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 1

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 1
Introductory Concepts

Chapter 1
Computer Security: an Ongoing Problem

I first considered writing an introductory chapter called "Is there a Computer Security Problem?" but after a little consideration I decided that computer security problems are so widespread and so well known that such a chapter would only bore readers. At the time of writing, so many computer security problems have been so widely reported that even most non-specialists are aware that there are serious problems.

Significant examples of this include infamous viruses and worms such as Stuxnet, Duqu and Flame, hacker break-ins at Sony, Citigroup, Google, the International Monetary Fund, an Iranian atomic energy plant, Paypal, Sega, Nintendo, the US broadcaster PBS, the Australian National Broadband Network, the Hong Kong Stock Exchange and even the CIA, the Pentagon[3], the US Senate and the NATO HQ, not to mention the so-called "Shady RAT" hackings discovered by McAfee, involving break-ins over 5 years at 72 corporate companies, government computers, and private and public organisation in 14 countries, including the International Olympic Committee.

Such problems are not new. They have been happening over many years. Amongst the most spectacular and well known cases from the 1980s is that described by Clifford Stoll [2]. Realising that a hacker was regularly breaking into his computer system at the Livermore Laboratories in the US and was using this as a base to break into other supposedly highly secure sites (including US Department of Defense sites), Stoll spent many months patiently tracking the hack-

[3] It was reported on the Australian ABC News 24 channel that there are over 6 million attempts to break into Pentagon computers every day.

er down to Germany. A story well worth reading!

The favourite target – and the favourite tool – of white collar criminals is the computer. Hackers use computers on a routine basis to break into other computers. Computer crime might seem a bit remote from most of our lives. But it is important, because it can involve much higher spoils than say a normal bank robbery, and it is far more difficult to detect.

In the modern world computers are ubiquitous. They are capable of performing astounding feats of calculation. This alone brings us tremendous benefits. For example without large fast supercomputers the improvements in weather forecasting which we now enjoy compared to a few years ago would be quite impossible. The security problem does not exist primarily because computers can calculate. It stems much more from the fact that they also have prodigious memories, which are used to store enormous amounts of information about almost every aspect of our lives, our finances, our businesses, and so on. Our employers, our lawyers, our doctors, our dentists, our hospitals, our banks, our insurance companies, our clubs and our governments all store lots of information about us. Commercial firms have records of trillions of financial transactions in their computer databases.

Unauthorised access to such information can lead to serious violations of data privacy. It is perhaps worth noting at this early point that data privacy and computer security are *not* synonymous terms. Privacy is not the same as security. Privacy is a legal issue involving special laws to protect the citizen from misuse of his information stored in computer databases and other systems.

This book is not directly about privacy. It is about security, which is an important precondition for achieving privacy. All the privacy laws in the world will not solve the privacy problem unless they can be backed up by secure computers: computers which don't give up their secrets about us to unauthorised parties. So if you are interested in data privacy then you should also be interested in computer security.

Computers also contain lots of information which has nothing to do with personal privacy but which is nevertheless very sensitive. For example they store information which affects the values of stocks and shares on the financial markets, secret marketing or design information about new company products, and intelligence information gathered by security agencies. Such information can be very valuable to competitors, foreign governments and the like.

So there are plenty of reasons why people might want to break into computer systems to acquire information. What can sometimes be even worse: a huge amount of damage can be done by people changing or destroying information and/or programs stored in computers. Thieves can steal from banks by modifying banking records. The wrong operation could be carried out on a hospital patient if his computerised records are changed. A possible consequence of a company's database being completely wiped out is that the company will go bankrupt. And terrorists could blow up nuclear power stations if they could tamper with the computer programs controlling them.

This is not an alarmist book. I just want to make the point that computer security is important and is growing in significance as we store more and more information in computers. We already find ourselves in the information age, the age in which humanity's most valuable resources are information and knowledge. Industrial spies, terrorists, white collar criminals and many others have much to gain by breaking into the computer systems where this information is stored. And sovereign states must also prepare themselves for the fact that cyber warfare is already taking place, and is likely to be the most dominant form of warfare in the 21st century.[4]

In 1985 the U.S. Department of Defense (DoD) set a trend by publishing its now famous "Orange Book". The real title of this report is *Trusted Computer System Evaluation Criteria*, but it acquired its nickname from the colour of its cover. The security criteria defined in the Orange Book had three major aims, to guide manufacturers regarding the security measures to build into their future computer products, to provide users with a yardstick for assessing how much trust can be placed in a computer system, and to serve as a basis for security specifications in future DoD acquisition specifications. It was envisaged in the Orange Book that evaluations of actual systems can be performed, either in terms of specific application environments or in terms of general systems, and that security certification and accreditations can be approved where appropriate by Designated Approving Authorities. The mechanisms of such accreditations and approvals are of less interest to us than the recognition that breaches of security fall into three broad categories [3].

— Breaches of *confidentiality* result in a flow of information to unauthorised persons.

— Breaches of *integrity* result in information being incorrectly recorded in the system.

— Breaches of *availability* lead to loss of use of the system or some of its resources.

This book is about making computers secure, about making information safe so

[4] see https://en.wikipedia.org/wiki/Cyberwarfare. For a recent discussion of cyber warfare, see "Ten cyber-warfare threats (and how to fight back)" https://bcshq.org/9u7-6b3lw-6i93ke-3mbfu0-1/c.aspx, and Prof. Claudio Cilli "Cyber-warfare and the New Threats to Security" https://nlondon.bcs.org/pres/ccmay19.pdf

that it is inaccessible to those not authorised to read it, or change it, or destroy it. And it is about protecting programs, whether they are proprietary programs which need protection from piracy, or sensitive real-time programs controlling machines and systems such as airplanes, nuclear power stations or hospital equipment, which need protection from criminals and terrorists.

The seriousness of the consequences of unauthorised access to computer systems is now recognised by legislators and in many countries there are severe penalties for breaking into computing systems[5]. But laws alone will not prevent such violations, just as laws alone do not prevent the thief from stealing your television set. Even the police are not all that successful at stopping burglaries. We all know that in practice the best strategy is prevention, so we put locks and bolts on our doors and we buy burglar alarms and install security cameras to warn us that a thief might be active.

In fact one way to keep computers secure is to lock them away physically, behind bolted doors in impregnable vaults. While such measures may help a little, they do not solve the fundamental problem for most users. Computers are most useful when they are networked with other computers, or when they are provided with new programs which are introduced to computer systems by downloading from the internet or by using CDs or the like. Herein lurk a myriad of dangers for computer security: the computer equivalents of Trojan horses, viruses, worms and bugs! Suffice to say at this stage, securing computers physically is not necessarily going to make your system immune to the world of electronic insects and other dangers. It is essential to provide internal mechanisms within the computer which prevent such dangers from being effective, and which safeguard the security of information and programs.

This book is primarily about describing a set of computer mechanisms equivalent to locks and keys, bolts and burglar alarms. You might interject that modern computers are already fitted out with some such mechanisms, which on the whole are not very effective against the professional or determined hacker. And of course you would be right. But the mechanisms which I describe in later chapters are quite different in their nature from those with which you are familiar or which are employed in conventional computer systems. They include a new way of designing computers and a radically different approach to designing operating system and application software. But before we look at the new mechanisms we must consider why the current mechanisms are so ineffective.

There are in fact lots of reasons. One is that since the early 1980s very little fundamental research has been carried out into making computer protection

5 Ironically, most governments do not see a problem in allowing their own agencies to break into the computers of their own citizens!

mechanisms effective. Of course, many specialists have devoted many hours to producing more secure mechanisms, and I do not want to belittle their work in any way. But this is not what I mean by fundamental research. Basically their work consists of inventing (often very clever) techniques and programs which are best viewed in my opinion as techniques for patching up an inadequate core. Their work starts from the assumption that the basics around which computer systems are built (e.g. the design of computer processors and memories, the design of operating systems, database systems and programming languages) are fundamentally in order. The implicit view is that these aspects of computers perhaps leave some room for improvement but they cannot be fundamentally changed. This book will show that this view is far from correct.

Another, not unrelated, reason for the ineffectiveness of current systems in the face of security attacks can be explained by the *panda principle*, first formulated by the evolutionary biologist Stephen Jay Gould [4, 5]. This in effect says that once an inadequate mechanism or concept has firmly established itself successfully, it is then extremely difficult to replace it with a newer and more effective principle. Gould illustrates this principle with the example of the panda's thumb (which is not really a thumb in the normal sense of the word), but also with the example of the QWERTY keyboard [6, pp. 322-324].

A few newer operating systems have appeared in recent years, but the principles upon which they are based are not dramatically different from those with which they are competing. At the level of hardware design, the current principles upon which processors are built are concerned with improving processing speed, but very little fundamental research on how processor design can contribute to improvements in security has been carried out since the 1980s. Dislodging the firmly established but insecure operating systems which are widely used today is an enormously difficult task, and it is even more difficult to dislodge the current direction of processor design.

However, exactly that is what is needed if we really want secure computer systems. The current approach to achieving security is to wait until loopholes appear (and lots of them do), and then attempt to patch these loopholes up.

But this re-active approach can never be fully successful, because the principles on which current operating systems are based are themselves fundamentally insecure, as we will attempt to show in later chapters. Instead we need to rethink the design of operating systems and computer processors in such a way that these inadequate basic principles are replaced by totally new, fundamentally more secure, principles. In other words, we need to replace the current re-active approach with a new pro-active approach. Explaining in broad outline how a new approach might look is the main purpose of this book.

Before we embark on this task it will help if we clarify a few basic issues.

1 Complexity and Simplicity

The security mechanisms which have been built into or added onto current computer systems are many and various. Some of them are directly built into the computer hardware, designed for example to stop one program from writing into or reading from another program's memory space or to ensure that only the right programs are allowed to execute certain sensitive instructions.

Some security mechanisms are typically built into the operating system. At this level of the system, for example, the passwords of users are usually checked, and the hardware protection mechanisms are applied to programs and data. A large part of most operating systems is the file system, which is responsible for controlling access to the information (usually in the form of data files and program files) which users store on discs and similar storage devices.

In many large computer systems the information in files is organised by a database management system, which usually adds its pepper to the protection frying pan in the form of some extra protection controls.

To that we often find some more security mechanisms added by the software responsible for networking computers together.

Then on top of that it is possible to buy proprietary software packages which have been especially designed to improve the security of the system by patching up weaknesses in the other mechanisms.

The end result is that we have lots and lots of security mechanisms but, if the news reports of computer crime are anything to judge by, very little security. A former colleague of mine once likened this situation to the Berlin Wall, which was also a security device – for keeping East German citizens in East Berlin. Although it prevented a lot of people from reaching West Berlin, the Wall was not a very successful security mechanism, because about 5000 people succeeded in escaping. Compare this, my colleague said, with the almost perfect record of the notorious Alcatraz prison in the United States.

Why was Alcatraz much more successful than the Berlin Wall?

In fact the Berlin Wall was not a continuous wall, but a whole variety of mechanisms, such as a stretch of river, walls of houses which had been evacuated, barbed wire sections with armed soldiers in watch towers, booby traps which exploded if you stood on them, and so on. The escapees could often exploit this very multiplicity of mechanisms, finding escape niches between the mechanisms, because the way that security worked was not coordinated enough and simple enough to be effective.

On the other hand, while there were of course some obstacles within Alcatraz to prevent escapes, there was one very simple to understand and very effective mechanism: the extremely cold waters and hazardous currents of San Francisco Bay!

This comparison makes a very important point about security mechanisms. The more complicated and the more cumbersome you make the mechanisms, the more likely it is that people will find weak points, by playing the mechanisms off against each other, or by slipping between them undetected.

This is exactly the situation we find today when we examine computer security. Every day hackers manage with ease to slip through the cumbersome technical mechanisms which we currently use in computer hardware, operating systems, file systems, database systems, networking systems, special software packages, etc. What is lacking is a simple but effective concept, comparable with the freezing waters around Alcatraz!

The security mechanisms found in present day computer systems are certainly not simple and can hardly be considered effective. One of the main reasons is that current processor hardware has not been designed with such a concept in mind, and just as important, the software which controls the systems is incredibly complex. It has become a common cry of despair amongst computer programming experts that the software systems have become incredibly complex, so much so that no single person is in a position fully to understand how a modern operating system works. It is hardly surprising in this situation that computer security mechanisms are not very effective.

A quotation from John Dewey, the renowned U.S. philosopher, psychologist and educationalist, helps to explain how this situation has arisen:

> "But the easy and the simple are not identical. To discover what is really simple and to act upon the discovery is an exceedingly difficult task. After the artificial and the complex is once institutionally established and ingrained in custom and routine, it is easier to walk in the paths that have been beaten than it is, after taking a new point of view, to work out what is practically involved in the new point of view. The old Ptolemaic astronomical system was more complicated with its cycles and epicycles than the Copernican system. But until organisation of actual astronomical phenomena on the ground of the latter principle had been effected, the easiest course was to follow the line of least resistance provided by the old intellectual habit." [7, p. 30]

One of the themes throughout the later chapters of this book will be to expose the complexity of current systems and to suggest ways of replacing it with simpler, more efficient and more effective alternatives.

It may come as a surprise to some to discover that there is almost no mathematics in this book. Many people, including many computer science academics, tend to believe that computer science is a branch of mathematics. Indeed there are many computer scientists who think that a book about computer science (and a computer science Ph.D. thesis) which is not liberally sprinkled with mathematics is worthless. I hope that this book will show the invalidity of such an extreme view. Mathematics can and should be used in computer science as a useful analytical tool, and it offers valuable theoretical insights into the nature and limits of computing, as well as providing effective encryption techniques.

But mathematics has few insights to offer when it comes to the task of discovering constructive simplicity in computer science. We cannot simply develop some equations which will find a simple, elegant and effective security mechanism for us. But finding such security mechanisms is what this book is about. After we have discovered a good security mechanism we might apply mathematical techniques to quantify how efficient it is and so on, but these techniques won't find the mechanism for us.

In this respect computer science is much more like engineering than mathematics or natural science. Natural systems are already present, all around us. They do not have to be designed by humans. It is appropriate and very helpful to analyse them mathematically, and it is challenging to go deeper and deeper into the detail. When we are on the right track this reductionist approach can offer many benefits, as the chemical industry, for example, has more than proved.

But mathematics can sometimes appear to be just as convincing when we are on the wrong track. The ever more intricate Ptolemaic cycles and epicycles provided 15th and 16th century mathematicians with a field day, until it was finally realised that the Earth is not the centre of the Universe!

Like most engineers, most practical computer scientists are primarily constructors and creators of useful artefacts. Our main job is not primarily to reduce things to their lowest levels and analyse them in ever greater detail, but to construct new and effective systems. As we know from other engineering endeavours, simplicity and elegance of design often go hand in hand to produce an effective system. If a system starts to get too complex, then this is a warning sign of a bad design.

