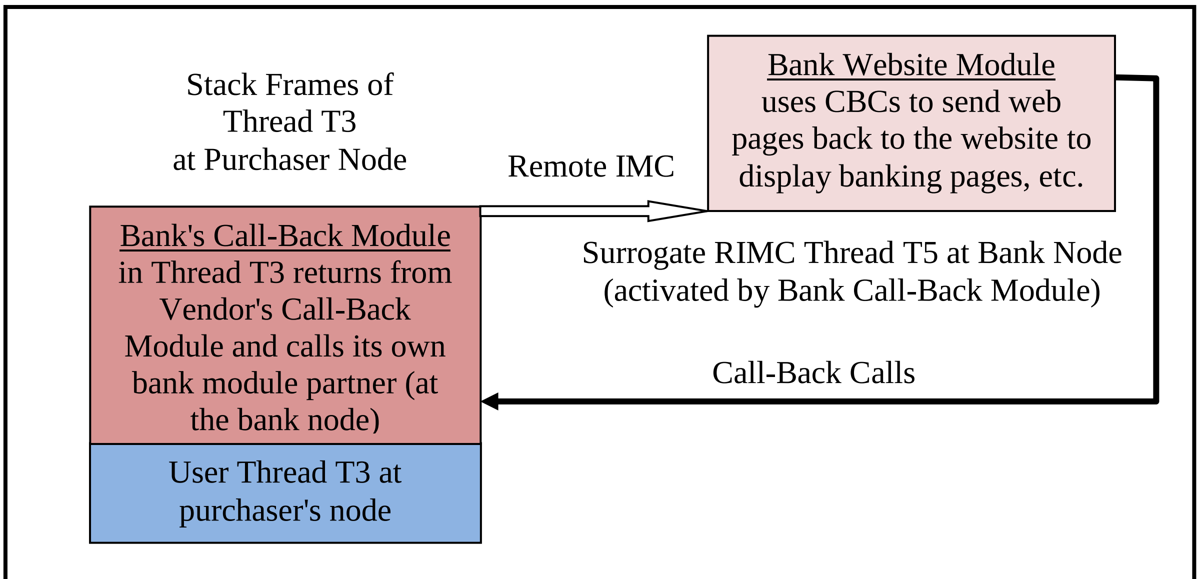

Stack Frames of Thread T3 at Purchaser Node
Remote IMC
Bank Website Module uses CBCs to send web pages back to the website to display banking pages, etc.
Bank's Call-Back Module in Thread T3 returns from Vendor's Call-Back Module and calls its own bank module partner (at the bank node)
Surrogate RIMC Thread T5 at Bank Node (activated by Bank Call-Back Module)
Call-Back Calls
User Thread T3 at purchaser's node

Figure 37.11: The Bank's Call-Back Module calls the Bank Module

6.3.2 Coordinating with the Vendor in the Bank Module

From the vendor's viewpoint the transaction is not complete until the vendor receives a confirmation that the payment has been made. Thread T4 provided the details needed to pay but before he can get the necessary confirmation T5 must confirm the payment. We now have a similar situation in the bank module to that which occurred in the vendor's call back module (see Figure 37.9), where the two threads (T1 and T3) needed to pass information in a synchronised manner. In the present case the information is a simple confirmation/reject message regarding which the two threads T4 and T5 must be synchronised, but this time in the bank module. The same basic solution as shown in Figure 37.9 can be used. In this case T5 is the producer (it produces a confirmation) while T4 is the consumer waiting for the confirmation. Hence T5, after it has confirmed the payment, executes a P(empty) request and places an evidence of payment in the single buffer then executes a V(full) statement. Meanwhile T4 executes a P(full) request, removes the confirmation from the buffer and issues a V(empty) statement. T4 can then place the confirmation into a return parameter before exiting from the bank module back to T2 in the vendor's module. This can exit back to T1 at the purchaser's node. T5 can either carry on banking or can exit to T3 and return.

6.3.3 Comments on the Online Shopping Approach

The user must have a SPEEDOS online banking facility on a home computer, a smartphone or other computing device, but the solution described above does not require registration with an outside authority or company. Nor does it require either the purchaser or the vendor to pay any "middleman" fees over and above the normal cost associated with online banking. The only requirements are that (a) the vendor provides two standardised interfaces for his website call-back module, and that the bank provides a special interface routine in the main bank module which allows vendors to present invoices for payment.

Notice also that the scheme is not limited to payments by banks. Any financial company (e.g. a company offering a credit scheme) can use the same model to allow its customers to make online payments to businesses (without the necessity for passing credit card numbers around).

This example provides a good illustration of how the in-process system works in practice in SPEEDOS. All the threads described are owned by the purchaser (who is also the bank customer). On the one hand the websites visited have the advantage from this that they can easily check the identity of the user on whose behalf they are working. But on the other hand the user can retain considerable control via the thread security register, the settings of which can be determined by the user.

7 Direct Debit Facilities for Recurring Payments

Direct debit facilities allow payees to withdraw money directly from payers' bank accounts. This usually means that the payer has previously provided an authorisation which specifies a number of rules that might define the frequency, a maximum amount per withdrawal, what happens when the payer's account does not have sufficient funds, what fees are involved, whether and within what time limit a the payer can re-claim an incorrect withdrawal, etc. In some systems the payer can object to a withdrawal and can reclaim the amount withdrawn within a specified period from the date of the withdrawal (e.g. 6 weeks in Germany).

Such facilities are widely used in financial systems, especially in situations where regular payments which vary in amount from month to month (e.g. telephone bills, heating bills) must be made by private individuals to companies.

The permitted rules are usually defined by the country in which the transactions are planned to take place, and transactions are normally restricted to that country. These rules vary very substantially from country to country[266]. In view of this and the fact that end-users are not directly involved online once the system has been set up, I make no attempt present how a system might look in SPEEDOS, except to say that it will obviously involve accesses to user accounts and online transactions between banks, which can obviously be implemented in

[266] see https://en.wikipedia.org/wiki/Direct_debit

SPEEDOS, and such implementations can of course take advantage of the SPEEDOS security mechanisms.

8 Banking Cards and Mobile Banking

The conventional solution for most mobile banking activities before the advent of smartphones was the use of debit or credit cards (which I shall refer to collectively as banking cards), and these still have an established place in current banking practice, e.g. for using at ATMs and for paying bills. Not all banking card systems work in the same way and in some cases they are country specific[267], so I will make some general assumptions.

A debit card is linked directly to a bank account, and it can only be used for making payments which can immediately be fully paid from the bank account's current credit value (including in some cases a small overdraft limit). Payments are made immediately from the bank account, using a radio or other link to the bank. Debit cards are usually protected by a short (mostly four digit) personal identification number (PIN).

A credit card is linked to a credit institution, to an account which is recognised from a credit card number, and payments are also operated using a radio or other link to the credit institution[268]. The credit card holder usually receives a monthly statement but has a credit limit, so that instead of paying the full amount he has the option of paying a part of the debt and carrying over the rest of the debt (on which he pays interest and fees) successively to the following month(s).

Some credit cards have an associated 3 digit security code on the credit card, which is known to the merchants and can be used to ensure (e.g. over the telephone) that the user actually has the card which he claims to have.

Banking cards can be a source of problems, as we now discuss.

8.1 Some Problems with Paying by Card

One very significant issue is that cards store information needed by the bank or credit institution to carry out the banking operations which the customer wishes to make. On earlier cards this information (e.g. account number, withdrawal limit, etc.) was recorded on a magnetic strip, which could be read by a payee with the appropriate device. If the payer and payee were physically in the same place, there was the added security that the card also held a visible signature of the payer, so that the payee could convince himself of the payer's identity by com-

[267] for more details see e.g. https://en.wikipedia.org/wiki/Credit_card and https://en.wikipedia.org/wiki/Debit_card

[268] In earlier systems, transactions were carried out on paper using a mechanical device and carbon paper to capture the details (including an embossed credit card number).

paring this with his signature on the payment order which he signs. In later versions of banking cards the magnetic strip was replaced by a microchip. Encryption was introduced to make the process more secure.

One fundamental weakness of banking cards is that to be useful the information on the card can be read by a device provided by banks to payees (e.g. merchants). But if the payee's device can read the information on the card it is clear that enterprising thieves could also find ways to read or copy the card and use this information to steal money from the card holder's bank or credit account. Since in many cases such theft takes place without face-to-face interaction, the signature on the card is also no help, and even in face-to-face interactions some thieves are good forgers!

Even less secure are transfers where the payee simply provides (e.g. over the Internet or by telephone) a credit card number. And in this case the problem arises that when a payee receives a credit card number, he can store it for future use. This then leads to a problem that when a business, a club, or other organisation stores credit card numbers on its computers, these become a tempting target for thieves. Over the last decades there have been numerous examples of break-ins to the computers of organisations which hold vast files of credit card numbers of their members or customers and the theft of these card numbers.

The addition of a radio facility to banking cards (contactless smart cards[269]) is even less secure, since a clever thief can obtain the card details by holding his reading device within distance of the radio waves, e.g. by reading the information from the pocket of a user in a crowd or in a supermarket queue. It would obviously be desirable therefore completely to eliminate the need for debit and credit cards.