Engineers often have an important advantage over computer scientists, however. They mostly build physical objects, and these can often be judged not only by their theoretical qualities but also by their physical appearance. If an engineer were to build a really clumsy bridge, or airplane, or ship, you would often be able to see by looking at it that it is so clumsy that it will fall down, or won't fly, or will sink or whatever. Of course appearances alone are not important, but they help us to remove a whole area of design "space" which is obviously inadequate.

Computer scientists don't have this advantage. You cannot just look at a complex computer program and see at one glance whether it is likely to be safe or correct or reliable. The result is that we in fact often build quite grotesque and incredibly complicated computer programs. Such constructions are inevitably riddled with errors, and this has provided those with mathematical tendencies with a new field day comparable with that of the pre-Copernican astronomers. They have invented a discipline called software engineering. This discipline quite rightly emphasises that good software products should be built on sound engineering and mathematical principles – a most laudable aim. But on the whole this approach has led to an emphasis on the mathematical and analytic aspects while ignoring the creative aspects of good engineering design.

A good example of this in relation to our present theme is the tendency to confuse a secure computer system with a correct computer system, where correctness means viewing a program in the same way a mathematician views a theorem, and then proving that it is correct.

I do not wish to belittle this approach – correct programs of course play a very important role in security. But in order to be able to prove a program correct, we must have a definition of what the program should do, a specification of the program against which we can measure its correctness. Such a specification must be a formal mathematically rigorous specification, if it is to be amenable to the mathematical approach. Unfortunately developing such specifications is a notoriously difficult problem. Generally speaking, it is almost impossible to specify anything but toy programs formally. If we do manage to specify a real one, the specification is usually almost impossible to understand, which amongst other things means that we cannot be sure that it is really specifying what we want the program to do! So we may prove a program "correct" only to find that it doesn't do what we really wanted in the first place.

The reason that very little mathematics appears in this book is because we are concerned with finding simple, elegant and efficient security mechanisms. The emphasis is on getting the overall picture right in the first place. Let us first find the forest which suits our purposes before we start measuring the heights of the individual trees and counting the number of leaves which they have.

2 The Role of Computer Architecture

In the 1970s a particular approach to computer security created a lot of interest in the research community. The idea was to base the design of computer systems on a concept called *capabilities*, which can be thought of as a kind of equivalent within the computer to locks and keys in the physical world. While some of the research was based purely on a software implementation of capabilities, other researchers integrated it into experimental computer architectures[6]. This was generally combined with ideas which eventually led to the idea of object-oriented programming. Unfortunately the architectural aspects of this research direction were prematurely killed off, because capability based computers were associated with the kind of computer design which came to be known as CISC (complex instruction set computers). In the early 1980s an alternative approach to the design of computer processors emerged, known as RISC (reduced instruction set computers). The RISC idea led to fundamental improvements in the efficiency of computers, and is one of the main factors behind the fact that microprocessor performance since the 1980s has been able to improve at an astonishing rate.

There can be no question of going back to the old-style of CISC architectures, but this does not necessarily mean that we have to abandon security in computer systems at the architectural level. Security is a theme which is at least as important as performance. As the originators of the RISC movement themselves wrote in the 5th edition of their standard textbook on computer architecture:

> "Security and privacy are two of the most vexing challenges for information technology in 2011. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Hence, both researchers and practitioners are looking for new ways to make computing systems more secure. Although protecting information is not limited to hardware, in our view real security and privacy will likely involve innovation in computer architecture as well as in system software." [8, p. 105]

I share this conviction that a simple well-chosen architectural extension to the RISC philosophy can harness the high performance with vastly improved security. In chapter 16 and in the Appendix we will in fact describe a very simple architectural extension to the RISC idea which is capable of fulfilling these expectations.

This conviction is primarily responsible for the ideas presented in the rest of this book. For this reason readers should not expect to find a fully balanced treatment of all aspects of computer security. Instead they will hopefully find a reasoned argument for reconsidering the idea that computer architectures, along with appropriate operating system and programming language ideas, have a major role to play in achieving secure systems in future.

In the following chapters we introduce some fundamental protection and security concepts. Then in later chapters we discuss the basic concepts of com-

[6] "Computer architecture", often abbreviated in this book simply to "architecture", refers to the computer science discipline concerned with the relationship between computer hardware and the programs (software) which execute on it, e.g. the design of a computer's instruction set, its basic memory protection mechanisms, etc.

puter architecture which are relevant to the issue of security. In these chapters we review a number of different memory management and addressing models, because the addressing of computer memory is a key factor in achieving security for information contained in the computer memory. We also describe a number of different models and techniques which are particularly relevant to the issue of security. Thereafter we describe the essential ingredients of the RISC philosophy and show that they are not inconsistent with the aims of highly secure computing environments.

In later parts of the book we develop some operating system principles and show how the SPEEDOS system (a combination of hardware and software) could put them into practice.

Chapter 2
Basic Security Concepts

This chapter describes a basic security model, which is concerned primarily with controlling the right to access information. This model leads us to a discussion of some of fundamental security issues.

1 Lampson's Matrix

In 1971 Butler Lampson described a simple but important model for expressing access controls [9]. According to this model a computer system can be viewed as a collection of subjects, a collection of objects and the access rights which subjects possess for objects. The model is expressed as a matrix, with each column representing a subject and each row representing an object, as is shown in Figure 2.1. An entry in the matrix defines the access rights which the subject in the appropriate column can exercise over the object in the corresponding row.

		Subjects			
		Subject 1	**Subject 2**	**Subject 3**	**Subject 4**
Objects	**Object 1**	access rights	access rights	access rights	access rights
	Object 1	access rights	access rights	access rights	access rights
	Object 1	access rights	access rights	access rights	access rights
	Object 1	access rights	access rights	access rights	access rights

Figure 2.1: Lampson's Access Matrix

For example, let us suppose that the subjects are users called Jill, Jack, Joan and John, and the objects are files called My File, Your File, Her File and His File. The access rights determine whether the users can read from and/or write to the files. An actual access matrix might then look like that shown in Figure 2.2.

		Subjects			
		JILL	JACK	JOAN	JOHN
Objects	My File	read, write	read	read	—
	Your File	—	read, write	read, write	—
	Her File	read, write	—	—	—
	His File	—	read, write	write	read

Figure 2.2: An Example of Lampson's Access Matrix

This defines for example that Jill has read and write access rights for My File and for Her File, but she has no access rights for Your File or for His File. Joan on the other hand is permitted to read from My File, to read from and write to Your File, but she can only write to His File and she has no access to Her File.

Now that the basic idea behind Lampson's matrix is clear, we can look closer at some of the concepts behind the model.

1.1 Subjects and Objects

Subjects are the active components of the system. They carry out the operations for which access rights (in this example read and write operations) are needed. Subjects can be – and often are – users of the system, for example people communicating with the system via a computer keyboard and monitor. But subjects needn't be human agents. The matrix model can be applied not only at the human level to subjects and objects, but to situations inside the computer itself. Thus it is possible to think of a program as a subject, if for example it accesses a file. In that case it appears in the matrix along with the other subjects. On the other hand we can also treat a program as an object in the matrix, since users can have access rights which allow them to execute programs. Figure 2.3 shows how an Editor program can be treated as an object which Jack and Joan (but not Jill) may execute. At the same time the matrix shows that the Editor may read and write Your File and may read (but not write) Her File.

Just as a subject need not be a human user, an object need not be a file or a program. It can be anything in the computer system that can be operated on and over which the right of access has to be controlled. For example it might be a segment of the memory or input-output equipment such as a printer.

		Subjects			
		JILL	**JACK**	**JOAN**	**Editor**
Objects	**My File**	read, write	read	read	—
	Your File	—	read, write	read, write	read, write
	Her File	read, write	—	—	read
	His File	—	execute	execute	—

Figure 2.3: An Example of Lampson's Access Matrix with an Object which is also a Subject

2 Unique Names

You might think that it goes without saying that the names of subjects and of objects should be unambiguous. If they were ambiguous, we couldn't be sure what subjects and/or what objects were actually intended in the matrix. For example if there were two users called Jack or two files called My File we would be in trouble, as it wouldn't be clear which subjects have the right to access which objects. We obviously have to take care to ensure that names can be uniquely associated with a single subject or a single object.

But in practice ambiguous names can easily slip into a system if we don't take care. Suppose for example that two users of a system each decide to call one of their files "Temp" (a favourite name for a temporary file)! Clearly there must be a way of distinguishing between them. And it would clearly be unacceptable to expect all the users in a system to check all the names of all the files they and others have created before naming a new file.

The problem of ambiguous names is not just a problem in computer systems. In the real world public authorities and private companies have to overcome the problem of non-unique names. They usually use one of two methods: hierarchical naming or timestamping.

2.1 Hierarchical Names

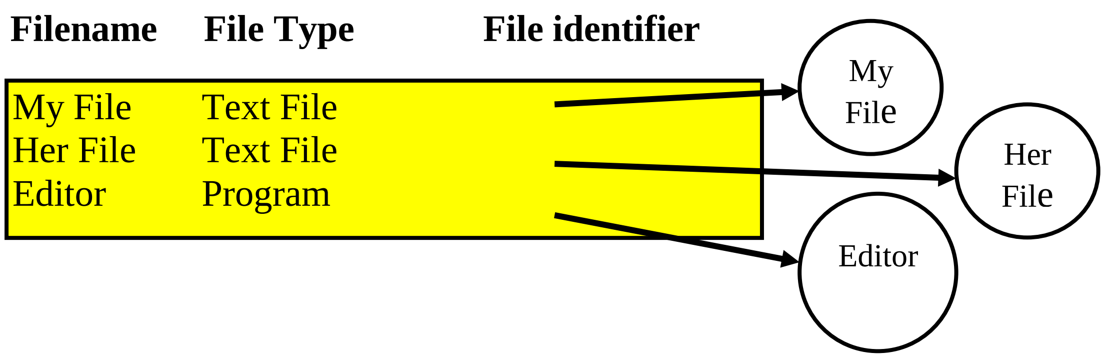
To understand hierarchical naming schemes you can think of what would happen if you tried to use just your local telephone number to ring your spouse from another city. You would probably end up being connected to a stranger. This is because the telephone authorities use the same set of local numbers independently in different cities. But you can call up your spouse from another city by prefixing the local number with an extra dialling code for the city. Even this is not enough if you are trying to ring from a different country, because the same city codes can be used in different countries. So then we prefix another extra code for the country.

This process uses a hierarchical structure in order to produce unique telephone numbers worldwide. It has two important advantages. First, the local numbers can be allocated in each city without having to check whether the same local number is being used in any other city. Similarly, each country can allocate city dialling codes without reference to those used in other countries. The second advantage is that in a local context telephone numbers are short.

A similar technique is often used to make names in computer systems unique while keeping them short in a local context. A good example of this is frequently found in file systems. A file system is that part of an operating system which manages files. Files are logically related collections of data (e.g. a payroll file, a bank accounts file, a letter) or programs which are stored over longer periods in the computer's secondary memory (e.g. its disks and tapes).

Hierarchically structured file systems make it possible for different users to allocate names for their files without worrying about the names used by other users. If a user called Smith creates a file called Temp, then the system knows that this file is really called something like Smith/Temp, and is able to distinguish it from another file called Temp created by Jones, because to the system this is called Jones/Temp. Here the *context* plays an important role.

In fact file systems usually allow users to introduce their own hierarchies of names, enabling each user to work in several contexts. This is done by introducing file *directories* (folders). A directory contains a list of files with differing local names, together with information describing each file (e.g. the file type) and how it can be located (e.g. a disc address), as is shown in Fig. 2.4.



The pointers represent file identifiers, which might e.g. be addresses on disc or pointers to further information enabling the file to be physically located (e.g. on a disc).

Figure 2.4: A Simple File Directory

If a directory is itself regarded as a file, information about it can be held in another directory, with the file information noting that it is a directory. This means that directories can point to other directories, which can point to further directories, and so on. The names of the directories can then be used in a hierarchical fashion, rather like a telephone number with dialling codes, to provide files with unique names. This is illustrated in Figure 2.5.

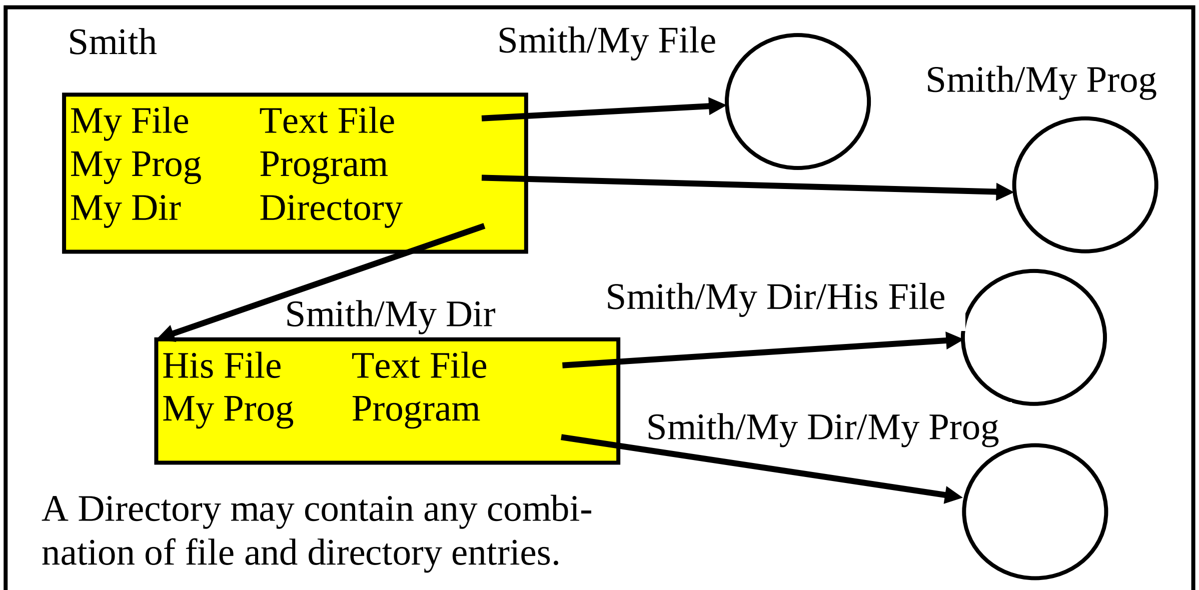


Figure 2.5: A Hierarchical Directory Structure

In this way a user can create different contexts in which he can work. For example if Jones is a professor who has a lot of files on the computer, he might want to create different contexts called Admin, Teaching and Research. If he wants to he can then have a file called Temp in each context. The full names of these files would be Jones/Admin/Temp, Jones/Teaching/Temp and Jones/ Research/Temp. As long as he is working in the context Teaching (and the system knows this) it suffices for him to call his file Temp; the system can add the context. (This is like being able to call your own telephone number in your own city.)

One point about the telephone numbering system needs to be mentioned: it is not time independent. Telephone authorities usually reallocate telephone numbers after people cancel their connections. This can then lead to the situation where you ring somebody you haven't contacted for a while and to your surprise a total stranger – who has been allocated the number which your friend previously had – answers your call.