9 Automatic Teller Machines

When a bank customer wishes to withdraw cash from his account, it has become the universal practice (because banks want to save on staff salaries) to use automatic teller machines (ATMs). To do this the customer inserts a card into the ATM and typically provides a four digit PIN number as verification of his identity.

As its name implies, an ATM has more or less the same functionality as a human bank teller. Probably its most important function is to allow customers to withdraw money but it can also be used for other functions such as transferring money from one account to another (of the same or a different person). Hence

[269] https://en.wikipedia.org/wiki/Contactless_smart_card, https://en.wikipedia.org/wiki/Contactless_payment, and https://en.wikipedia.org/wiki/EMV

logically it can be viewed as being similar to online banking, except that while online banking allows a customer to transfer funds between users and also between the accounts of a user it does not provide a mechanism which allows the customer to obtain real cash. That is perhaps the key difference from ATMs, which have a mechanical mechanism that allows a customer to withdraw money. The question therefore arises to what extent ATMs could follow a similar implementation pattern to online banking, but with an extra cash mechanism added.

9.1 An Approach Using Cards designed for SPEEDOS Systems

There are some interesting similarities. An ATM has a display screen and a keyboard (i.e. a limited set of input keys) via which the user can input his transaction requests and see the results; this is in principle similar to the online user at his home computer or his smartphone. Is it therefore possible to envisage that the ATM's software can have more or less the same functionality as a bank call-back module and that the customer's transactions are managed by remote inter-module calls from an ATM's local call-back module and the central bank module?

In this case the software in the ATM is comparable with the call-back module described for online banking. It can store the *structural* information required for display and the bank's central module can be reached via a remote inter-module call, which returns information as a <page number, parameter list> for display on the ATM screen.

We therefore assume that the ATM software includes a (rudimentary) SPEEDOS system with a call-back module which is used in a similar manner to that for online banking. However, there is a fundamental difference. Because the ATM uses the bank's own device, the implicit checks as described in section 5.5 cannot be carried out, and the kernel's environmental instruction `current_thread_owner`, which allows the online version to locate the correct customer information, cannot be used to identify the customer. (It could however be used to identify the ATM.)

One possibility for identifying the customer would be to fall back on a currently used mechanism (e.g. a card with a PIN number – but preferably with a much wider choice of PIN values!) and to use the information stored in the card's microchip (which could include the customer's unique SPEEDOS identifier). I regard this as a fallback solution, for the case that banks are unhappy about the following alternative proposal.

9.2 An Approach without the Use of Cards

A better alternative, in my view, is simply to drop the implicit checks described in section 5.5 but to carry out (a) a check that the request is from a recognised ATM and (b) to follow similar procedures to those described in sections 5.6.1 and 5.6.2. The aim is still to satisfy both the bank and the customer about the validity of the ATM user's identity and in this case to allow the bank to discover the customer's unique SPEEDOS identification, which can then be used to access his accounts, etc.

Notice that the apparent alternative of initially requesting the ATM user to provide his unique identifier alone at the start of an ATM session is not adequate, because this is not kept secret. It can be used as part of the identification process to locate the (alleged) user's records (see section 4.7) but further proof is needed for the bank to be sure that the ATM user is who he claims to be. The bank can then devise its own identity tests, which can vary from customer to customer and might, for example, include a PhotoTAN or similar procedure.

After the bank checks have succeeded it is desirable that the bank software should also use the customer's own authentication module to ensure that the user is really who he claims to be. In this way the user of the ATM cannot gain access to the bank accounts without passing the real checks provided by the user himself. This should make the use of ATMs more secure than is presently the case with checks which simply rely on a card that uses only a short PIN number.

10 Conclusion

This chapter has illustrated how a secure banking system might be organised, specifically with respect to bank accounts and to online transactions. Many other file modules would be needed (e.g. to list the customers who have safe deposit boxes, to organise telephone banking, to pay staff salaries, etc.) to complete the system.

I have restricted the discussion largely to online mechanisms using home computers and smartphones, because this is where I see the future of banking systems. In most situations smartphones will probably largely replace the use of card based payment systems, which, as I have indicated in section 8.1, are inherently insecure. It would of course be possible for card based payment methods to be integrated into SPEEDOS, along similar lines to those proposed for ATMs. This could perhaps be organised by providing merchants who want to use payment cards with a device into which cards are inserted which, as I suggested for ATMs, would in effect have an in-built rudimentary SPEEDOS call-back mechanism in which the card user identifies himself using a PIN in combination with a unique user identifier, or preferably a mechanism whereby a simple user authentication module (e.g. with a dynamically changeable PIN) could play a role.

Our purpose, as in the entire book, has not been to provide a complete and detailed specification of a system, but rather to indicate the main facilities available to a software designer in a SPEEDOS system and to show how they can be used in practice. It would be the function of the designer of a specific system to consider how the SPEEDOS protection mechanisms (including for example module capabilities, the thread security register, bracket routines, call-back module, the environmental checks, etc.) can best be used in his system design, whether in a banking system or some other system. Finally it is worth recalling in the banking and other contexts that all Internet communication between SPEEDOS computers is automatically encrypted by the kernel, and that all SPEEDOS discs are also automatically encrypted.

Chapter 38
Making Life Difficult for Hackers

In this concluding chapter we recall some of the key ideas in SPEEDOS which are not found in other operating systems/kernels. These are the main ideas which help to explain why I consider that SPEEDOS systems will be considerably more secure than operating systems which are based on conventional computer designs.

1 The Hardware and Related Features

As the first volume explained in considerable detail, the hardware modifications required to support a SPEEDOS kernel appear to be fairly trivial but are absolutely essential. The key difference is in the way that the virtual memory is organised and addressed. Without this, it would be extremely difficult to have an efficient SPEEDOS system.

1.1 The Orthogonal Segmentation and Paging Model

Chapter 11 of volume 1 explains this model, which combines logical segments and physical pages in such a way that these two units for implementing virtual memory are almost completely independent of each other. From the efficiency viewpoint all accesses to the memory are paged, but from the viewpoint of the user (and of the compiler) all information is held in segments, without the one being dependent in a direct way on the other. Page boundaries do not determine where segment boundaries are placed, and segment boundaries do not determine where page boundaries occur.

This may at first sight appear to be just a matter of efficiency – and it does imply that in this respect SPEEDOS will be more efficient than systems which use a conventional paging or paged segmentation scheme. But that is not the key issue. This scheme makes it possible in practice to support both very short segments and very long segments not only efficiently but also in such a way that memory protection can be equally well applied to both, since the memory protection scheme works at the level of segmentation – regardless of segment size. This crucial factor means that small segments can contain significant protection information, including capabilities, and that access to these can be controlled exclusively by the kernel, thus laying the fundamental basis for security, protection and privacy. Conventional schemes for segmentation and/or paging are not able to do this.

1.2 Segment Registers

A crucial further step is the introduction segment registers. Segments can be accessed only via segment registers. These are hardware-based addressing registers which allow the hardware to check a program's right to access a segment and the mode of this access during program execution. In conventional systems such checks normally use information stored in page tables, thus restricting the minimum size on which protection can be applied to an entire physical page. In SPEEDOS this restriction does not exist, making it much easier to apply protection to logical segments, regardless of their size.

Segment registers can only be loaded and manipulated by the kernel, thus making it possible in SPEEDOS to eliminate conventional segment tables completely. Instead SPEEDOS uses protected pointers to organise the logical structure of programs and data, e.g. as linked lists. This simplifies both the address translation hardware and the structuring of software. And above all it allows the kernel to support protected capabilities, as will be discussed shortly.

1.3 Persistent Virtual Memory

SPEEDOS realises the concept of direct addressability, which was an aim of the Multics project, and extends the idea such that a separate file system is entirely eliminated; all information, whether persistent or temporary, is held in the virtual memory. Once again this increases the efficiency of the system, because only one set of mechanisms is required to manage the persistent memory, for which the main memory is simply seen as a "cache". Apart from the efficiency advantage of this approach, a single uniform set of security mechanisms applies, in contrast with conventional systems, where separate mechanisms are needed in the conventional virtual memory and in the file system.

1.4 Distributed Persistent Virtual Memory

The next significant step is the application of the same persistent virtual memory concept to all SPEEDOS computers throughout a network (including the Internet). This is achieved in practice primarily by allowing inter-module calls – which are the heart of the SPEEDOS protection mechanisms – to be made to modules at remote SPEEDOS nodes in a network. This has the effect that a single set of mechanisms applies for SPEEDOS computers throughout an entire network.

1.5 Encryption of the Virtual Memory

The final unusual aspect of the virtual memory is that all information transferred between SPEEDOS computers across a network is automatically encrypted, and similarly all information stored on disc is likewise encrypted. Thus unencrypted information appears only in the main memory.

2 Some Key Kernel and Operating System Features

The next step is to review how the SPEEDOS kernel makes life very difficult for would be hackers.

2.1 The Information Hiding Module Structure

All software units in SPEEDOS are constructed as modules. These consist of a code part, which has multiple entry points, and (optionally) a data structure part. Together these can model any software structure in conventional systems.