In order to have time-independent telephone numbers the telephone company would have to use much larger local numbers. Most people prefer shorter numbers and are prepared to accept the risk that they occasionally get a wrong number.

Similarly in file systems users may want the freedom to reuse names in the same directory. This need not be forbidden, and in practice file systems do not prevent it. In fact it can be quite useful for example if a user gets an updated version of a file or program which he wishes to use in place of the old version. If there is a new issue of a word-processing program from which certain errors have been removed, the normal user will probably want the system to delete the old version and replace it with the new version, using the same symbolic name in his directory.

At this point it is useful to draw a distinction between external and internal names which are used in computer systems. It is a common practice in computer software design to use integers (whole numbers, such as the number 23) instead of symbolic names (consisting of alphabetic and other characters, such as My File) in the internal parts of the system. This is advantageous because numbers are usually shorter, they have a fixed length when stored in the computer's memory and they can often be used as indices into tables.

Whereas it can be useful to allow symbolic names, such as Temp, to be reused in a local context, we run into greater difficulties if internal names are reused over time. To illustrate this we can consider the pointers in Figures 2.4 and 2.5 as examples of internal names. A real problem arises if such a pointer is ambiguous over time. Suppose for example that there are two directories (for two users) each with an entry for the same file (Figure 2.6).

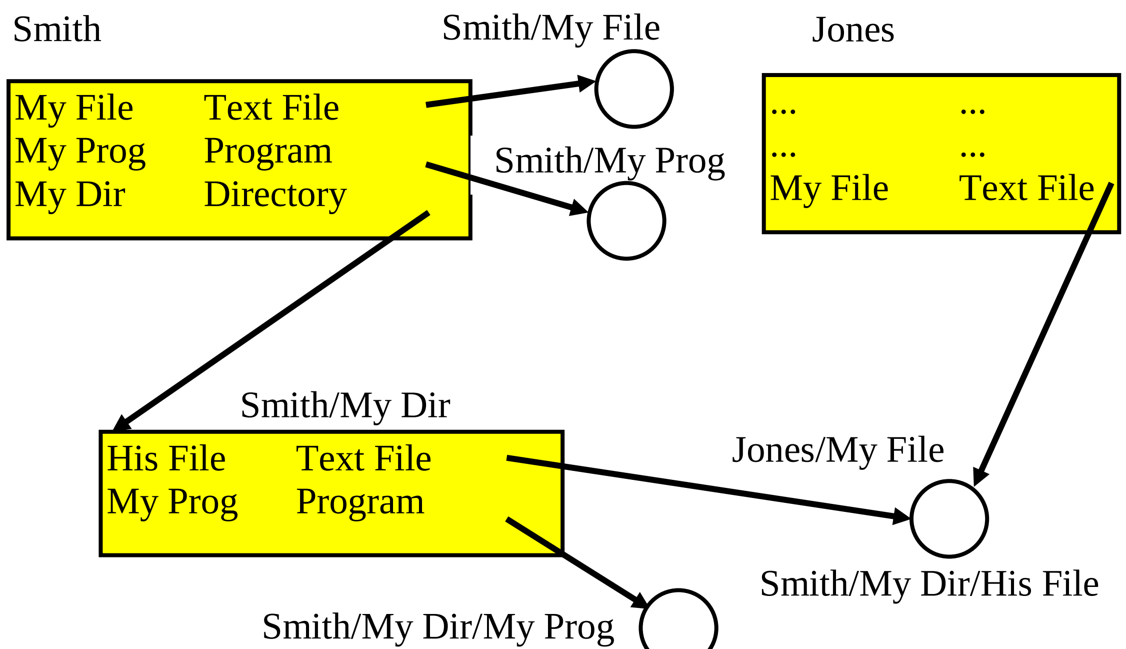


Figure 2.6: Two Directory Structures containing Different Pointers for the Same File

If one user deletes the file and the disk space gets used to create another file, the other user could end up accessing the wrong file. The fundamental problem is that a disk address is not a unique identifier for a file *over time*. At different times different files can be placed at the same address. (Similar considerations apply to using main memory addresses as identifiers.)

This is an example of how getting a "wrong number" may mean that a breach of security can occur. For security to work correctly there must be a level in the system, which cannot be by-passed, where subjects and objects are uniquely identified over time. Usually this will be at the level of internal names, because at the level of symbolic names it is useful for users to be able to rename objects and even have several names for the same object.

2.2 Timestamps

Names of people are not guaranteed to be unique – think of names like Peter Smith or John Jones. It is quite usual for authorities to make such names into unique identifiers by using the name and the place and date of birth together as a single identifier. This is analogous to a method frequently used in computing, called *timestamping*. When some subject or object (e.g. a file) is created, a timestamp, which is a record of the time the object was created, is associated with the name. Timestamps are typically represented in milliseconds (thousands of a second) or in microseconds (millionths of a second).

Timestamping is especially useful with internal names, but it is not the sort of name which a user normally wants to associate with his file. We will see an example of timestamping in a later chapter.

3 Access Rights

Lampson's Matrix defines the *access rights* which a subject can exercise over an object. An *access right* gives a subject permission to carry out a particular operation on a defined object. Similarly a *set of access rights* confers on a subject the right to carry out various operations on a defined object.

3.1 Basic Access Rights

In computer systems there are at least two levels at which the exercise of access rights is of interest from a security viewpoint. At the hardware level the computer executes instructions which carry out the individual stages in a computation. In order to execute an instruction the instruction itself must be fetched from the main memory of the computer and in the course of its execution it may read values from and/or write values back to the main memory. Since these three actions on the main memory (the fetching of the instruction, the reading of information and the writing of information) are distinct actions at the hardware level, the hardware can detect what is happening and in most computer systems provides protection mechanisms which check whether read access, write access, and/or execute access is permitted, as it executes instructions. A *read access right* allows items stored in the computer memory to be read. A *write access right* allows items stored in the memory to be changed. An *execute access right* allows locations of the memory to be fetched as instructions. We call these kinds of access rights *basic access rights*. As we shall see in later chapters, different parts of the memory may be designated as readable, writable and/or executable.

At the hardware level basic access rights can be very useful. For example, given that program instructions and data items are usually held in the same memory, it is of considerable help when debugging (finding errors in) new programs if the hardware detects that an attempt is being made to modify constants (data in memory which should not be modified), or to execute data items as code, or to treat code as data. When the hardware detects such an error it halts the execution of the program. Without such protection the computer might for example execute many supposed "instructions" which are in fact data items, leaving the contents of memory in such a confused state that it would be difficult to reconstruct the chain of events leading back to the original error.

3.2 Semantic Access Rights

At the file system level information stored in the computer's secondary (disc) memory can be viewed as a collection of files, which may contain data or programs. At this level file access rights determine what operations may take place on these files. Many file systems simply reflect the basic rights which appear at the hardware level. Users may have read access, write access and/or execute access to a file. This access rights information is usually stored, along with other information, in the file directories.

In the case of file systems a read access right confers the permission to read the information which is stored in the file concerned. A write access right gives permission to modify information in the file. (Sometimes write access rights also confer permission to append new information to the end of files and/or to delete the files; alternatively these can be treated as separate access rights.) An execute access right in a file system confers the permission to invoke the file as a program containing executable instructions.

At the level of accessing files, the basic access rights provide some controls, but generally speaking they do not sufficiently distinguish between the different kinds of access controls which are needed in real world computer applications. For example the right to read a file does not sufficiently distinguish between the kinds of read operations which might be involved. It is one thing to allow a payroll clerk to access a company's personnel file to read information about an employee's salary, but it is quite another thing to allow a trade union official to read the file in order to obtain the names and addresses of employees. Payroll clerks have a right to read salary information, but trade union officials may not have this right. Yet in both cases the same file is being read. So simply giving the trade union official read access could lead to him also reading the salary information, which would be a clear breach of confidentiality. Yet without it, he could not read the information which he is entitled to have.

Similarly permission to write to a bank account file is insufficient to distinguish between allowing a bank teller to record withdrawals and deposits associated with a customer account and allowing the bank manager to raise the overdraft limit for a customer.

The point is that files contain logically related collections of data which consist of many individual items of information about objects, e.g. the names, addresses, account numbers, current balances, overdraft limits, interest payments associated with many bank accounts. Different people may have differing needs and differing entitlements to different parts of the information, which cannot be adequately expressed by a right to read from a file or to write to it.

The simplest, but least flexible, way to solve this problem is to partition the data into separate files corresponding to what may be read or written by different subjects. This is often inconvenient and does not correspond to the natural structure of data. It also creates difficulties when different users need partial access to the same information.

Another approach is to write separate programs which access the same file but only provide their users with that part of the information to which they are entailed to receive. Thus there might be separate programs for the payroll clerk and the trade unionist. In this case each user has the right to execute only some of the programs.

A third solution is to build a complex software mechanism to serve as a watchman. For example a data dictionary, implemented as part of a database system, might contain information about the structure of the data stored in the database. It is possible to record access control information in such a dictionary. Another watchman solution which is sometimes used is to have a special software package which works as an extension of the operating system.

The fourth possibility is the most flexible and most interesting. It involves defining the access rights in terms of the operations on objects as they appear in object-oriented systems. We shall talk more about object-oriented systems in a later chapter, but here is a small foretaste.

An object-oriented system is a software system designed around the idea of object classes. An object class defines the routines which, in a well-defined system, correspond to the "natural" operations for objects of that class. As an example a bank account can be defined as an object with an appropriate set of routines, as shown in Figure 2.7.

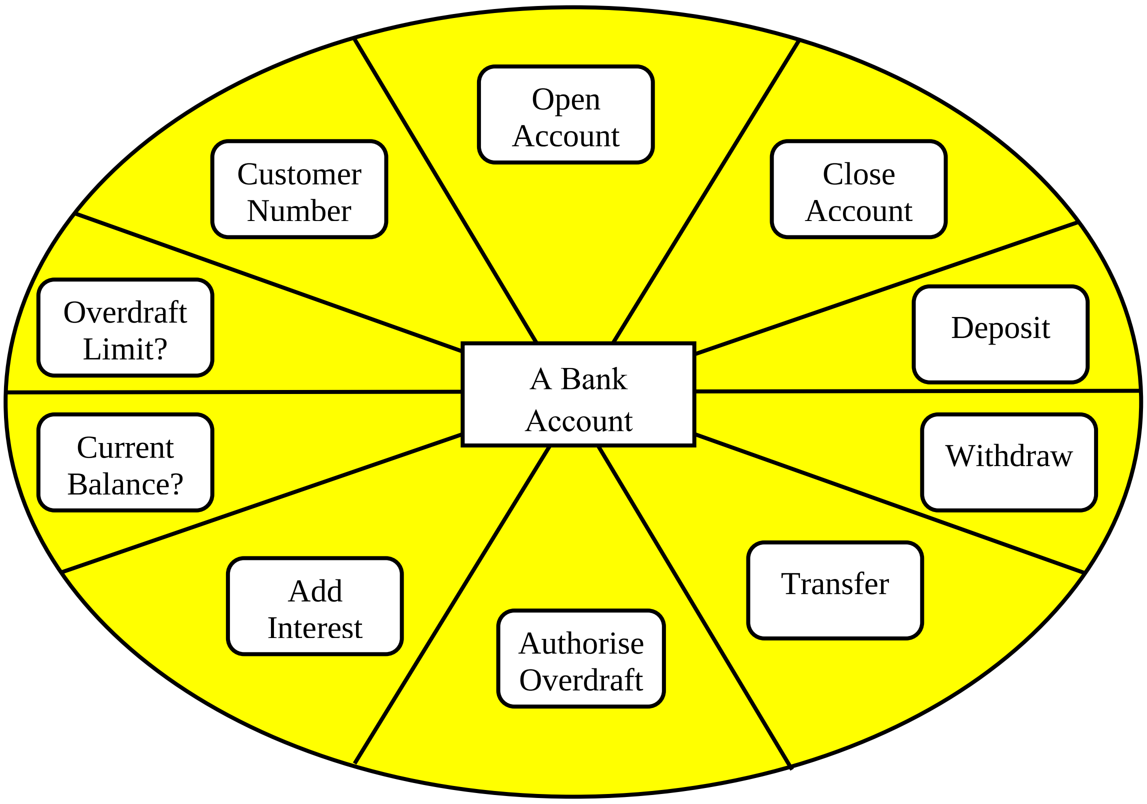


Figure 2.7: A Bank Account with Semantic Operations

We can divide these routines into two groups: operations, which in some way manipulate and change the state of the bank account, and enquiries, which provide the subject with information about the account without changing its state. 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 'overdraft limit?' and 'current balance?'. The enquiries can be viewed as different kinds of read routines, which return specific information to the caller.

Provided that the routines defined for a class of objects such as bank accounts correspond to the natural operations which occur in the real world, they naturally become ideal candidates for defining access rights. We can express such a set of access rights as shown in Figure 2.8.

Notice that this diagram is of a different kind from those which was used to illustrate Lampson's Access Matrix. An entire column in Figure 2.8 corresponds to a single access rights field of Lampson's Matrix. What this means is that Figure 2.8 refers to the access rights for a single object. 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 bank account, but that should not automatically give me the right to withdraw money from yours!

	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	√	√	√	√

A tick indicates that the subject at the head of the column may carry out the operation in the corresponding row.

Figure 2.8: Access Rights expressed as Semantic Operations

We call the access rights corresponding to natural operations on objects *semantic access rights*. In contrast with basic access rights, these allow us to distinguish for example between a bank teller's right to record withdrawals and deposits and the bank manager's right to raise the overdraft limit for a customer.

3.3 Generic Access Rights

It is often useful to augment semantic access rights by adding a small set of *generic access rights*. These are additional access rights based on operations which are common to all objects, for example the right to create an object, to delete an object, to copy an object, to change its ownership, etc.

3.4 Metarights

Finally access rights can themselves be subject to access control rules. For example, there may be access rights which determine the right to pass on access rights to another user, to change (e.g. restrict) one's own access rights, to change the access rights of others, etc. We call these *metarights*. This brings us to our next question. Who has the right to control access rights?

3.5 Mandatory Access Controls

There are two kinds of views about how this question should be answered. The first is the "authoritarian" or "organisational" view, which can be characterized as follows.

"The information to which access is being controlled belongs to some organisation. Each organisation has a head, who in principle has sole control of the database. This database can be accessed by others only according to his wishes, which may include delegating some of his control to others."

In other words, the organisational head (or his computer expert) controls the metarights in the system. Typically, large organisations seem to prefer a hierarchical access control structure, reflecting their management structures. Various security models are based on this way of thinking, such as the Bell-LaPadula model [10], which we discuss, along with some similar models, in a later chapter.

The authoritarian view of access control leads to systems which are characterized by *mandatory access controls.*

3.6 Discretionary Access Controls

The alternative approach, found in "open" systems, involves *discretionary access controls*. In a discretionary system each individual subject in the system is personally responsible for controlling access to the objects which he creates and owns. The discretionary approach is typical of time-sharing environments (such as Unix). Notice, however, that in such environments individual users are often not entirely free of external controls, since they usually have a "controlling" user, e.g. the Unix "superuser" or "root". We discuss this issue in a later chapter.

4 Implementing Lampson's Matrix

So far we have discussed issues relating to the subjects, the objects and the access rights which appear as the components in Lampson's Matrix. We now consider the question of how the Matrix might be implemented.

At first sight it might seem that the most obvious way to implement Lampson's Matrix in an operating system would be as a two dimensional array. However, there are at least two reasons why this is not a realistic approach in practice.

First, in a multi-user operating system an Access Matrix is usually very sparse. Most of the entries indicate that subjects have no access to most objects. Consequently a lot of memory would be consumed by repetitive information.

Second, an Access Matrix is very dynamic. In other words it is frequently changing. It is not only the access rights which change but, more importantly, subjects and objects are added and removed frequently. This would involve a great deal of adjustment to the rows and columns of the data structure.

For these reasons two alternative implementation models can be used in practice. The first of these is based on Capability Lists (sometimes called C-Lists), the second on Access Control Lists (usually known as ACLs).

4.1 Capability Lists

A Capability List is a list which exists separately for each subject. It lists objects to which the subject has access, along with the corresponding access rights for the objects. Thus a C-List corresponds to one column of Lampson's Access Matrix, and there is a separate C-List for each column. Entries in the Access Matrix indicating that no access is permitted need not be included in the C-Lists. The C-Lists corresponding to the Access Matrix in Figure 2.3 are shown in Figure 2.9.

C-List for Jill

My File	read, write
Her File	read, write

C-List for Editor

My File	read, write
Her File	read, write

C-List for Jack

My File	read
Your File	read, write
Editor	execute

C-List for Joan

My File	read
Your File	read, write
Editor	execute

Figure 2.9: Capability Lists for the Access Matrix in Figure 2.3.

An entry in a C-List corresponds to the general concept of a *capability*. Generally speaking, a capability consists of an object identifier, which should *uniquely* identify the object, and a set of access rights for that object. The possession of a capability implies the right to access the object in the ways defined by the access rights. Thus a capability can be thought of as being like a bunch of keys which will open some of the doors in a building. The building is the object, the doors which can be opened by the keys are defined by the access rights.

Just as in a key system it is relatively easy to distribute keys to those who need them (for example to students who need to work in the university at weekends) but is often difficult to get them back later (when the students have completed their studies), so in a capability system it is usually straightforward to distribute capabilities to subjects, but it can be difficult to get them back when they are no longer valid. This gives rise to a well-known problem in capability systems, the *capability revocation* problem. It can be illustrated as follows.

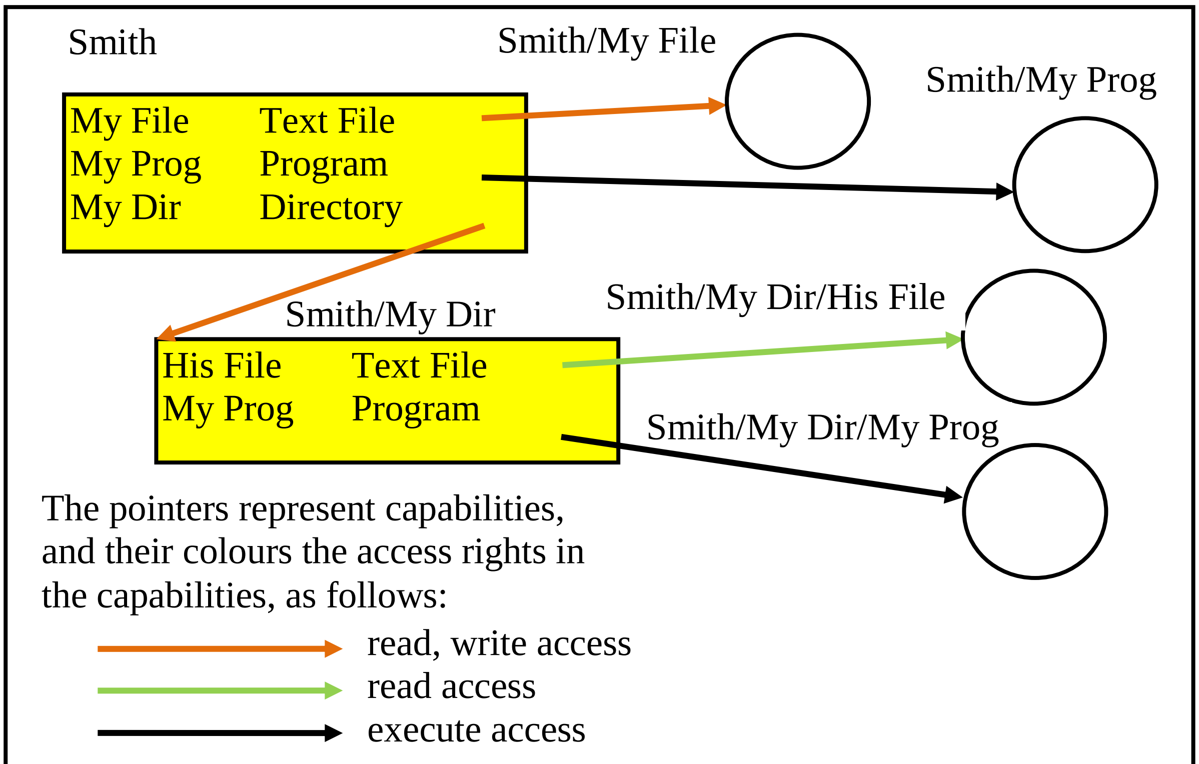


Figure 2.10: A Hierarchical Capability Directory Structure

In a practical capability based system a subject's C-List can be organized as a set of directories. Each entry in a capability-based directory might consist of a symbolic name (which need only be unique within the directory), some information about the object (e.g. its type), and a capability. The capability contains a unique object identifier, which serves as a pointer to the object (similar to the pointers in Figures 2.4 and 2.5), and the access rights for the object. Figure 2.10 is a revised version of Figure 2.5, showing a small capability directory structure for our user Smith.

Let us now suppose that the file we have called Smith/My Dir/His File actually belongs to Jones, and the capability with read access has been given to Smith by Jones. Jones will have his own capability for this file, with read and write access. He has possibly even given it a different symbolic name (Jones/My File), as is illustrated in Figure 2.11. This is an advantage of capability systems, and it works because the same object identifiers are used in the different capabilities, uniquely identifying the same object. It is as if Jones labels a key on the bunch as for "My Office" and gives a key for it to Smith, which Smith labels "Jones's Office".

But now we see the problem which Jones has, if he wants to revoke or modify the capability which he has given Smith to use. In order to take the capability back he needs to have access to Smith's directory called My Dir. Even if Smith had given him a capability for this directory, there is no guarantee that Smith has not moved the capability to another directory or made a copy of it. It is of course possible to devise extra rules about not moving and not copying capabilities, but these are often too restrictive to allow capability systems to function reasonably. In fact the problem is often worse than we have shown, because flexible capability systems do not even insist that capabilities are stored in directories. Smith might even have buried the capability in one of his programs. It is as if he has taken the key off his bunch and put it somewhere entirely different. So Jones has no idea where he should look to find it!

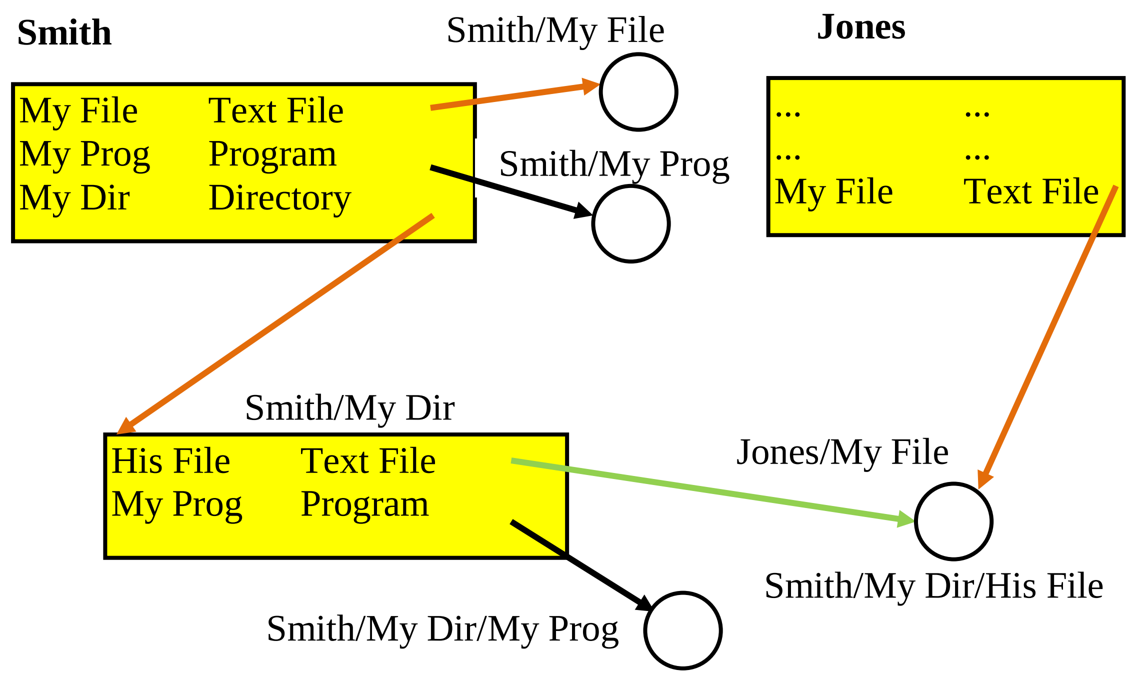


Figure 2.11: Two Directory Structures containing Different Capabilities for the Same File

There are in fact some solutions for the capability revocation problem, as we shall see in later chapters, but they depend on other aspects of how systems are implemented. There is no simple general solution which does not depend on the implementation nor restrict the freedom of users. There is even a view that the capability revocation problem should not be solved, because a capability should be seen as a guarantee of access [11, pp. 3-8].

4.2 Access Control Lists

An Access Control List (ACL) can be considered as the inverse of a C-List in that a list exists for each object, defining all the subjects who have access to the object, along with the corresponding access rights for these subjects. Thus an ACL corresponds to one row of Lampson's Access Matrix, and there is a separate ACL for each row. Entries in the Access Matrix indicating no access need not be included in the ACLs. Figure 2.12 illustrates the ACLs corresponding to the Access Matrix in Figure 2.3.

ACL for My File

Jill	read, write
Jack	read
Joan	read

ACL for Your File

Jack	read, write
Joan	read, write
Editor	read, write

ACL for Her File

Jill	read, write
Editor	read

ACL for Editor

Jack	execute
Joan	execute

Figure 2.12: Access Control Lists for the Access Matrix in Fig. 2.3

4.3 Differences Between ACLs and C-Lists

It would appear that ACLs and C-Lists can represent the same information, and it is therefore natural to assume that they are equivalent. In theory they are but, surprisingly, they are not equivalent in practice! In real systems they can have quite different effects.

Whereas a capability can be thought of as a bunch of keys for a building, an ACL is more appropriately compared with a watchman who sits at the entrance to the building and accompanies visitors to different rooms, ensuring that they can only enter those rooms for which they have permission.

Revoking or changing access rights is usually no problem in an ACL system. The owner of the building just advises the watchman of changes, which the latter writes down on his list. Next time a visitor comes, he can only enter if he is on the changed list. So much simpler than the capability system, it would seem! We see in Figure 2.13 how this looks in the case of our previous example.

Notice that entries in an ACL directory are not capabilities, which has the effect that there is only one directory entry per object. Smith now has no entry for the file we previously called Smith/My Dir/His File. Instead there is an ACL associated with the file itself which gives him permission to read it, but the file is now only called Jones/My File. Smith doesn't need a capability, he only needs to name the file, and the access list shows that he has permission to read it. (The subject name in the ACL should of course be a unique identifier, not just Smith!)

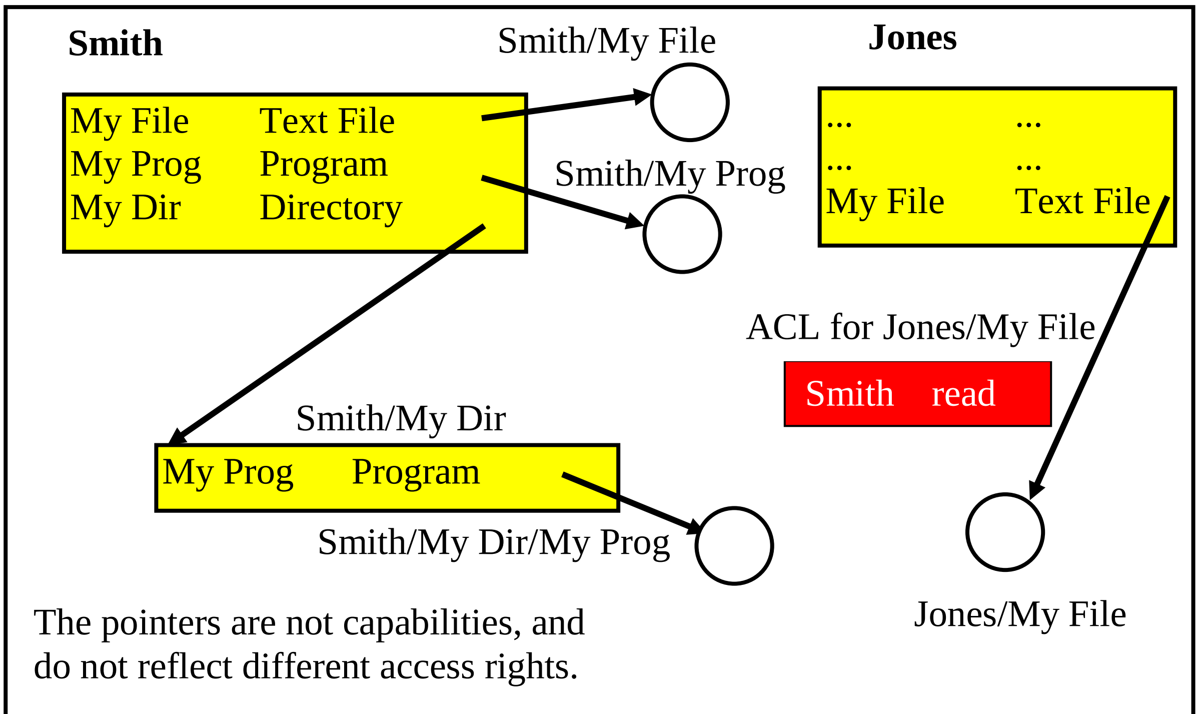


Figure 2.13: Two ACL Structures giving Access to the Same File

But now we have a problem. Smith has to be able to find the file in order that the ACL can be found and checked. How he does this depends on where the ACL is stored. In Figure 2.13 we have placed the ACL somewhere between Jones's directory and the file itself. In practice an ACL can be stored either with the object for which it defines access or with a directory entry describing the object. The latter is more usual.