The most common form of module is a file module, i.e. a module consisting of (a) a persistent data structure and (b) a number of routines which provide semantic services for the data structure. The only way of accessing the data structure from outside the module is via its semantic routines, which have predefined functionality. This alone is a hindrance to hackers, as we now show.

In conventional systems a capable hacker can write a program which directly accesses the information of interest to him stored in a file in the file system as Figure 38.1 illustrates.

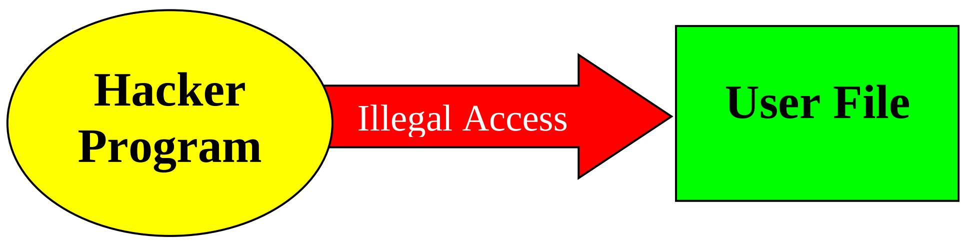


Figure 38.1: Hacking a File in Conventional Systems

This is not possible in SPEEDOS because the file data can only be accessed via its semantic routines, as Figure 38.2 shows.

At a more technical level, what prevents direct access to the file is that this nowhere appears in a directory, in contrast with conventional systems. The issue then becomes: How can a hacker attempt to find the data which he wishes to access. This issue is discussed in section 2.3, but first we must consider a special potential risk.

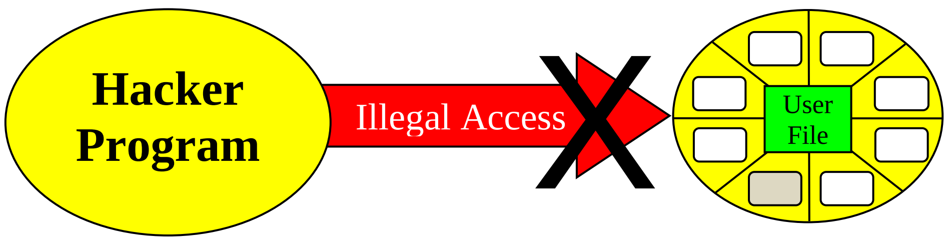


Figure 38.2: The File Information is protected by the Semantic Routines

2.2 Free Capabilities

In section 2.1 we ignored one possibility which a hacker might attempt to use in his attempts to by-pass the security of a system, viz. free capabilities. These allow a thread to call a module which is given read access to the root data structure of a different module (see chapter 18 section 8, chapter 19 sections 13.5 and 14) and this will undoubtedly be a mechanism which hackers try to use to their advantage. However, the misuse of free capabilities can easily be prevented by the owner of a file unsetting the capability use right "Free Cap") as is described in chapter 26 section 2. Once a right has been unset, the kernel will under no circumstances permit it to be reset. An aid for programmers which helps them not to forget this issue is described in section 2.7.

2.3 Directories and their Structures

In conventional systems directories are part of the file system and as such have a predefined structure which is certainly known to hackers and helps them to locate the information to which they seek access. But this is not necessarily the case in SPEEDOS.

As is described in chapter 30, directories in SPEEDOS are simply modules which can be designed and implemented like any other module. One implication of this is that a SPEEDOS node which is likely to become a target for hackers can design and implement its own directory modules in any way that it chooses. These need not contain the same entry structures nor offer the same services in their semantic routines nor even use a standard set of entry point numbers for obtaining these services.

This alone makes life difficult for a hacker, but even more security is offered by the fact that the semantic routines of user-written directories can themselves contain checks to ensure that hackers cannot reach the information for which they are searching. For example they might contain environmental checks (see chapter 26) which examine the unique identities of the owners of the threads calling them against a list of users permitted to access them (which might also list the individual entries which these users are permitted to access). Alternatively – and more flexibly – they can place such checks in bracket routines (to be discussed later).

Perhaps most significantly, a careful user can embed the capabilities for his more sensitive data in segments of his normal user programs (e.g. those which need access to the sensitive files). This makes them virtually unfindable for a hacker who does not have insider information.

2.4 Capabilities

Capabilities are the fundamental key for accessing information in a SPEEDOS system. This means that protecting them is a very important issue. As we have just seen, directly protecting capabilities held in directories can make life very difficult for hackers. But this does not mean that they are entirely safe.

The biggest insider risk is that a user who has legitimate access to a capability either carelessly or knowingly passes copies of this to unauthorised persons. The system has a number of mechanisms which can help to prevent this from happening. For example the capability itself contains a substantial number of metarights which can be used to prevent or limit the further distribution of capabilities, including restrictions which prevent the passing of capabilities between users, between different kinds of segments of a user and/or between different nodes. A restriction can also be applied that a capability can only be used once (and is then invalidated). A special mode is also provided to restrict a directory from accessing the capabilities stored in it. These are just a sample of the available controls over capabilities, which are described in detail in chapter 26.

If a hacker does somehow obtain a capability, the kernel ensures that he cannot increase the rights to allow him greater access than the capabilities already contain, since this is a general rule for all rights.

But the hacker who has gained access to a capability can read the unique module number of the module addressed by the capability (by copying the capability to a data segment). From this he could then try to find the file module behind the capability. On an inter-module or similar call the kernel finds the module (which it recognises as a file module from the type field in the capability) by first establishing the number of the *container* in which the module is located, and reading in its page 0. From this he obtains the start address of the module's data root by examining the container's Co-Module Table. These are all internal kernel operations which cannot be imitated by a hacker unless he can turn on privileged mode and thus access the kernel's tables (including making a disc access). There is no way that a hacker could achieve this under normal circumstances and so we conclude that even a knowledge of the capability's content will probably not help him.[270]

[270] Kernel security is discussed in section 3.

2.5 Controlling Access and Confining Information and Capabilities

Even if a hacker manages to gain access the system his access can be severely restricted (and he may be detected before he can steal data) by means of bracket routines (see chapter 24). These are a very flexible tool for use against hackers. For example call-in brackets can be used to record information about the callers of a bracketed module, to reject calls and even to serve as decoys which feed false information back to hackers calling the module. They can also monitor the information (including capabilities) which is being passed back to a calling module and either deliberately falsify this information or invalidate the capabilities (and even replace them with capabilities for decoy modules). Similarly call-out modules can examine the information (including capabilities) which a module is illegitimately passing on to a further module. Thus bracket routines (of both types) can easily be used to confine information.

2.6 The Process Structure

In order to execute code on a SPEEDOS system a hacker needs a thread which executes his code. But before this is even possible he must develop and install a code module which the thread can call. And then he needs a process module. All these stages present him with difficulties if he is not an accredited user at the node concerned, since the process and thread creation mechanism only works on a local node. SPEEDOS provides no mechanisms for simply writing code which can then be executed.

If in a multi-user system a hacker tries to highjack a persistent thread belonging to another user, he will have great difficulties in logging in (provided of course that the user has taken the trouble to use some carefully designed authentication modules, which are tailor-made to suit his needs, see chapter 22 section 11). His first problem is that there is no central repository of authorisation information which he can target (in contrast with conventional systems) and these is no standard procedure to be followed when logging in (again in contrast with conventional systems). Consequently he has no standard starting point for carrying his hacking operations.

And even if he succeeds in hijacking one thread, the user can protect each of his threads in a different way.

2.7 Environmental Information

A fundamental difference between SPEEDOS and conventional systems is that SPEEDOS requires each user to have a worldwide unique identity. This is stored in a place accessible only to the kernel in each container which a user creates (including process containers). Software in normal modules, including bracket routines, can check this (provided that they have an appropriate kernel capability) and so identify the user executing the current thread or the owner of the currently active module (or of the module which called this, or the module about to be called, etc.). This mechanism can be used in a straightforward way to check access permissions, but it can also be used to identify hackers (or discover whose threads they have hijacked) and to identify spammers, etc.

2.8 Simplifying the Setting of Rights

One of the risks which might face users is that they make a mistake in the unsetting of rights, e.g. in capabilities and in the Thread Security Register. This could lead to serious problems, for example if a user forgets to unset the "Free Cap" right (see section 2.2 above) in a capability which falls into the hands of a hacker.

Although the following solution is not a part of the kernel it is strongly advised that the developers of a SPEEDOS operating system should provide a carefully thought out set of template masks, i.e. bit patterns which can be used to "and out" access rights in capabilities and in the Thread Security Register. These should represent typical patterns of usage in common situations and be given easily understandable names. They could be stored in a module which is publicly accessible via a set of semantic routines which accepts a capability (or a bit string representing the TSR) as an input parameter, and return this capability/TSR string) with appropriately reduced rights as a returned value. The name of each routine should reflect that of the mask which it uses (e.g. **`unsetFreeCap`** or **`setFreeCapOnly`**). Such a mechanism, which is carefully designed, would not only be of assistance to users[271] but would also make the system more secure.