Most ACL-oriented file systems solve the problem by having a single global directory hierarchy for all users of the system. The first level may represent a super-user (or the system itself), the next levels represent users, and the lower levels the users' individual directories and their files. In effect there is a single global hierarchical directory structure representing the ownership of all files in the system, as shown in Figure 2.14. This is a simplified view of directory structures such as that found for example in the Unix system.

With this kind of system users have no difficulty in finding the files to which they have access via an ACL, if, as is usual, they can browse through the global directory structure. But herein lies a big danger for security. If a single global directory structure exists, hackers can take advantage of this to discover what interesting files there are in a system, making it easy for them to locate potentially useful information. (If I have called one of my files "Computer Architecture/Exam" I dare say it might be of interest to some student hackers!) And once a hacker is in a directory then with a little skill he can manage to include himself in the access control lists for the objects held there.

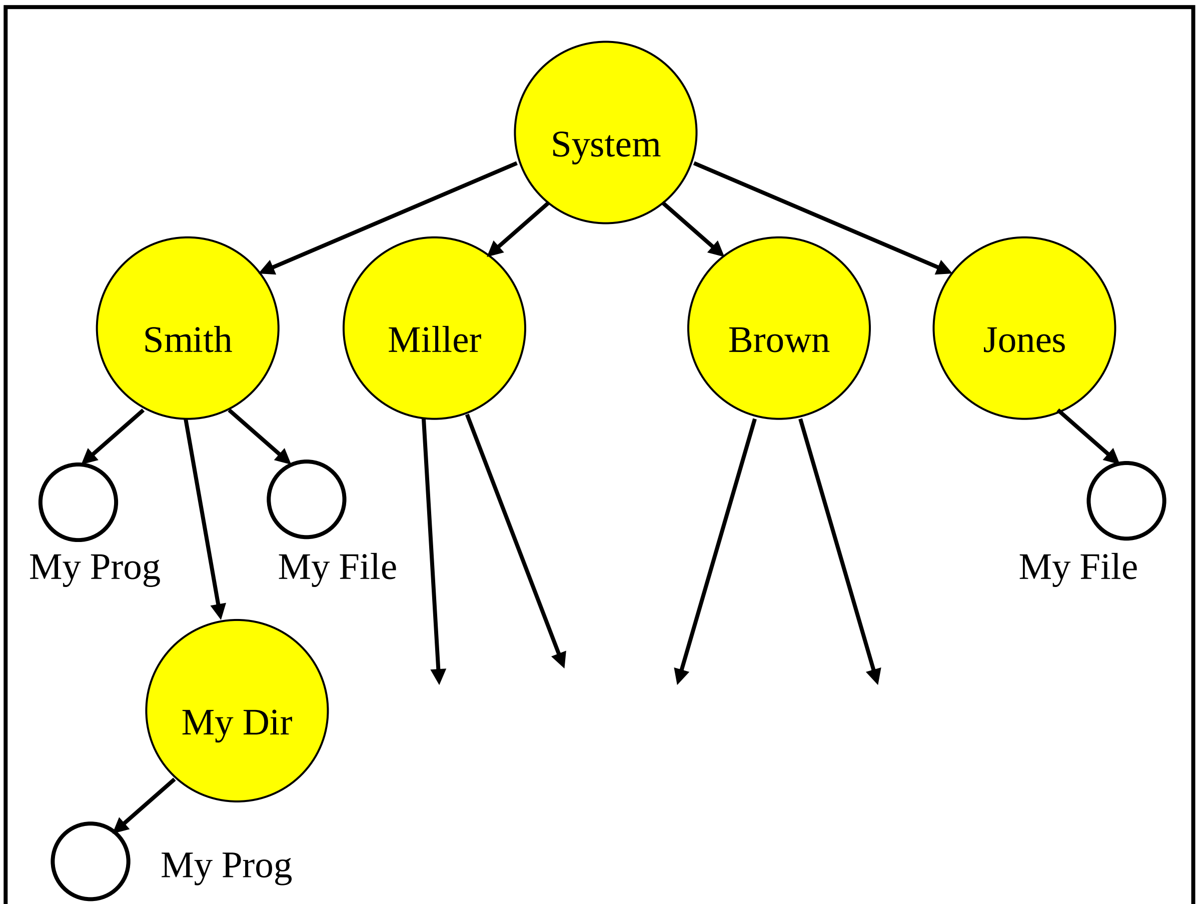


Figure 2.14: A Global Hierarchical Directory Structure

By contrast, in the case of a capability based directory system a global structure is unnecessary, and it is easy to ensure that hackers cannot browse through all the files in the system. The need to browse has been removed!

The alternative strategy for storing an ACL, i.e. storing the access rights with the object itself rather than in a directory, would in conventional file systems have the effect that it would possibly be harder for hackers to find the access rights, but it would be equally difficult for users to find the files of others to which they have legitimate access! This strategy is not normally adopted in current systems, because it would complicate the file system too much.

We shall return to the subject of C-Lists and ACLs at various points later in the book. We have by no means discussed all the important points which they raise. In particular we have mainly looked at them in relation to file systems. But issues of representing access rights arise at several levels in a system. In later chapters we shall be especially concerned with resolving such issues at lower levels of the system, for example in relation to the protection of pages and of segments in the virtual memory. Although the principles which we have discussed apply at that level also, quite different implementation decisions play an important role, as we shall see in due course.

In the next chapter we examine some further security concepts.

Chapter 3
More Security Concepts

The previous chapter described the properties and possible implementations associated with Lampson's Access Matrix. In this chapter we review some security issues which cannot be resolved via that simple model.

1 The Confinement Problem

An example of a problem which cannot be described in terms of Lampson's Access Matrix is known as the *confinement problem*. This is basically the problem of ensuring that subjects who/which themselves have a legitimate right to information do not reveal it to unauthorised third parties.

As a simple example, consider the process of printing a file. The actual low level commands which have to be issued to printer devices (e.g. ink jet printers, laser printers, etc.) are not only complicated, but they differ quite substantially from one printer type to another. It would be an unreasonable overhead to expect all programmers of application programs to produce their own code to drive printers, so what normally happens is that printer driver programs which offer a uniform and easy to use interface are provided by the operating system or the printer manufacturer. But can these driver programs be trusted not to reveal the information to which they need access in order to print user's files?

Furthermore, it is in the interest of all users in a multiprogramming system that the printers have a high throughput and that individual users do not have to wait until a printer becomes free before they can run their application programs which produce printout. Both these objectives are usually achieved by using printer "spoolers". These are processes which print continuously as long as there is something to print. They take their input from files which have been created on disc by application programs. Thus the application programs never write their printout directly to the printers but instead they write what is to be printed into files on disc. (Discs have two advantages over printers: the rates of data transfer are much faster, and they are sharable devices which can be used "at the same time" by many different programs.)

What all this means is that when an application program wishes to have information printed out, it first writes this to a disc file, then it puts the name of (or places a capability for) the file onto a queue of work for the spooler process.

The spooler is expected to read the file and output its contents to a printer. But by reading the file the spooler has (legitimate) access to potentially secret information. The confinement problem in this case is how to stop the spooler from secretly printing user files at another location or from copying them into different files with access rights that allow unauthorised parties to read them.

A program which cheats in this kind of way is often known as a *Trojan horse*, i.e. a program which in addition to its "official" job carries out undesirable secret actions. What is needed is the ability to "confine" the spooler in such a way that it cannot retain information which it receives from its caller and cannot copy it or output it other than as required by the application program using its services.

The confinement problem cannot be formulated in terms of Lampson's Access Matrix (and therefore also not in terms of C-Lists or ACLs), and it is also an extremely difficult problem to solve in practical systems. That is one of the reasons why we hear such much in news reports about insecure computer systems.

To make matters worse a Trojan horse need not necessarily use an overt channel (such as a file or a printer) to pass information to unauthorised users. It may resort to *covert* channels. Suppose for example a spooler had access to a simple piece of information ("yes" or "no") which it wants to pass on illicitly to a third party, it might encode this information by printing two files in the order A then B, meaning "yes" or in the order B then A (meaning "no"), or it might cause a noticeable delay (e.g. 20 seconds for yes, 40 seconds for no), between printing two files, or send an unwarranted error message, etc.

In a later chapter we shall show how the simple confinement problem can be solved, but we do not pretend to have a general answer to the problem of closing all covert channels! When we discuss the Bell-LaPadula model later in this chapter we shall see a different example of the confinement problem, but before doing this we first look at some further security issues which are not addressed by Lampson's Access Matrix.

2 Rule-Based Access Rights

The simplest way to express access rights is to say that a subject S may access an object O with a set of (basic or semantic) access rights R. This is basically what Lampson's Access Matrix achieves. However, this is often not fully adequate to meet particular situations. For example, it may be desirable to determine that a subject S (e.g. an employee) may have access rights R to an object O *only during working hours*. The latter is a simple example of rule-based access. Another example would be the rule that the owner of a bank account may make a withdrawal operation *only if this does not result in an account balance less than the overdraft limit*. We shall see a quite different example of access rules when we consider the Bell-LaPadula model.

We shall refer to the simple Access Matrix form of access controls as *unconditional* access rights, in contrast to those with associated conditions, which we call *rule-based* access.

In our discussion of the issue who controls access rights we distinguished between *mandatory* and *discretionary* access rights. It is important to note that the concepts of *unconditional vs. rule-based access* and of *mandatory vs. discretionary access* are orthogonal, i.e. they are independent themes which can be combined in any way (giving four possible kinds of system). In other words both unconditional and rule-based access rights can be used independently of the decision about whether a user himself or an organisation head determines the metarights for the objects of a system. This point is often not clearly recognized, because most mandatory systems (cf. Bell-LaPadula) are also rule-based and most discretionary systems (cf. Unix) support only unconditional access rights.

3 The Access Rule Model

A few years ago a colleague and I proposed what is effectively a simple extension to Lampson's Access Matrix, which we called the Access Rule Model. This allows rules to be specified as part of the process of defining access rights [12, pp. 67-82]. As it is intended to be a model which allows security decisions to be formulated in a formal way, the rules look a little mathematical. But don't let this put you off. The idea is basically very simple. The description which is presented here differs a little in detail from the original description.

An *access rule* specifies a condition which must be fulfilled in order that access may proceed. It takes the form

```
condition: subject –> object.{access_rights}
```

The rule

```
C: S –> O.{AR}
```

means that the subject S has the set of access rights AR for the object O, if and only if the condition C is fulfilled at the time the access is attempted. If the access rights are listed separately, a comma is used to separate the individual access rights in the set.

The *condition* is a logical expression, which may use predicates about any aspect of the system.

Lampson's Access Matrix is a special case of the model, which can be expressed as follows:

```
true: S -> O.{AR}
```

What this means is that it is always true that the subject S has the access rights AR for the object O. Thus we can express the first column of the Access Matrix in Figure 2.3 using the following rules:

```
true: JILL -> My File.{read, write}
true: JILL -> Her File.{read, write}
```

By using the quantifier "for all", which is abbreviated to the symbol ∀, in conjunction with subjects, access rights and/or objects, we can neatly express in a single access rule many fields of Lampson's Matrix. Thus

```
∀S: S -> spooler.{print}
```

This means that all subjects, here referred to as S, have the right to print using the spooler. Similarly

```
∀O: Superuser -> O.{read}
```

means that the Superuser can read all objects in the system, here referred to as O. The rule

```
∀AR: Smith -> My File.{AR}
```

means that Smith has all access rights, here referred to as AR, for My File.

By introducing sets, the rules can discriminate more finely. The symbol ∈ means "in the set". For example the rule

```
∀x ∈ bank_tellers: x -> account.deposit
```

means that all members, here referred to as x, of the set bank tellers (i.e. all bank tellers) can make deposits into the object account.

More complex conditions can be expressed by using boolean operators in the conditions. For example the symbol ∩ means "and" and ¬ means "not", so that the rule

```
∀x ∈ bank_tellers ∩ ¬ account.overdrawn:
                                        x -> account.withdraw
```

is interpreted as: all bank tellers have the right to withdraw from the account if it is not overdrawn.

These examples illustrate that the access rule model fits well with both the object-oriented approach to software design (which we further discuss in a later chapter) and with the use of semantic access rights, which we discussed in the previous chapter. The expressions *account.overdrawn* and *account.withdraw* can be understood as operations on the object *account*.

It is even possible to express confinement if we introduce the predicates *confined* (x) (i.e. the operation x is confined) and *confined_to* (y, z) (i.e. the object y is confined to invoking the object z), as follows:

```
  confined(print) ∩ confined_to(spooler, printer):
                                      S -> spooler.print
```

What this means is that the subject S can invoke the print operation of the spooler if the operation print is confined and the spooler is confined to using only the printer.

The use of such a general access rule model means that it should in future be possible to specify in a precise form what the security requirements for a system are. This is an important step in achieving secure systems.

However, specifying requirements and implementing them are two quite different things. For example, specifying a confinement condition does not mean that it is easy to implement it!

4 The Bell-LaPadula Security Model

This model [10] is a rule-based mandatory model which aims to control the flow of information between subjects. Through its incorporation into the U.S. Orange Book[7] and similar publications by other governments it has substantially influenced general thinking about security.

Each subject and each object in the system is classified as belonging to a particular security class. A subject has a *clearance* level and an object (e.g. a document) has a *classification*. For example we could use the military classifications

```
  unclassified < confidential < secret < top secret
```

where the symbol < means "is at a lower level than" (e.g. *confidential* is at a lower level than *secret*).

The aim is to control information flow between different subjects. For example subjects with a clearance level *secret* may read objects (which we shall call documents) with a lower classification *confidential* or *unclassified*, but they may only write to documents classified as *top secret*. Objects at the same classification may be both read and written. Thus a subject with the *secret* clearance level may both read and write *secret* documents.

In addition there is a non-hierarchical grouping into areas of concern (which we shall call projects). These are disjunct, i.e. projects do not overlap with each other. Each subject and each object may be associated with a set of projects. A subject may only read a document if he is a member of all the pro-

[7] The U.S. Orange Book defines a set of criteria laid down by the U.S. Department of Defense (DoD) in 1985 for evaluating the trustworthiness of computer systems. It has now been withdrawn.

jects with which it is associated. He may only write to a document if the document is associated with all the projects of which he is a member.

The hierarchical classification rules and the non-hierarchical project rules are applied in combination. In other words both sets of rules must be fulfilled before an access may take place.

The rules for reading and writing can be expressed mathematically as follows:

Reading of Objects (simple security property):

```
(clearance (subject) ≥ classification (object))
      ∩ (projects (subject) ⊇ projects (object))
```

This rule ensures that a subject cannot receive information from higher classification levels or from projects of which he is not a member.

Writing of Objects (*-property, pronounced 'star-property'):

```
(clearance (subject) ≤ classification (object))
      ∩ (projects (subject) ⊆ projects (object))
```

This rule ensures that subjects cannot transmit information to lower classification levels or from projects not associated with the document.

In addition there is a rule regarding the introduction of new users into the system. This states that a subject S can only create a subject T if the projects for T are a subset of those for S and the clearance of T is not higher than that of S.