3 Securing the Kernel

The discussion so far has assumed that the kernel at a node being attacked by hackers is completely secure. Under what conditions can this assumption be realised?

3.1 Correct and Accurate Code

So far we have assumed that the kernel code is both correct and accurate. Since no formal specification exists for the kernel there can be no formal proof that it correctly fulfils a specification. It would of course be good if such a specification and proof were to exist, and I would certainly welcome any attempt to achieve this.

However, I consider it more important that the code is accurate, i.e. it does

[271] This is my response to Lampson's rather negative comments in [29], where he points out that users see the management of security as a "pain" (see chapter 5, section 7.2).

what it is *intended* to do, nothing more and nothing less. Formalists will argue that this is impossible without a specification. However, the whole purpose of this book has been to explain the *intention* behind SPEEDOS. Parts of the description may be unclear and some details have not been completely stated, but I feel confident that readers who carefully study the text will understand my intentions, and although some details will have to be filled in by implementers, I am confident that an accurate implementation can be achieved.

Of course an implementation of the kernel code must be exhaustively tested and only released when the results of tests are repeatedly positive, and after a professional hacker team has attempted to discover errors.

3.2 Secure Installation of the Code

It would be problematic if at a node which claims to be a SPEEDOS node the code is not a true version of the SPEEDOS kernel. In order to prevent this from happening (to the MONADS-PC, forerunner of SPEEDOS) a technique was devised which uses public key encryption to ensure that a system can be safely booted by the correct kernel [24].

3.3 Human Aspects

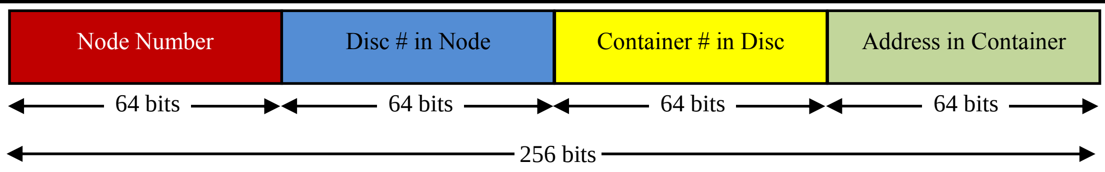
The least secure part of any operating system, including a SPEEDOS system, will almost certainly be the human element. Users can accidentally or deliberately introduce errors in the settings for the capabilities which they distribute to others or in their settings for the Thread Security Register, etc., and these might lead to security breaches.

Companies, public utilities, government espionage agencies, etc. might deliberately place their own employees in positions of trust at target nodes, so that they can provide these with insider information. Thus there can be no guarantee that any system is totally secure.

But SPEEDOS at least provides tools which can be used to minimise any damage, e.g. by extensive use of bracket routines to log activity on sensitive modules and to inhibit the transfer of information using confinement techniques described in Part 6, including the use of settings in capabilities, the Thread Security Register and the container confinement rights which can, for example, prohibit the sending of information and the use of remote inter-module calls to other SPEEDOS nodes.

APPENDIX
Formats of Some SPEEDOS Structures

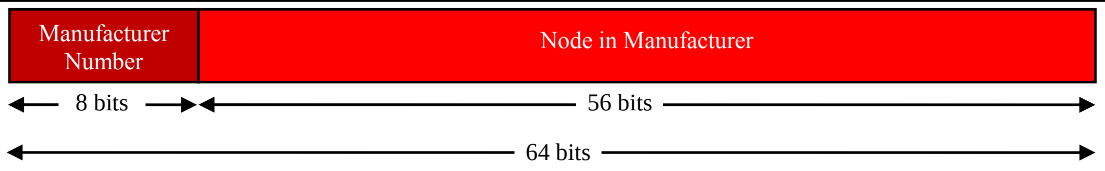
1 A Worldwide Unique Virtual Address



A SPEEDOS Full Virtual Address

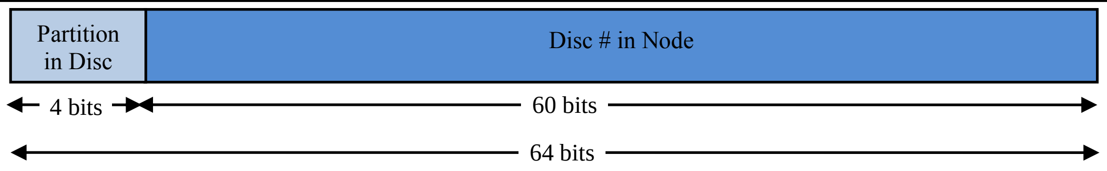
The main fields are subdivided into subfields as is indicated below. Some bits are spare.

2 Subfields of a Node Number



A SPEEDOS Node Number

3 Subfields in a Disc Number



A SPEEDOS Disc Number

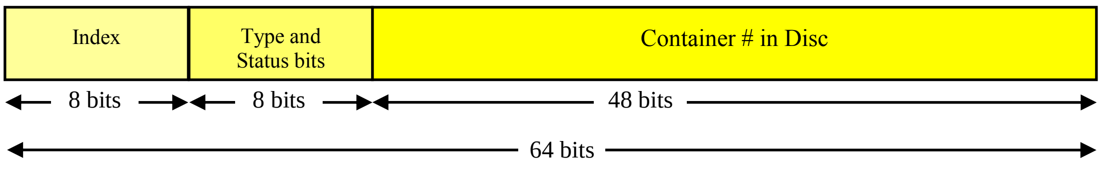
A physical disc can have up to 15 partitions (logical discs). If the partition field is all zeros, the disc number refers to the disc as a whole or to its initial "partition".

4 Subfields in a Container Number

The index field in a SPEEDOS container number indicates a data or code module number or a thread number within the container, as indicated by the type field of a capability.

The type field uses three bits, with the following meanings:

{kernel, data, code, thread, process, disc, container}

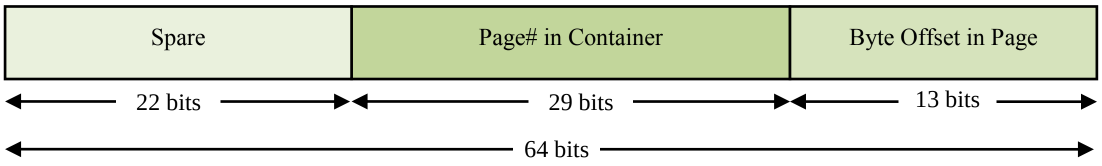


A SPEEDOS Container Number

A further bit indicates whether the capability is INVALID.

The status bits consist of two bit pairs (the capability origin bits and the copy restriction bits), as described in chapter 26 section 3.4.

5 Subfields in a Within Container Address

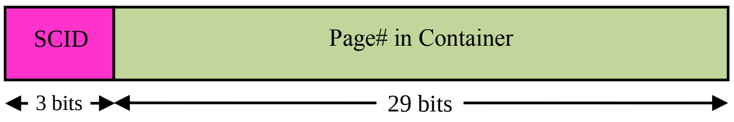


A SPEEDOS Within Container Byte Address

This allows for a container to be up to 4 TB in length, which would allow an entire 4 TB disc to be viewed as a single container.

6 A Virtual Page Number used by the ATU to Translate Short Container Addresses

To enable a conventional address translation unit to be used, the page number presented to the ATU is a 32 bit "page number", consisting of a 3 bit SCID (Short Container Identifier) and the Page Number within Container:



The Page Number Presented to the ATU

The proposed values for the SCID are as follows.

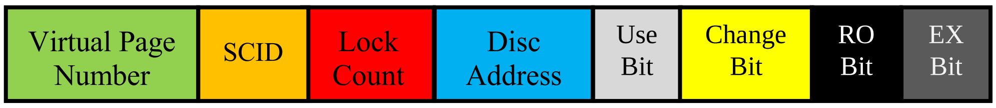
000 identifies the process address space of the currently active thread.
001 to 011 identify the currently active code address spaces, i.e. for the main code address space and up to two active code libraries.
100 to 111 identify up to four data address spaces.

A Possible Allocation of Short Container Identifiers

7 The Main Memory Page Table

In view of the size of modern main memories the hardware TLB (Translation Lookaside Buffer) will probably not be large enough to hold an entry for each page currently in main memory, so that this is supplemented by a Main Memory

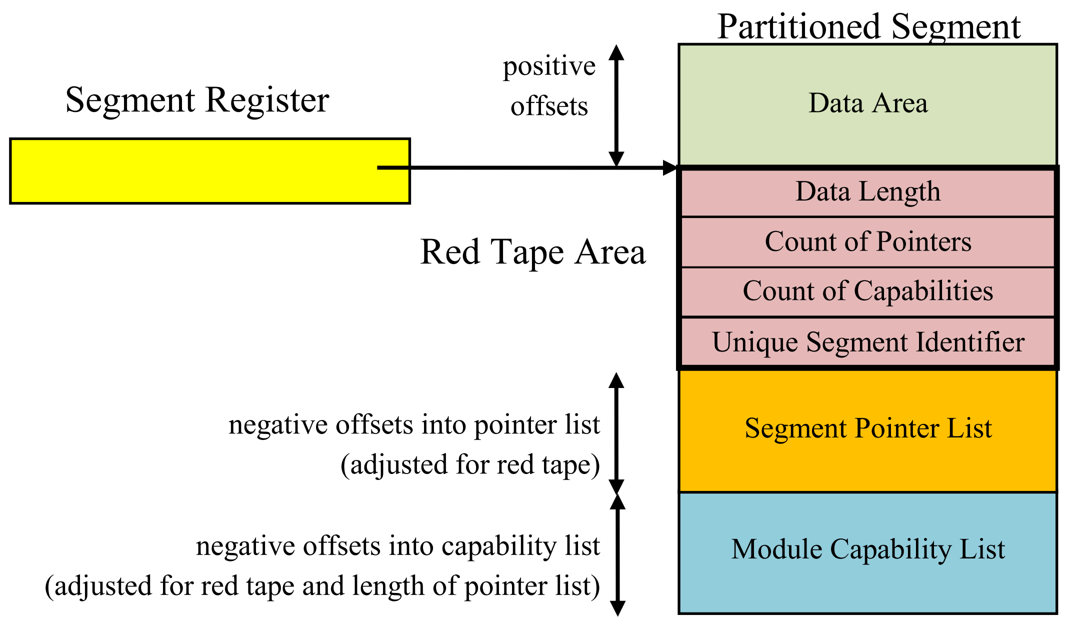
Page Table (MMPT) held in the kernel.



An Entry in the Main Memory Page Table

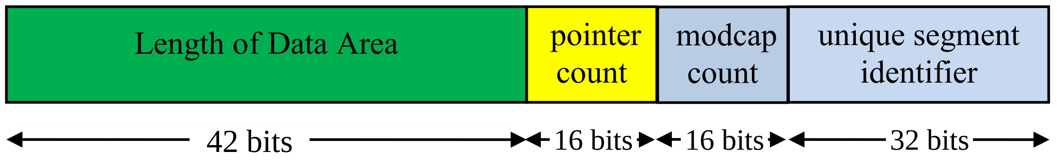
8 Segment Structure

A segment holds user information in 3 segment partitions: a data area, a list of pointers to other segments in the same container and a list of module capabilities. It is addressed via a (protected) segment register, which addresses the base of the data area. The data area is directly addressed via a segment register.


Partitioned Segment
Segment Register
positive offsets
Data Area
Data Length
Count of Pointers
Count of Capabilities
Unique Segment Identifier
Red Tape Area
negative offsets into pointer list (adjusted for red tape)
Segment Pointer List
negative offsets into capability list (adjusted for red tape and length of pointer list)
Module Capability List

SPEEDOS Partitioned Segments

Immediately below the data area is a red tape area, which can only be accessed by the kernel, using negative addresses from the base of the data area. It consists of two 64-bit words:

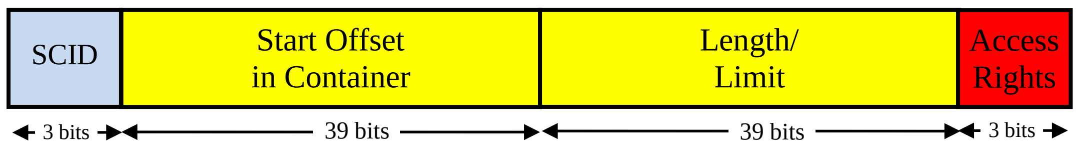


The Red Tape Area of a Segment

Each entry in the segment pointer list consists of a 63 bit pointer and a one bit 'read only' indicator (NOTE: a segment must begin on a 2 word boundary and the 'read only' bit is set to 0 when loaded into a segment register.)

The kernel provides special instructions (a) to load and store segment pointers in the segment pointer list, and (b) to access, store and use module capabilities in the module capability list.

9 Data Segment Registers



A Data Segment Register

Users can only address memory segments via segment registers. 16 such registers exist for addressing data.

SCID: short container identifier.

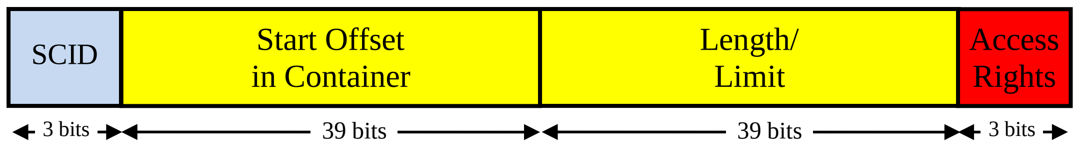
Start Offset in Container: Because segments begin on a word boundary, the 42 bit byte address defining the start address of the container can each be reduced to 39 bits. The hardware treats these as word addresses.

Segment Length: Similarly the length of a segment is also rounded up to a full word.

Access Rights: One bit is used to mark the segment register as *invalid*, one to mark the register as *read only* (or read-write) and one indicates whether the register values *can be stored* (see for example chapter 20 section 6.2.1)

10 Code Segment Register

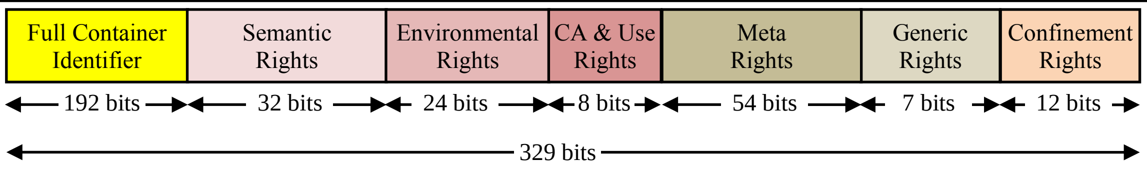
A single code segment register is used to address code segments.



The Code Segment Register

Access Rights: One bit is used to mark the segment register as *invalid*, one to mark the register as *execute only* or *execute and read* and one is spare. The latter might be used for example to indicate kernel use or main memory addresses.

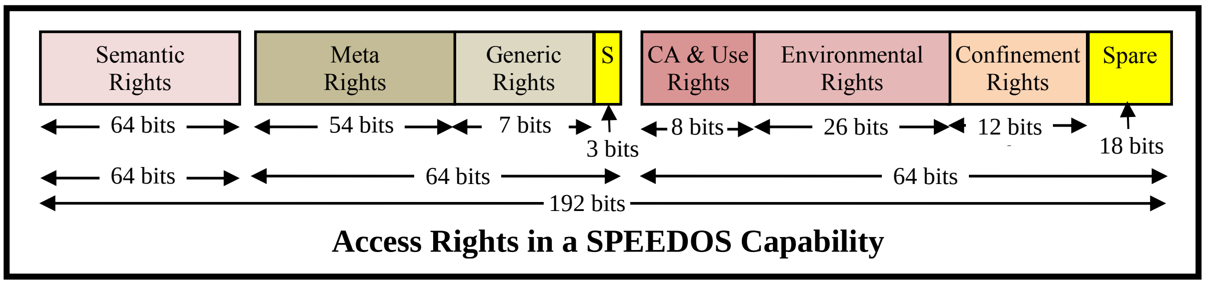
11 Capabilities



A SPEEDOS Capability

The full container identifier consists of the first three 64-bit words defined in sections 1 to 4 above.

The remaining fields are based on the discussion of security mechanisms in chapters 24 to 26. They can be implemented in three 64-bit words as follows:



Access Rights in a SPEEDOS Capability

Thus the Full Container Identifier field and the Access Rights field in a capability are each 3 words long and the entire capability is 6 words long. Note: It would be possible to move the index field and the type and status bits from the container number into the access rights of the capability, and still leave a few bits spare.

12 Semantic Rights

These rights indicate on an individual basis which entry points to a module can be called using the capability. The first sixty two bits define up to 62 semantic routines which can be called. In addition there are two bits which allow the list to be overridden by the following special bit settings:

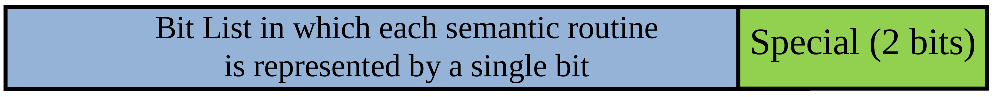
00 = none, i.e. no semantic routine can be called;

01 = all, i.e. all the semantic rights can be called;

10 = read only, i.e. only enquiries can be called;

11 = use the list of 62 semantic rights.