Creation of subjects:

```
Subjects creates Subject ⇒
 (projects (Subject) ⊆ projects (Subjects)
     ∩ (clearance (Subject) ≤ clearance (Subjects))
```

All these rules can be easily expressed using the Access Rule Model, as follows:

```
∀O ∩ ∀S ∩ (clearance (S) ≥ classification (O)
     ∩ (projects (S) ⊇ projects (O)):
          S -> O.{read}

∀O ∩ ∀S ∩ (clearance (S) ≤ classification (O))
     ∩ (projects (S) ⊆ projects (O)):
          S -> O.{write}

(projects (St) ⊆ projects (Ss)
     ∩ (clearance (St) ≤ clearance (Ss)):
          Ss -> user_manager.{new_user (St)}
```

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.

However, it does not guarantee the *integrity* of objects, because it permits subjects with a lower clearance to write to objects with a higher classification. Some researchers (e.g. [13]) have therefore suggested that the writing rule should be modified to disallow the writing to or creation of objects at higher classification levels.

Other models for controlling the flow of information have been proposed (e.g. the Lattice Model [14]).

5 The Biba Security Model

Not all security models are concerned with controlling the flow of information. A model superficially similar to the Bell-LaPadula model, the Biba Model [15] is also a rule-based mandatory model, but – unlike Bell-LaPadula – it is concerned with guaranteeing the *integrity* of information. For another example of a security model aimed at ensuring the integrity of objects (the Clark-Wilson Model) see

As in the Bell-LaPadula model subjects and objects are classified into hierarchically organized security classes. A subject has a *clearance* level and an object (e.g. a document) has a *classification*. In addition there is a non-hierarchical grouping into areas of concern (e.g. projects). Such projects are disjunct.

The Biba model has the following rules:

Reading of Objects:

```
(clearance (subject) ≤ classification (object))
     ∩ (project (subject) ⊇ project (object))
```

This rule ensures that subjects cannot receive information from lower classification levels or from projects of which he is not a member.

Writing of Objects:

```
(clearance (subject) ≥ classification (object))
     ∩ (project (subject) ⊆ project (object))
```

This rule ensures that subjects cannot transmit information to higher classification levels or from projects not associated with the document.

Creation of subjects:

```
Subjects creates Subject ⇒
      (project (Subject) ⊆ project (Subjects)
            ∩ (clearance (Subject) ≤ clearance (Subjects))
```

This rule states that a subject S can only create a subject T if the projects for T are a subset of those for S and the clearance of T is not higher than that of S.

Whereas the subject creation rule and the project membership rules for reading and writing are the same as in Bell-LaPadula, the *hierarchical* reading and writing rules of the Biba model are the inverse of those in the Bell-LaPadula model. This ensures the *integrity* of objects but not the confidentiality of information. It does not solve the confinement problem [16].

In bringing our descriptions of some standard security models to a close, we would like to emphasize that from the viewpoint of this book the important issue is not which security model is best or even good. These models are important to us rather because they serve as *examples* of real security models which cannot easily be implemented on conventional computers using conventional implementation techniques. Thus they provide a practical testbed for the usefulness of the security mechanisms which we shall propose in the later parts of this book.

6 Protection Domains

The mandatory security models which we have seen – and in fact most security models – make an implicit assumption that only one protection domain exists in a particular computing environment, and that the protection mechanisms of the system are directed towards implementing that particular security policy or model. And in practice current systems often support only a single protection domain.

In later chapters we shall take up the challenge of showing that it is possible to provide mechanisms which are flexible enough and yet secure enough to implement different models side by side in a single system. In particular we shall attempt to demonstrate that it is possible to implement a combination of policies involving mandatory (authoritarian) and discretionary (open), as well as unconditional and rule-based models alongside each other. Such mixed systems are of practical relevance both for providers of computing and networking services.

The general conclusion which we might draw from this chapter is that there are some security problems which are not too difficult to explain or even specify formally. But they are by no means easy to implement in a general way. This provides us with one of the challenges to be overcome using the security mechanism which we propose in the remainder of the book.

Chapter 4
External Security Threats

So far we have considered security in a fairly abstract way. In this and the next chapter we turn to more practical issues, taking a look at some of the threats and weak mechanisms which in practice place computer systems at risk. We first review the threats which arise from unauthorised persons attempting to penetrate a system. Then in the next chapter we review the kinds of internal threats which can be created by persons who are either legitimate users of the system or by unauthorised users who have managed to penetrate the system, concluding with a discussion of some of the weak security mechanisms and policies which make it easy for them to breach security.

1 Threats from Outside

Most systems include a list of users, who – assuming that they can provide authentication of their identities – are authorised to use the system. There are several groups of persons who may *not* be authorised to use a system, who may nevertheless be interested in penetrating the system. Here are some examples.

— Amateur hackers often penetrate systems simply to satisfy their curiosity or to prove that they can beat the challenge of breaking in. Although this may be relatively harmless it can involve breaches both of the privacy rights of individuals and of the confidentiality of corporate information. If curiosity is accompanied by mischievousness or malice then the result may also be threats to the integrity and/or to the availability of systems.

— Criminals may wish to penetrate a system for their own benefit. The aim is often to commit a financial crime and may involve breaching either the confidentiality of the system (e.g. to obtain "insider" information) or the integrity of the system (e.g. by modifying information about financial transactions or bank accounts, etc.). They are less likely to threaten the availability of the system, especially if they wish to go undetected.

However, it has unfortunately become quite common for criminals to blackmail the owners of computer sites by encrypting them, with the result that the computer system becomes useless (unavailable). Anonymous payments are now possible in the Darknet[8] using cryptocurrencies[9] such as Bitcoin[10], which thus facilitate blackmail payments.

— Commercial and industrial spies generally wish to obtain information by penetrating competitors' systems, and so they are primarily a threat to the confidentiality of the system. They may of course also modify information to disadvantage their competitors. Only in extreme cases is it likely that they will wish to disrupt the availability of the system, as they too will normally be more concerned to go undetected.

— Social media and other companies avidly gather information about as many individuals and companies as possible with the primary aim of building profiles which can be used to place customised advertising especially in the Internet. While this is not always illegal it often represents a severe threat to privacy[11], and in the longer term (because of the massive databases which are accruing and the ability to manipulate users' opinions) to democracy [17]. The most common technique which they use is to install trackers which follow and record their browser history, i.e. their accesses to internet sites, especially websites. The most common reason for this is to provide advertisers with information which will help them to sell their goods and/or services, but there are other uses (e.g. for law enforcement authorities follow the activities of criminals and terrorists[12]. Trackers often use cookies on the computer of their targets to store their information.

— Whilst it cannot be excluded that terrorists aim to obtain and modify information, they are more likely to be concerned with disrupting or destroying a system, and so mainly represent a threat to the availability of systems.

— Military and governmental spies and hackers are interested in gathering information about the activities of other countries (and sometimes about their own citizens) and therefore present a threat to the confidentiality of systems. Disinformation techniques may also lead to the modification of information and so provide a threat to integrity. And with the growing threats of cyber warfare a major aim has become to disrupt availability.

[8] see https://en.wikipedia.org/wiki/Darknet

[9] https://en.wikipedia.org/wiki/Cryptocurrency

[10] https://en.wikipedia.org/wiki/Bitcoin

[11] The European Union has introduced extensive privacy laws, which force websites to reveal information about the privacy relevant activities of their websites and allow the user to restrict some of these. However I doubt the efficacy of this, since it considerably slows down their work, and I suspect that most users simply take the line of least resistance by allowing all activities. I see this as a typical example of how officials often fail to realise their good (but bureaucratic) intentions!

[12] see https://en.wikipedia.org/wiki/Web_tracking

— It appears that foreign governments are now infiltrating the social media networks in what might be successful attempts to pervert the outcomes of democratic elections.

Although these are quite distinct categories of persons who may wish to penetrate systems, they have at least one thing in common. Before they can do their damage they must in some way gain access to the system or to the information they need. Here are some of the techniques which they can adopt.

They can physically steal information, e.g. by breaking into an organisation and stealing magnetic media devices on which information is stored. There are at least three kinds of precautions which can be taken against this kind of threat:

— physical security measures to prevent theft,

— holding multiple copies of data to ensure that theft does not lead to unavailability,

— making the data meaningless for the thief (e.g. using encryption techniques).

As a further possibility they can listen in to communications transmissions, e.g. by wiretapping, by observing internet traffic or by monitoring satellite messages. This technique may be used directly to obtain the information being sought. But it can also be used for example to discover the passwords of users coming on line. The most effective form of defence against this kind of threat is to encrypt the data, i.e. to encode it into a form which makes it appear meaningless.

The third possibility is for an unauthorised person to present himself as an authorised user of the system, e.g. by stealing or guessing a user's password or by systematically testing passwords until successful.

In this book we are concerned primarily with *technical* mechanisms for achieving secure systems, and so we ignore the issue of physically securing the data. Furthermore the focus of our interest is primarily on a secure computer architecture and the basic aspects of operating systems, rather than on the Internet[13].

From this review it is evident that at least two mechanisms are relevant for countering attempts to penetrate systems: encryption and authentication techniques. We now consider these in turn.

2 Avoiding Eavesdropping

Avoiding eavesdropping on communications lines or open channels such as satellite links can best be achieved by the use of encryption techniques. Such tech-

[13] I do not claim to be an expert on the Internet.

niques are highly mathematical and it is not our intention here to discuss detailed algorithms, but merely to sketch out some of the aspects which are of general interest for preventing security breaches.

The basic idea of encryption is that a text or bit string, the content of which is to be kept secret, is transformed by an algorithm into a different form, so that the meaning is no longer self-evident.

For example, suppose we start with the following text:

```
this is a secret text
```

We could use a simple algorithm to transform it into the following text

```
uijt jt b tfdsfu ufyu
```

and then send it over the internet in the hope that if an eavesdropper manages to access the message he will not understand it. In fact our hopes will almost certainly be dashed. You have probably already worked out the simple encryption algorithm which was used. All we did was change each letter in the text to the next letter in the alphabet.

We could make the algorithm rather more flexible by introducing the idea of a *key*. For example we could apply a three letter key, which the algorithm uses to determine by how many letters in the alphabet the letters in the text are to be shifted. Thus we might use the key BIT to produce the following encoding of the text:

```
vqcu rl c byeayv cyzc
```

What we have done is to take the first, fourth, seventh … letters and have shifted them by 2 (because B is the second letter of the alphabet). Then we have taken the second, fifth, eighth … letters and shifted them by 9 (because I is the ninth letter of the alphabet) and finally we shifted the third, sixth, ninth … letters by 20 (because T is the twentieth letter of the alphabet). To implement such an algorithm we need a routine which is parameterised.

The advantage of using a key is that different keys can be used, at the will of the encoder (not the programmer), to produce different encodings. Even if the eavesdropper knows the algorithm, he still has to find the key.

During and since the Second World War a huge amount of effort and government money has been invested in encryption techniques and in ways to crack them. As a result relatively safe encryption algorithms have been devised. These are highly mathematical. We can usefully distinguish between two kinds of algorithms.

Symmetrical encryption algorithms (cf. DES [18]) and more recently AES[14]) use the same (secret) key to encrypt and to decrypt a text. The simple

[14] see https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

example which we have just given is symmetrical. If you know both the key that was used to encode the text and the algorithm, you can decode the text.

Asymmetrical algorithms (known as "public key" encryption) use two keys, a *public key* and a *private key* [19]. The public key, which need not be kept secret, is used to encode the message, but the private key, which has to be kept secret, is needed to decode it.

Both methods have their advantages and disadvantages. From the organisational viewpoint an important advantage of symmetrical techniques can be that the encoded text can be the same length as the plain text, e.g. it takes up the same amount of disc space. Another advantage is that symmetrical techniques can use very much faster algorithms than the asymmetrical technique. But a disadvantage is that the recipient of a message which has been encoded must know the same key as the sender. This can mean that the key may also have to be transmitted and is therefore at risk of being discovered by eavesdroppers.

In a public key system no keys have to be transmitted in this way. Each agent has his own private key and his public key. An agent who sends an encoded message uses the public key of the receiver to encode it. The recipient uses his private key to decode it. He doesn't need to reveal the private key to anybody. The recipient of a message can even publish his public key in a newspaper if he wants to. But this advantage does not come free. First, a public key algorithm works very much more slowly than a symmetrical algorithm. The other problem is that the encoded text is normally longer than the plain text.

To take advantage of the strengths of both methods, they can be used in combination. The text to be transmitted is encrypted using a symmetrical algorithm. This has the advantages that the encoding is fast and that the encoded text is no longer than the original. The key which was used for the symmetrical encoding is then itself encoded with the public key of the receiver and transmitted to him. He decodes this with his private key and then decodes the text.

A combined method along these lines can be used for example for communication between the different computers in a network, if each computer has its own public and private keys. Variants of this technique can also be used to enable a mutually suspicious sender and recipient of a message to authenticate the other's identity [20]. But this is by no means a simple issue and there is a large body of literature on this subject.

However, we shall not consider the issue of using encryption in attempting to authenticate the identities of mutually suspicious agents, as this is not the subject of the book. Instead we shall concentrate on issues concerned with the structuring of computer architectures and operating systems. We shall see that the SPEEDOS operating system can also make a contribution to the problem of identifying senders and recipients of messages.

3 Encryption of Information on Disc

Information stored on removable discs (including external hard discs) and similar media is vulnerable to theft. Having stolen or copied a disc the thief can take it to his own computer system and read the information at his leisure. In this case too encryption can play a role in safeguarding the *confidentiality* of the information. (To ensure the *availability* of the information the user must have made a second copy before the theft.) In practice few systems encode information on disc, so what we describe in this section are potential scenarios.

In practice only the symmetrical method of encryption comes into question because of the faster speed of encoding and decoding and because the information remains the same size. The latter is especially important for discs, since they work with fixed size data blocks which correspond to fixed size blocks in the main memory of the computer.

In the simplest scenario the operating system can encode the plain data form of the information when it is written from the main memory to disc and decode this whenever it is read back into the main memory.

There are several options for using keys. The simplest is for the operating system to use the same key for all discs which it uses, but this increases the ease with which a thief can crack several discs which are stolen. Thus a stronger option is to use a different key for each disc. It is even safer to use a different key for each file on each disc.

Then there is the issue of choice of keys. This can be made by the operating system or by the owner of the disc or of the file in question.

There is also the question of whether and how the keys are themselves remembered. The first alternative is for the keys not to be stored on disc at all, but then the onus lies with the owner of a disc or file to remember the key and to provide this to the operating system when he loads the disc or when he opens each file. This is both inconvenient for users and is open to the risk of forgotten keys.

For these reasons it is probably more attractive to store the key(s) on the disc or with the file in question. This has the advantage that the user can then simply load his disc on a different system (assuming it provides the same encryption facilities) and they can then be read without difficulty.