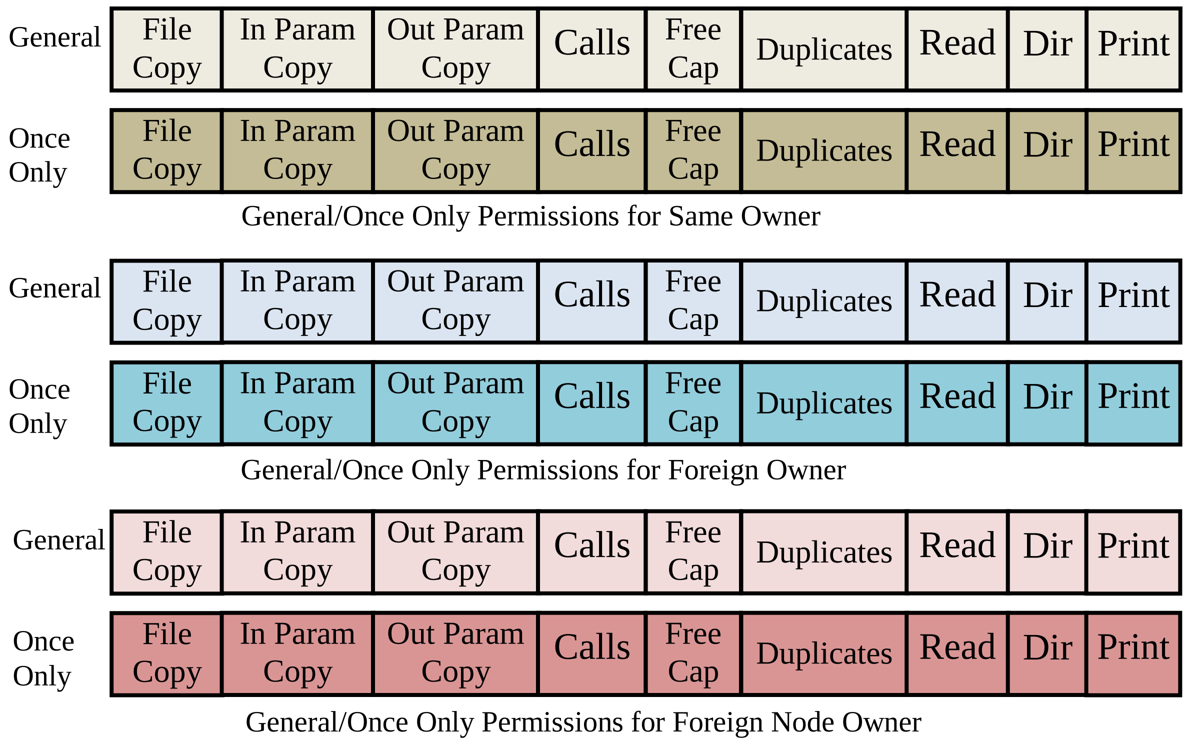
The first three of these are useful shortcuts for users. For modules which have less than 62 semantic rights the unused bits are set to 0.


Bit List in which each semantic routine is represented by a single bit
Special (2 bits)

Semantic Rights in Capabilities

Bracket routines are not considered to be semantic routines and can never be invoked directly. However, if an executing bracket routine presents a capability to call a module, the above rules apply as normal to the call which it is attempting to make.

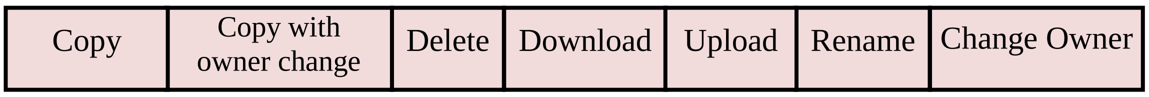
13 Metarights



Metarights in Capabilities

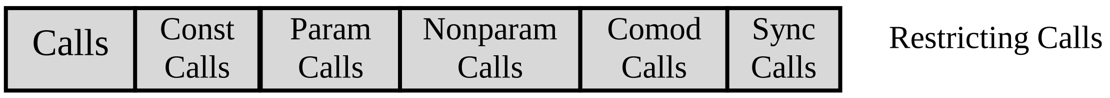
The metarights in capabilities are explained in chapter 26 section 3.3.

14 Generic Rights



Generic Access Rights in Capabilities

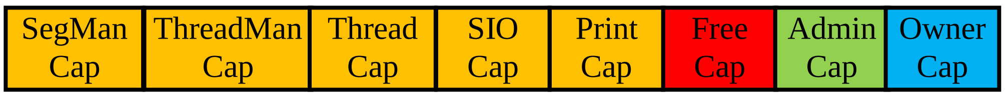
The generic rights are explained in chapter 26 section 3.2.



Module Call Confinement Rights

The principles underlying these confinement rights the subject of chapter 25.

15 Capability Accessibility and Use Rights



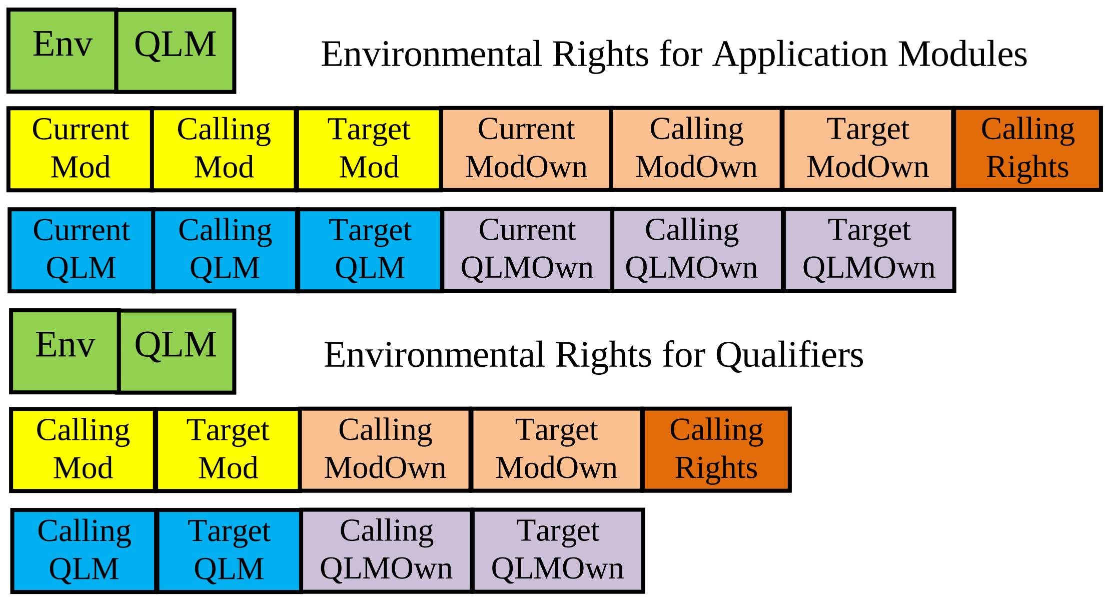
Capability Accessibility and Use Rights: An Overview

The capability accessibility and use rights are described in chapter 26 section 2. The capability accessibility rights define which capabilities can be obtained by a thread via kernel instructions (see chapter 19 section 5 and chapter 26 section 2.

It is not sufficient for a thread to execute capability accessibility instruction on the basis of the rights in a capability. The appropriate right must also be set in the Thread Security Register (see below) and a kernel capability must also be presented with this right set.

The rights conferred in the final three use bits can only be used if they are set in the capability.

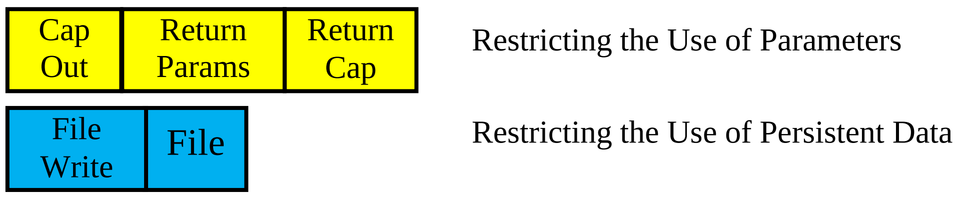
16 Environmental Rights



Environmental Rights

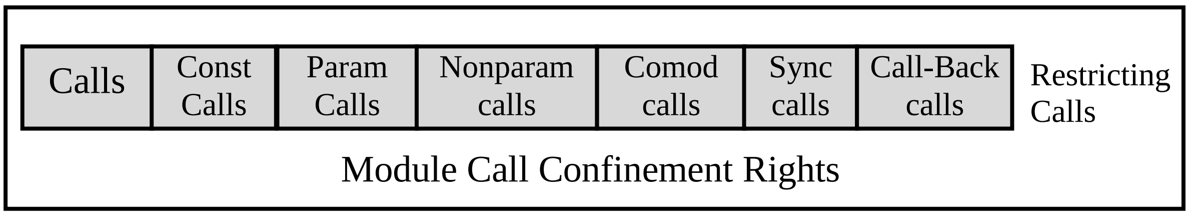
This is a bit list with 26 permissions. These indicate which of the kernel's environmental instructions can be executed from within the module by threads executing in it. As in the case of the capability accessibility rights a it is not sufficient for a thread to execute an environmental instruction (see the list in chapter 26 section 1) on the basis of the rights in a capability. The appropriate right must also be set in the Thread Security Register (see below) and a kernel capability must also be presented with this right set.

17 Confinement Rights


Cap Out
Return Params
Return Cap
Restricting the Use of Parameters
File Write
File
Restricting the Use of Persistent Data

Information Confinement Rights

The confinement rights are in two parts. The first of these is concerned with confining information which is available in a module to that module. The second is concerned with restricting the kinds of calls available to a thread while it is active in a module, based on the type of the origin segment of the capability being used to make the call.