But if the owner of the disc can read it, why not a thief? Here the public key approach can be a help. Just as the sender of information over a communications link can send his symmetrical key using the public (asymmetrical) key of the recipient, so a user planning to take his disc to another system could request the operating system to encode his symmetrical key(s) on the disc, using the public key of the system on which the disc is to be read. This would make the key inaccessible to a thief trying to read the disc on his own system.

4 Authentication of Users

When a user logs into a system, his authentication usually takes place in two stages. First he is asked to identify himself. We shall refer to this stage as the *identification* stage. Then he is required to provide proof that he is the user he claims to be. This is the *authentication* stage.

The identification stage normally involves the user in typing in the (unique) username with which he is registered in the system. The operating system needs this name in order to set up an appropriate context for the user to work in. Usually this involves creating a process and executing some commands in this process to tailor it to the user's needs (as well as ensuring that this process has the correct security context).

5 Password Systems

The authentication stage can take several forms. In most systems the user is requested to supply a password, which is a secret code known to him and to the system. The way this security mechanism is usually organised can lead to several serious security weaknesses.

When users choose their own passwords, they naturally tend to choose names which are easy to remember. For example there is a tendency to use the name of one's spouse or parent or child. Even if the choice is not so obvious, many users are likely to choose a normal word which appears in a dictionary.

In 1979 Morris and Thompson [21] demonstrated how easy it is to discover most passwords. Within a week they cracked 86 % of 3,289 passwords! Ten years later a similar study by Riddle, Miron and Semo [22] produced similar results.

Another weakness of some password systems is that the passwords themselves are stored by the system in a password file. If a hacker succeeds in secretly penetrating this file, he can impersonate any user and thus obtain easy access to all the files in the system. In older versions of the Unix operating system it was even possible for all users to read the password file. In this case the passwords themselves were stored in encrypted form, but this does not prevent users from seeing at a glance which other users do not have a password. Furthermore, the encryption algorithm is publicly known and can therefore be used to carry out comparisons with entries in the password file.

There are several methods which can be employed to make systematic attempts to discover passwords. For example a hacker can wiretap the connection between a user and his system in order to discover his password. Or he can use "brute force" methods such as trying out all combinations of the alphabet of valid password characters or trying out all the words in a dictionary.

One way of countering brute force methods is for the system to adopt a policy of allowing a user only a small number of login attempts. If he exceeds the limit then the system refuses any further login attempts for say 24 hours. But this only stops the most blatant and direct attempts at impersonation. If the hacker has succeeded in obtaining encrypted passwords, for example from a wiretap or from a password file, then he can combine the brute force methods with encryption algorithms to crack passwords at his leisure. And of course the computer is an ideal tool to help him in such an endeavour.

A more sophisticated method of discovering the passwords of others is the use of so-called Trojan horses. We shall discuss this in the next chapter.

6 Improving the Security of Passwords

The literature is full of suggestions for improving the security of passwords. Here are some examples.

6.1 Password Length

The shorter a password is, the easier it is to crack by brute force, since the number of combinations of letters in the permitted alphabet rises sharply with the length of the password. For example if the alphabet consists of only the 26 small letters from *a* to *z*, and a password can be a single letter, then at most 26 attempts suffice to crack the password. If it has two letters then at most $26^2 = 676$ attempts are needed. With three letters the number increases to $26^3 = 17{,}576$, with 7 to $26^7 = 8{,}031{,}810{,}176$ and so on. With ten letters the number is more than 141 thousand billion (where 1 billion is a thousand million). To understand what this means, let us suppose that it takes one tenth of a second of computer time to make an attempt to crack a password (which is in practice far too long!). With three letter passwords the longest time needed is 1,757.6 seconds, just less than half an hour. But with 10 letter passwords it would take 14.1 thousand billion seconds to crack every password, which is nearly 450 thousand years! But as I indicated, one tenth of a second is unrealistically slow.

6.2 Range of Characters

A further improvement can be gained by increasing the range of characters which can be used in passwords. For example the use of both small and capital letters theoretically makes a big difference. For example with even three letter passwords this gives a range of $52^3 = 140{,}608$ possibilities, compared with 17,576 for just small (or just capital) letters. If the ten decimal digits are added then this increases to 62^3 = 238,328, and if we add ten extra characters, such as full stop, comma, hyphen, question mark and so on, then with even three characters the number of possibilities is now 72^3 = 373,248.

One problem with increasing the length of passwords and/or with widening the range of the alphabet used, is that users will tend not to make use of the extra length or the extra characters, because their passwords become more difficult to remember. To avoid this, some systems place restrictions on the passwords which their users are permitted to register. For example they might say that a password must contain at least one capital letter, one number, and a special character and be at least eight characters long. Other restrictions which might apply are that passwords which appear as words in dictionaries or passwords are not allowed. It might also be required that passwords are changed at regular intervals, e.g. at least once a week or once a month.

But generally speaking such systems are not popular with users, because they make it hard to remember passwords. Even less popular are systems which themselves determine which passwords are to be used.

6.3 Complicated Password Requirements

The effectiveness of making passwords more complicated is in fact somewhat questionable, since the more difficult a password is to remember, the more likely it is that the user will write it down on a piece of paper or store it in a computer file. If he does this the vulnerability of the system rests on this piece of paper being lost, stolen, or left at a computer terminal, etc., or on a hacker breaking into his system and finding the file.

6.4 Dynamic Passwords

A technique which can help to get around this difficulty is to use *dynamic* passwords, i.e. passwords which automatically change according to some rule. Usually this involves storing in the computer not a password but a *function* for each user. When used as an authentication technique the system challenges the user with an argument and he has to respond by typing in a reply.

Here are some very trivial examples of functions:

```
f(x) = x +3
```

Here the result is the value of the (numerical) argument, plus three. For example if the system challenges with the number 67 then the correct password is 70. Here is another example.

```
f(x) = d * h    (d = day of month, h = actual hour)
```

The function in this case ignores the argument entirely and instead calculates a result which consists of multiplying the day of the month by the hour of the day at the time the authentication challenge is made. Suppose for example the user with this function logs in at 2 p.m. (= 14.00) on the 5th June and the system challenges him with the argument 67, then he responds with 70 (= 5 x 14).

Notice that the same argument can produce the same response in both examples, so that even if a hacker is looking over the user's shoulder when he logs in, or manages to eavesdrop on the challenge and the response on a communications line, he cannot deduce what the right response will be to any future challenge which the system makes if he attempts to impersonate the user.

The main limitation on dynamic passwords is that the function must be simple enough for the user to remember it and to calculate it. (In the case of a computer identifying itself to another, this need not be the case.)

6.5 Cognitive Passwords

Another approach is to use *cognitive* passwords. This is another form of challenge and response system, in which the challenges take the form of questions, to which the user has to supply the answers. For example the system might challenge with the question:

```
"What is the name of your maternal grandfather's dog?"
```

to which the user might respond with the answer:

```
"Fido"
```

The system can be supplied with a large selection of such questions and with the expected answers to them. Then each time the user attempts to log in, the system can choose one or more of the agreed questions at random. To make the problem more difficult for hackers the user might choose to supply the system with "incorrect" answers. Thus instead of listing the name of grandfather's dog as "Fido", which might be known to a hacker who knows the user, the careful user might have determined that the required answer is:

```
"Are you deaf?"
```

or something equally irrelevant. But the problem then becomes remembering the answers expected by the system!

6.6 Required Actions

Yet another mechanism which can be employed to make it difficult for hackers to impersonate registered users of a system is to monitor the first few commands which a user invokes after he has (apparently) successfully passed the normal authentication test. The real user has agreed with the system which commands he will first type in. If the system detects a different sequence then it assumes that the active user is a hacker.

This technique, which we shall call *required actions*, has one drawback. If the commands are actually carried out, this may lead to breaches of security, so that it is desirable that such commands are not actually really executed, but are simulated in a form which the hacker cannot detect.

7 Alternatives to Passwords

Not all systems use passwords or similar challenge-response systems to authenticate the identities of users. Other methods rely on various kinds of physical proofs.

7.1 Plastic Cards or Other Similar Objects

One common technique is to require the user to prove his identity by demonstrating that he possesses some kind of object. We are all familiar with the use of plastic cards which banks issue for this purpose. These are not without their problems. In particular theft or accidental loss of these cards can lead to security breaches. (The 10,000 combinations of PIN numbers – four digit numerical passwords – which are typically used in conjunction with these cards offer little security against the professional thief.)

7.2 Personal Characteristics

It might seem that the methods which rely on personal characteristics of the user to prove his identity are more reliable. We include in this category techniques such as voice recognition, finger print examination, blood analysis and the like. However, the equipment needed to carry out such tests is generally expensive. And in reality even these techniques are not foolproof. Voices can be captured on discs, users can be physically forced to provide their fingerprints or blood samples, and so on.

Furthermore, in my own experience with smartphones using fingerpreint analysis is far from reliable!

It seems that there is no absolutely foolproof method of authenticating the identities of users.

8 A Fundamental Weakness

What almost all systems have in common is that it is the *operating system* which carries out the authentication procedure. This may seem to be the obvious way to organise things, but in fact it is the root cause for a very fundamental security weakness in most systems.

In the first place a standard authentication procedure gives the hacker the important advantage that *he knows what he has to do* when he sets about penetrating the system! For example, he has to input the right password.

He also has a second advantage. A centralised system procedure for authenticating users implies that *there is a central repository of authentication information*, such as a password file. This provides the hacker with an ideal target. If he can crack this file then he can easily gain unrestricted access to all information in the system.

In view of these disadvantages of centralised authentication procedures carried out by the operating system, there is much to be said for the idea that *each user should be able to carry out his own authentication in whatever way he sees fit*. In other words individual users should have the freedom to check their own identities using their own authentication module. If such a module is a freely programmable user module, then each user can individually employ any of the methods which we have discussed in order to authenticate his own identity. If such a mechanism is in place hackers must not only discover the *content* of an authorisation procedure but also its form, and there is no central information to help him. He doesn't know whether he has to crack a simple password, a dynamic password, a cognitive password and/or conform to some required actions.

Of course not all users have the necessary skills to program an authentication module for themselves, but that should not distract us from the importance of this idea. Non-programmers could (in a world where this idea becomes normal) buy such modules from software houses and install them themselves. Such off-the shelf modules could be parameterised so that each user could tailor them for his or her own purposes. It would already be a help if operating systems provided a range of such parameterised modules.

Such a radical approach to authentication has not yet been implemented on conventional computer systems. A possible way of organising this would be for the operating system, after the user has identified himself by providing his user name, to create a process in which the user's security module is first executed. This would then determine whether the authentication is successful and, if not, destroy the process. But such an implementation would still have one weakness: there would still need to be a central file maintained by the operating system, which instead of holding passwords, would hold pointers to the security modules of all users. This itself would still be a good target for hackers.

As my colleagues and I have shown and demonstrated in practice in our experimental MONADS computer systems[15], even this problem can be avoided with the use of an unconventional computer architecture [23, 24]. In chapter 22 we shall see how it works in SPEEDOS.

[15] see the Monads website http://www.monads-security.org/persistent-protected-processes.html.

Chapter 5
Internal Security Threats and Weak Mechanisms

Once a user has gained entry to a computer system, having (rightly or wrongly) convinced the system that he is an authorised user, he typically works within a context in which he has privileges and access rights based on his identity. For example he may access his own files in a discretionary system or in a rule-based system he may access those which the rules allow him to. In other words he is subject to the access rights which are defined for him by the security policy in force at the computer installation where he is logged in.

However, in most practical multi-user systems there is no absolute guarantee that a user will actually be forced to remain in his intended access environment. There are several reasons for this. First, the security mechanisms of the system do not always function correctly. Second, they are often not powerful enough to implement the security model which is required. Third, the security model itself is often not adequate to meet the real security requirements of the users. Thus there is plenty of scope in most systems for expert authenticated users to break free from their individual security context and cause problems for other users.

We now consider some well-known ways a user may breach security or cause damage once he is in the system as an accredited user. We then briefly review why the mechanisms of a system are often not powerful enough to implement the security policy, and we consider why – in a discretionary system – the user may be dissatisfied with the system policy.

1 Threats at the Program Level

We first review some of the techniques which have provided practical threats to system security and have put systems at risk: bugs, viruses, worms and Trojan horses. These are relatively sophisticated forms of attack, since they rely on an ability either to write (or otherwise obtain) programs which do the damage or to take advantage of knowledge about computer systems.

1.1 Bugs

Bugs are errors in programs, which may have been accidentally or deliberately introduced. They have been used for example to allow hackers to gain illegal access to systems. Well known examples are errors which existed in the "finger" and "mail" programs of Unix, allowing hackers to gain access to systems, their password files etc. [2]. These programs ran with "superuser" rights (discussed later in the chapter), thus giving hackers the possibility of exercising all superuser privileges. In practice this meant that a hacker could at will read, modify or delete any file in the system.

1.2 Viruses

Viruses are pieces of program which can reproduce themselves. The dangerous functions of a virus are copied into various program files. When these programs are unwittingly executed by authorised users, the virus is further reproduced in more program files. Viruses often modify data files. Sometimes they are programmed to produce a spectacular effect (e.g. the destruction of files) at a particular time (such as at the turn of the millennium). They can also be programmed to do their damage when a particular event occurs, in which case they are sometimes called logical bombs.

1.3 Worms

In contrast with viruses, *worms* are complete programs. Worms deliberately reproduce themselves across networks, taking advantage of weaknesses in the security mechanisms of the computers in the network. They are generally a danger to the *availability* of a system, in that they are often designed to consume large amounts of system resources (e.g. processor time, main memory, disc space). The most famous worm was the "internet worm", which in a very short time led to the complete unavailability of 6000 computers in the USA.[16]

1.4 Trojan Horses

Trojan horses are programs which contain code designed to carry out hidden activities in addition to their intended tasks. A particularly dangerous example of a Trojan horse is a program which simulates the login procedure and thus can discover the passwords of unsuspecting users. Trojan horses are also often used to introduce viruses into systems.

[16] see https://en.wikipedia.org/wiki/Morris_worm

1.5 Direct Attacks

In addition to all these relatively sophisticated security threats there are of course the direct security threats, for example when a user of the system directly attempts to breach the confidentiality, the integrity or the availability of information belonging to other users, or to steal programs, in contravention of the security policy.

Security attacks, at whatever level of sophistication, are generally possible because of inadequacies in security mechanisms or security policies. We now consider some examples of such inadequacies, looking at security first as a compiler issue, then as a computer architecture responsibility and finally as an operating system problem.

2 Security as a Compiler Issue

A compiler is a program which translates another program, written in a high level language, into a series of low level instructions which are capable of being directly executed by a central processing unit (CPU) in the computer system. The program to be translated is usually called the *source* program, the translated form is the *object* program.

A compiler is specialised for compiling source programs written in a single high level language. Examples of well-known high level languages include Fortran, Cobol, Pascal, C, C++ and Java. There are however very many other programming languages which have been designed with a variety of aims and purposes.

Some programming languages emphasise the concept of *type*. This means that they insist that the variables used in programs have a fixed type which cannot be changed once the variable has come into existence. In this sense a type in a programming language defines a set of values which can be assigned to a variable of the type, together with a specific set of operations which can be used to manipulate these values.