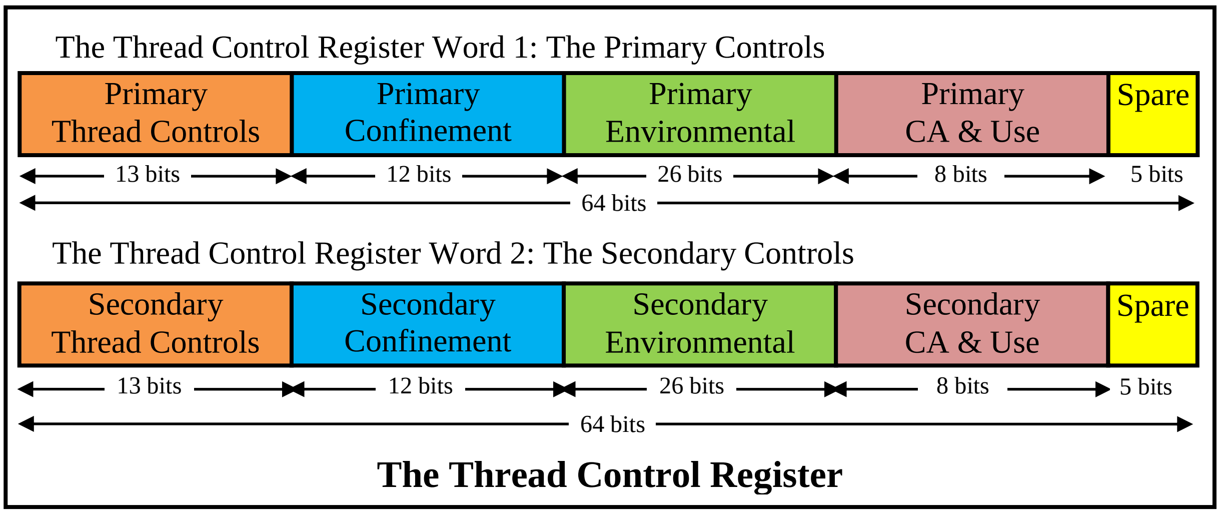


Module Call Confinement Rights

18 The Thread Security Register (TSR)

The TSR is a register (normally implemented in software at the base of a thread's stack) which is part of the state of an active thread and therefore, like the container registers, is copied into and retrieved from the stack linkage segments on an inter-module call and similar calls and on thread switches. It is only available for kernel access and is used to restrict the activities of a thread.

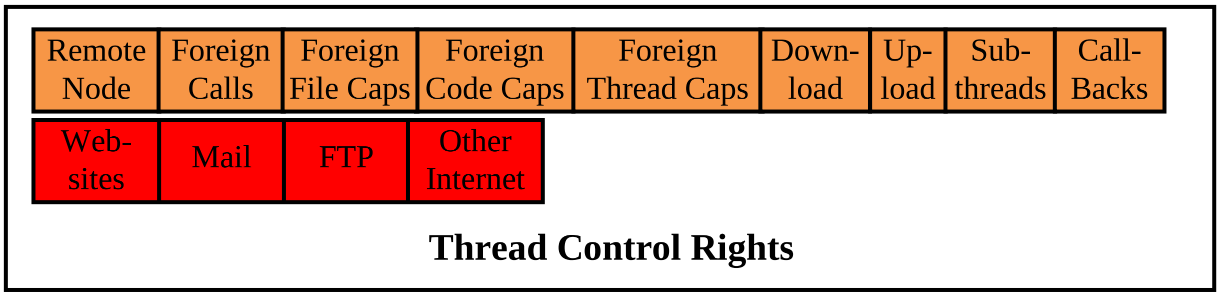
The TSR has four groups of rights, each of which has a primary section and a secondary section containing the same rights fields. The principles on which it works are described in chapter 26 section 4.



The Thread Control Register

The TSR is implemented in two 64 bit words.

19 The Thread Control Rights



Thread Control Rights

For an explanation of the individual Thread Control Rights see chapter 26 section 4. Note that those thread control rights coloured red are only relevant to systems which allow communication with and access to non-SPEEDOS nodes (see chapter 34 section 7.3.2).

20 The Confinement Rights

These are the same as those listed in section 17. There are twelve individual rights in each section.

21 The Environment Rights

These are the same as those listed in section 16. There are eleven individual rights in each section.

22 The Capability Accessibility and Use Rights

These are the same as those listed in section 15. There are eight individual rights in each section.

23 The Container Confinement Rights

The container confinement rights determine whether information in a container can be transferred to another node via a download or upload, whether they can be used by a thread the owner of which is not the owner of the container and whether they can be used by a thread belonging to another node which has been transferred temporarily to the current node following a remote inter-module call.

Foreign Download	Foreign Upload	Foreign Thread	Imported Thread

Container Confinement Rights

These rights are held in page 0 of each container, but neither in module capabilities nor the Thread Security Register.

References

[1] J. L. Keedy, “S-RISC: Adding Security to RISC Computers,” SPEEDOS Website (https://www.speedos-security.org/), 2023.

[2] B. W. Lampson, “Protection,” *ACM Operating Systems Review,* vol. 8, no. 1, pp. 18-24, January 1974.

[3] J. L. Keedy, M. Evered, A. Schmolitzky and G. Menger, “Attribute Types and Bracket Implementations,” in *25th International Conference on Technology of Object Oriented Systems, TOOLS 25,* Melbourne, 1997.

[4] K. Espenlaub, Design of the SPEEDOS Operating System Kernel, Ulm, Germany: Ph.D. thesis, The University of Ulm, Department of Computer Structures, Computer Science Faculty, 2005.

[5] J. Rosenberg, The Concept of a Hardware Kernel and its Implementation on a Minicomputer, Melbourne: PhD thesis, Monash University, Department of Computer Science, 1979.

[6] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack, “HYDRA: The Kernel of a Multiprocessor Operating System,” *Communications of the ACM,* vol. 17, no. 3, pp. 336-345, 1974.

[7] J. L. Keedy, TIMOR-An Object- and Component Oriented Language, 2020.

[8] J. L. Keedy, G. Menger and C. Heinlein, “Inheriting from a Common Abstract Ancestor in Timor,” *Journal of Object Technology,* vol. 1, no. 1, pp. 81-106, May-June 2002.

[9] A. K. Jones, “Capability Architecture Revisited,” *ACM Operating Systems Review,* vol. 14, no. 3, pp. 33-35, 1980.

[10] Cypress Semiconductor, SPARC RISC User's Guide, 2nd ed., 1990.

[11] B. Freisleben, Mechanismen zur Synchronisation paralleler Prozesse (1985), Darmstadt: PhD Thesis, Technical University of Darmstadt, Germany, 1985.

[12] B. Freisleben, Mechanismen zur Synchronisation paralleler Prozesse, Springer Informatik Fachberichte 133, 1987.

[13] R. Conradi, “Some Comments on Concurrent Readers and Writers,” *Acta Informatica,* vol. 8, pp. 335-340, 1977.

[14] R. H. Campbell and A. N. Habermann, “The Specification of Process Synchronisation by Path Expressions,” in *Operating Systems, an International Symposium,* Rocquencourt, 1974.

[15] D. P. Reed and R. K. Kanodia, “Synchronization with Eventcounts and Sequencers,” *Communications of the ACM,* vol. 22, no. 2, pp. 115-123, 1979.

[16] J. L. Keedy and B. Freisleben, “On the Efficient Use of Semaphore Primitives,” *Information Processing Letters,* vol. 21, no. 4, pp. 199-205,

1985.

[17] J. L. Keedy, J. Rosenberg and K. Ramamohanarao, “On Synchronising Readers and Writers with Semaphores,” *The Computer Journal,* vol. 25, no. 1, pp. 121-125, 1982.

[18] J. L. Keedy and B. Freisleben, “Priority Semaphores,” *The Computer Journal,* vol. 32, no. 1, pp. 24-28, 1989.

[19] J. L. Keedy, K. Ramamohanarao and J. Rosenberg, “On Implementing Semaphores with Sets,” *The Computer Journal,* vol. 22, no. 2, pp. 146-150, 1979.

[20] F. A. Henskens, A Capability-Based Persistent Distributed Shared Memory, Newcastle NSW, Australia: Ph.D. thesis, University of Newcastle NSW, Department of Computer Science, 1991.

[21] J. Rosenberg, J. L. Keedy and D. Abramson, “Addressing Mechanisms for Large Virtual Memories,” *The Computer Journal,* vol. 35, no. 4, pp. 369-375, 1992.

[22] D. A. Abramson, Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory, Melbourne: Ph.D. thesis, Monash University, Dept. of Computer Science, 1982.

[23] P. J. Denning, “The Working Set Model for Program Behaviour,” *Communications of the ACM,,* vol. 11, no. 5, pp. 323-333, 1968.

[24] B. Freisleben, P. Kammerer and J. L. Keedy, “Capabilities and Encryption: The Ultimate Defence Against Security Attacks?,” in *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence,* 1990.

[25] D. A. Abramson and J. L. Keedy, “Implementing a Large Virtual Memory in a Distributed Computing System,” in *Proceedings of the 18th Hawaii International Conference on System Sciences*, 1985.

[26] J. L. Keedy and J. V. Thomson, “Command Interpretation and Invocation in an Information Hiding System,” in *Proceedings of the IFIP TC-2 Conference on the Future of Command Languages: Foundations for Human-Computer Communication*, Rome, Italy, 1985.

[27] R. M. Needham, “Capabilities and Security,” in *Security and Persistence, International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, Germany, 1990.

[28] K. Lee, Shared virtual memory on loosely coupled multiprocessors, New Haven, CT 06520, USA): Ph.D. thesis, Yale University Department of Computer Science, 1986.

[29] B. W. Lampson, “Computer security in the real world,” *IEEE Computer,* vol. 37, no. 6, pp. 37-46, 2004.

Bibliography

Abramson, D. A. (1982). Computer Hardware to Support Capability Based Addressing in a Large Virtual Memory. Melbourne: Ph.D. thesis, Monash University, Dept. of Computer Science.

Abramson, D. A. & Keedy, J. L. (1985). Implementing a Large Virtual Memory in a Distributed Computing System. Proceedings of the 18th Hawaii International Conference on System Sciences, (pp. 515-522).

Campbell, R. H. & Habermann, A. N. (1974). The Specification of Process Synchronisation by Path Expressions. Operating Systems, an International Symposium (pp. 89-102). Rocquencourt: Springer.

Conradi, R. (1977). Some Comments on Concurrent Readers and Writers. Acta Informatica, 8, 335-340.

Cypress Semiconductor. (1990). SPARC RISC User's Guide (2nd ed.).

Denning, P. J. (1968). The Working Set Model for Program Behaviour. Communications of the ACM,, 11(5), 323-333.

Espenlaub, K. (2005). Design of the SPEEDOS Operating System Kernel. Ulm, Germany: Ph.D. thesis, The University of Ulm, Department of Computer Structures, Computer Science Faculty.

Freisleben, B. (1985). Mechanismen zur Synchronisation paralleler Prozesse (1985). Darmstadt: PhD Thesis, Technical University of Darmstadt, Germany.

Freisleben, B. (1987). Mechanismen zur Synchronisation paralleler Prozesse. Springer Informatik Fachberichte 133.

Freisleben, B., Kammerer, P. & Keedy, J. L. (1990). Capabilities and Encryption: The Ultimate Defence Against Security Attacks? Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence, (pp. 106-119).

Henskens, F. A. (1991). A Capability-Based Persistent Distributed Shared Memory. Newcastle NSW, Australia: Ph.D. thesis, University of Newcastle NSW, Department of Computer Science.

Jones, A. K. (1980). Capability Architecture Revisited. ACM Operating Systems Review, 14(3), 33-35.

Keedy, J. L. (2020). TIMOR-An Object- and Component Oriented Language.

Keedy, J. L. & Freisleben, B. (1985). On the Efficient Use of Semaphore Primitives. Information Processing Letters, 21(4), 199-205.

Keedy, J. L. & Freisleben, B. (1989). Priority Semaphores. The Computer Journal, 32(1), 24-28.

Keedy, J. L. & Thomson, J. V. (1985). Command Interpretation and Invocation in an Information Hiding System. Proceedings of the IFIP TC-2 Conference on the Future of Command Languages: Foundations for Human-Computer Communication. Rome, Italy.

Keedy, J. L., Evered, M., Schmolitzky, A. & Menger, G. (1997). Attribute Types and Bracket Implementations. In C. D. Mingins (Ed.), 25th International Conference on Technology of Object Oriented Systems, TOOLS 25, (pp. 325-337). Melbourne.

Keedy, J. L., Menger, G. & Heinlein, C. (2002, May-June). Inheriting from a Common Abstract Ancestor in Timor. Journal of Object Technology, 1(1), 81-106.

Keedy, J. L., Ramamohanarao, K. & Rosenberg, J. (1979). On Implementing Semaphores with Sets. The Computer Journal, 22(2), 146-150.

Keedy, J. L., Rosenberg, J. & Ramamohanarao, K. (1982). On Synchronising Readers and Writers with Semaphores. The Computer Journal, 25(1), 121-125.

Lampson, B. W. (1974, January). Protection. ACM Operating Systems Review, 8(1), 18-24.

Lampson, B. W. (2004). Computer security in the real world. IEEE Computer, 37(6), 37-46.

Lee, K. (1986). Shared virtual memory on loosely coupled multiprocessors. New Haven, CT 06520, USA): Ph.D. thesis, Yale University Department of Computer Science.

Needham, R. M. (1990). Capabilities and Security. In J. Rosenberg & J. Keedy (Ed.), Security and Persistence, International Workshop on Computer Architectures to Support Security and Persistence of Information (pp. 3-8). Bremen, Germany: Springer.

Reed, D. P. & Kanodia, R. K. (1979). Synchronization with Eventcounts and Sequencers. Communications of the ACM, 22(2), 115-123.

Rosenberg, J. (1979). The Concept of a Hardware Kernel and its Implementation on a Minicomputer. Melbourne: PhD thesis, Monash University, Department of Computer Science.

Rosenberg, J., Keedy, J. L. & Abramson, D. (1992). Addressing Mechanisms for Large Virtual Memories. The Computer Journal, 35(4), 369-375.

Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., et al. (1974). HYDRA: The Kernel of a Multiprocessor Operating System. Communications of the ACM, 17(3), 336-345.

Acknowledgements

I would like to thank all my PhD students who have either directly contributed to the design of SPEEDOS, or indirectly via the MONADS design and implementations. Key contributions were made by the following.

MONADS Design and Implementations

The original work on MONADS was carried out at Monash University and the University of Newcastle, NSW in Australia primarily in conjunction with my following former PhD students:

– John Rosenberg, who later became Professor at the University of Sydney, Dean of the Information Technology Faculty at Monash University, Deputy Vice-Chancellor at the Universities of Deakin and then Latrobe.

– Kotagiri Ramamohanarao, later Professor of Computer Science at the University of Melbourne; Head of Computer Science and Software Engineering, Head of the School of Electrical Engineering and Computer Science at the University of Melbourne and Research Director for the Cooperative Research Centre for Intelligent Decision Systems.

– David Abramson, later Professor and Head of Department at Monash University, then Director of Research at the Research Computer Centre of the University of Queensland (Co-supervisor Professor Chris Wallace).

– Frans Henskens, later Associate Professor at the University of Newcastle, NSW; Head of the Discipline of Computer Science and Software Engineering, Deputy Head of School of Electrical Engineering and Computer Science, Assistant Dean (IT) in the Faculty of Engineering and Built Environment and subsequently Professor in the Faculty of Health and Medicine at the University of Newcastle (Supervisor Prof. John Rosenberg).

Further Work on Operating Systems Design

During my period as Professor of Operating Systems at the University of Darmstadt in Germany some advanced synchronisation techniques were developed for MONADS by my PhD student

– Bernd Freisleben, later Professor of Distributed Systems at the University of Marburg in Germany.

At the University of Bremen in Germany the following contributed further ideas to the design of operating systems and database systems:

– Karin Vosseberg, later Professor of Software Technology at the University of Applied Sciences, Bremerhaven in Germany and Deputy Director for Study and Teaching.

– Peter Brössler, later a manager in various software companies and then a Freelance Management Adviser in Munich, Germany.

Engineering Support for the MONADS Systems

Special mention is due on the engineering side to David Koch at Monash University and the University of Newcastle, and to Jörg Siedenburg at the Universities of Bremen and Ulm.

Design of SPEEDOS

Many important contributions to SPEEDOS were made by my PhD student

– Klaus Espenlaub, now Software Development Director, Oracle VM VirtualBox, Oracle Corporation.

Design of Timor

In parallel with the SPEEDOS Project I led a programming language design project for an object-oriented language called Timor (Types, Implementations and More) at the University of Ulm. Since an important aim of this project was to provide a programming language suitable for implementing SPEEDOS, there was much interaction between the SPEEDOS project and the members of the Timor team, to whom my thanks are also due for their indirect and direct contributions to the SPEEDOS ideas.

The main members of the Timor team were

– Mark Evered, later Senior Lecturer at the University of New England, NSW Australia and Researcher at the Department of Primary Industries NSW.

– Axel Schmolitzky, later Professor at the University of Applied Sciences, Hamburg, Germany.

– Gisela Menger, now retired.

– Christian Heinlein, later Professor at the University of Applied Sciences, Aalen, Germany and Dean of Studies.

I would also like to thank all the students who worked on MONADS, SPEEDOS and/or Timor.

My work has been supported over the years by several competent secretaries, and my special thanks in this respect are due to Renate Post-Gonzales for organising the 'International Workshop on Computer Architectures to Support Security and Persistence' in Bremen in 1990.

Thanks are also due to the Australian Research Grants Committee for their financial support of the MONADS Project at Monash and Newcastle.

Above all I am enormously grateful for the love, patience and support which I have received from my wife Ulla and also from my son Nicolas.