For example most programming languages have a predefined type *integer*. Any variable of the type integer can only have a value which is a negative or positive whole number, or zero. (Thus integer variables cannot take on fractional or irrational values such as 1.32 or $^3/_4$ or $\sqrt{2}$ or π. For this purpose the programmer must use another type, for example the type *real*.) In practice the range of integers is either limited by formal definition in the programming language or by the fact that a computer word will only hold a finite number of values. For example many computers represent an integer value in a 32 bit word. With the most common way of representing negative integers this means that the largest integer is defined to be $+2^{31}-1$ and the largest negative integer -2^{31}. All integer values are then the whole numbers in the range bounded by these two limits.

The operations which are usually predefined on integers are + (addition), **-** (subtraction), **x** (multiplication), **mod** (modulo division) and **div** (division resulting in a whole number, ignoring the remainder). Notice that the normal division operation is not included, because it produces a result which may not be an integer! For example 21÷12 produces a result of 1.75, which is not an integer. Instead there are two division operations which always result in a whole number. The modulo division operation 21 mod 12 just gives the remainder part 9, while 21 div 12 gives an answer 1 (without the remainder 9). Integer operations which result in a value outside the defined range for integers may cause the computer to interrupt the program execution to indicate that this has happened.

In most high level programming languages the type integer is a *predefined type*, which means that it is built into the language and the compiler knows all about it. Other predefined types are often *real* (for fractional numbers), *boolean* (for the logical or truth values *true* and *false*), the type *character* (to express the letters of the alphabet, together with the decimal digits 0 to 9, punctuation marks and a few non-printable characters which need not concern us here), and possibly the type *string* (which allows individual characters to be strung together to form a piece of text).

Usually there are also some predefined types which allow other types to be combined together. (The *array* is an example of such a type, which defines a sequence of values of the same type. Another example is the *record* type, which allows several values of different types to be defined as a single entity.)

In most programming languages it is possible to extend the set of operations on a type by writing algorithms as *functions* (e.g. square root) which return a value of the desired type to the caller. Some languages (for example object-oriented programming languages) go a step further by allowing a programmer to define new types, building on the predefined types of the language and on other types which he has previously defined himself.

By now you are probably wondering what the concept of types in programming languages has to do with security. The answer is that it is sometimes claimed that the compiler for a *strongly typed* language can eradicate most security problems.

A strongly typed language is one which rigorously enforces type rules and does not allow the programmer to ignore them. Furthermore most of these rules can be checked by the compiler when it analyses a source program before producing the object program. Sometimes type rules are difficult to check at compile time, because the decision about whether a rule is being broken depends on the dynamic execution of the program. In such a case the compiler may insert some *run-time* checks into the object program itself.

To see how a language which is *not* strongly typed can easily lead to security breaches, consider again the array idea. In programming languages an item in an array is typically selected by naming the array and following this by the index value for the desired item in square brackets. To select say the entry holding the value for March in an array with twelve integer values called `Rainfall` we could write the expression `Rainfall[2]`[17]. The actual address of the location in the computer's memory is calculated by adding the index value (here `2`) to the address where the array begins. But suppose now that instead of writing `Rainfall[2]` we were to write something like `Rainfall[29435]` in our program, the result might be that the compiler creates instructions that add the integer value `29435` to the address which marks the beginning of the array, and that would produce an address which overshoots the end of the array by a long way. It may even be the case that the resulting address is outside the address range of our program. The effect of this could be that the program accesses a memory location in another program. Voilà! In an insecure computing environment we can read or modify information in another program.

How can this be prevented? The compiler could prevent it in this simple example by checking at compile time whether the index value `29435` is valid for the `Rainfall` array. Obviously it is not; only the values in the range 0 to 11 are valid, corresponding to the 12 months of the year. This is an example of a compile-time check.

But suppose that the program is a bit more complicated. Instead of nominating each index value explicitly, it might use an integer variable (say called *month*) and select an item using the expression `Rainfall[month]`. This is a quite normal way of writing programs, and in some programming languages it is no longer obvious at compile time whether the index variable is in the valid range. The compiler can usually only check this by inserting a run-time check which tests that the value held in the integer variable month is in the valid range before allowing it to be used as an index. Even this is not so easy if the length of the array itself is not known at compile time. A run-time check is still possible, but is more complicated. One problem with run-time checks is that they add more instructions to a program and so increase the time it takes to execute the program.

Another example of the same kind of danger occurs if a programming language allows addresses (usually called *pointers* in programming languages) to be manipulated explicitly. For example in a language which is not strongly typed (e.g. the widely used language C) it may be permitted to use integer opera-

[17] The first entry in an array is usually defined as entry 0, the second as 1, etc.

tions on pointers, so that arbitrary values can be assigned to or added to addresses, again allowing addresses to be produced which refer to the memory locations of another program.

In fact there are very many ways in which a program can be written which potentially attempts to access the memory locations of other programs. The important issue here is whether we should rely on compilers to ensure that this is not in practice possible. There are advocates of this view, who maintain that we should always use strongly typed languages, but the mainstream of computer science finds this approach unsatisfactory. Here are some of the reasons.

First, it assumes that the compiler itself is correct. If the compiler contains a bug (or can be classified as a Trojan horse) then there can be no certainty that the programs which it compiles will be correct.

Second, it assumes that only completely type secure languages can be used at a computer installation. This is a very risky proposition, because the proof that a reasonably complex programming language is completely type secure is very difficult.

Third, it means that many widely used programming languages, such as Fortran, Cobol, Pascal, C and C++, cannot be used.

Fourth, reliance on the correctness of a compiler creates enormous difficulties – both practical and with respect to security – for persons wishing to develop a new programming language or a new compiler for an existing programming language.

Fifth, it assumes that programming languages have complete control over all data in a system, which is generally not true, for example with respect to information on disc. (Most programming languages which are otherwise strongly typed allow file accesses which simply use the operating system or database facilities.)

For such reasons the view that security at the level of memory protection can be left to compilers is certainly inadequate for normal computer installations which use multiprogramming, allow any programming language to be used and allow users to develop compilers.

Nevertheless this is not an argument against the use of strongly typed programming languages. On the contrary these have many benefits, not least of which is the advantage that a compiler can find many errors in a source program before the object program is even produced.

3 Security as an Architectural Issue

Traditionally not the compiler but the computer architecture (the environment which the computer hardware provides for the execution of programs) takes responsibility for ensuring that programs executing in the main (or virtual) memory cannot interfere with each other. Most computer architectures achieve this in a fairly satisfactory way, provided that one accepts the extremely simple protection paradigms that they enforce. In particular these paradigms tend to be good at enforcing strict protection in the sense of complete separation.

However the other side of the coin of protection is *sharing*. It is not a particularly difficult challenge to isolate programs so completely that they cannot communicate with each other or easily share data and/or code. Basically this can be achieved by providing completely separate and non-overlapping contexts. (We shall see in later chapters how this is achieved in practice.) But when it comes to the challenge of allowing subjects to cooperate with each other in secure ways, most computer architectures fail miserably.

An analogous situation would be to imagine that we avoid burglaries by building houses which have no doors and no windows. This is all very well for keeping burglars out, but it isn't very good if you want a friend to visit you! In fact it has lots of other obvious disadvantages too! In the same way we shall see that conventional computer architectures have lots of disadvantages when it comes to sharing, and that this is one of the main reasons why the software systems which have to use these basic mechanisms (i.e. the operating system, the file system, the database system, etc.) are in practice excessively complex, for these have been given the job of making the sharing of data and programs possible. In doing so they have been forced to invent mechanisms which are quite unnatural and cumbersome. It is therefore not surprising that these software mechanisms are rather weak when it comes to guaranteeing security.

4 Security as an Operating System Issue

Operating systems usually have sole control of the hardware and of the architectural mechanisms of the computer. With the aid of these low level controls operating systems are traditionally responsible for providing the kinds of higher level mechanisms which are needed for implementing security policies.

One such higher level mechanism normally provided by the operating system is the authentication of users when they log into the system. We have already observed that this – ironically – can be regarded as a security weakness in that it implies the existence of a central authentication mechanism and of centrally held authentication information, thus helping the hacker both to know what he has to do to penetrate a system and where there is useful information to help him achieve this.

The operating system also carries out other important security activities, such as the allocation of space in the main and secondary memories and ensuring (with the help of basic architectural mechanisms) that programs cannot violate others' space. This implies for example that it is also the operating system's responsibility to ensure that memory is not allocated to a program until the previous contents of all memory locations have been cleared. In fact operating systems rarely do this because of the overheads involved – but if it is to be done in a highly secure system, then the operating system is the obviously right place.

Other security activities, such as deciding whether a user may start a job or claim particular resources, etc., are likewise policy decisions which are too complex for the hardware to carry out. They are the responsibility of the operating system. It is arguably the failure of operating systems to carry out such tasks effectively that allows worms to chew up system resources and thereby make these unavailable for legitimate uses.

5 Privileged Mode

It is clear that operating systems play an important role in maintaining computer security. Because the operating system is the traditional vehicle for implementing security policies, many computers support a privileged mode – sometimes called kernel mode, supervisor mode, executive mode or similar – which gives it greater powers than the rest of the system. This means that when the operating system itself, rather than a user process, is actually executing instructions, certain hardware protection checks are relaxed.

When the computer is executing in privileged mode, the main memory protection checks, which are otherwise carried out by the hardware to prevent programs from interfering with each other, are usually turned off.

Privileged mode usually also brings with it the right to execute certain sensitive instructions. These are typically instructions which are needed by the operating system to maintain its control over user processes and to guarantee the security of the system. One such instruction which many computers allow only to be executed in privileged mode is the instruction which initiates input-output activity, such as the writing of data to or reading of data from discs. If any program could use this instruction in an uncontrolled way, one effect would be that any user could access the files of others.

While it is clear that there is a need for certain operating system activities to be privileged, this feature can also be a serious source of weakness for the security of the system. There are at least two reasons for this.

First, many operating systems are very large and complex. Many of the activities which they carry out do not need to be privileged, but in practice these activities are often also executed in privileged mode. This can be a source of misuse. Trojan horses in the operating system can easily take advantage of their privileged status to breach security.

Second, the implicit assumption that the operating system has a right to be privileged is itself a highly questionable proposition. Suppose for example that the secret service organisation of some small nation such as Australia, which does not manufacture computers itself, buys a computer and uses it to hold highly sensitive information. Why should the operating system of some foreign computer manufacturer – which might contain a Trojan horse – have the right to access its memory space? Or, why should an operating system be able to read the information held in a banking system? How can the bank then guarantee the privacy of its customers' information?

In the final analysis an operating system has to have certain privileges which allow it to protect users from each other and from unauthorised users. There must be a software module which can read from and write to all the blocks of a disc, in order to manage disc space in an orderly way. The spooler must be in a position to print user files, and so on.

For such reasons it seems unlikely that we shall ever reach a situation in which the operating system has no special privileges. But that does not have to imply that *all* parts of the operating system have to have *all* the privileges, including unlimited access to all the information belonging to users.

6 Security Kernels

In the 1970s the idea became popular that the security sensitive parts of an operating system should be gathered together into a so called *security kernel*. This should be open to public inspection and should be small enough and clear enough that its correctness is self-evident or formally provable (see for example [25, 26, 27, 28, 29, 30, 31]).

The idea of security kernels has not been put into practice in widely used systems, because it is often difficult to isolate security functions from other operating system functions; consequently security kernels tend not to be so small as one would hope. This in turn means that it is not a simple matter to prove a security kernel correct. Nevertheless the principal idea, that privilege should be restricted to a small part of the operating system which is provably correct and open to user inspection (to demonstrate that it does not contain Trojan horses) obviously provides an important concept, which could play a significant role in improving the security of operating systems.

Finally it is worth remembering that a provably secure operating system is only in practice useful if this really is the operating system which is actually put to use! In other words, there is a security risk that the system which you think you are using has been penetrated after having been proved correct. This risk has therefore led to an issue known as *secure booting* of systems. What this means is that special measures are taken to guarantee that the operating system which you think you are initialising on the computer is the one that you actually intend to initialise. We shall not consider this rather technical issue further. Interested readers can refer for example to a paper written by two colleagues and myself [32, pp. 106-119].

7 Inadequate Security Policies

Weaknesses in computer security are not always the result of inadequate security *mechanisms*. In many cases the security policies in force at computer installations are themselves totally inadequate. By way of example we now consider some typical weaknesses in security policies which often appear in the design of general purpose operating systems with discretionary access controls, especially at the file system level.

7.1 The Superuser Role

There is a general assumption, even in discretionary systems, that there should be at least one highly privileged user, e.g. the "superuser" in Unix. This role, which in some systems allows a user to take on the security privileges of any other user at will, appears to be justified in order that certain administrative tasks can be carried out. Examples of such apparently essential tasks include the introduction of new users, the resetting of a password which a normal user has forgotten, etc. However the proposition that such activities require a superuser to be able to take on all the rights of other users is highly questionable.

The superuser role can easily be abused. Such abuse may take a direct form if the person filling this role has less than altruistic intentions. Indeed, if an organisation or group wishes to acquire information over a long period about another organisation, it is an obvious strategy to get one of its spies placed into the superuser role of the rival organisation!

The superuser role can be dangerous in at least two further senses. First, this role is often so essential to the running of the system that several people are entrusted with the superuser password, for example to ensure normal service when the main superuser becomes ill. The more persons there are involved, the less secure the system!

Second, if only the superuser can carry out certain functions or execute certain programs which other users need to use, or if it is much easier to carry out such functions in the role of the superuser, then there will be a tendency to give many users the superuser password. This happens every day in many computer installations around the world.

The superuser role has in fact proved to be one of the greatest security risks in discretionary systems.

7.2 Simplistic Access Control Policies

The basic file system, the largest component of most operating systems, organises user data and programs as files. It is responsible *inter alia* for checking the rights of subjects attempting to access these files.

The inadequacy of the policies underlying the operation of file systems is itself a major weakness in most discretionary systems. The problem centres on a failure to provide a proper implementation of Lampson's Access Matrix for individual subjects and objects. For example in the VMS operating system for DEC's VAX computers access to a file was organised on the basis of four user categories:

```
[S:] System administrator (system)
[O:] Owner of the file (owner)
[G:] Group of the owner (group)
[W:] All other users (world)
```

For these categories access rights for four operations could be specified:

```
[R:] Read
[W:] Write
[E:] Execute
[D:] Delete
```

This means that for any file 4 x 4 possible combinations of access rights exist. These are represented by 16 bits in a matrix (where 0 = no right; 1 = right present).

In Unix the superuser has all rights, so that a category equivalent to the VMS system administrator category is not necessary. Furthermore, only the owner of a file has the (implicit) right to delete a file. Thus in Unix one column and one row of the access matrix used in VMS are redundant. (In fact the situation is a little more complicated than this, but it doesn't materially alter our point.)

In the Siemens BS2000 operating system, to take a rather older example, there were only 2 x 2 possible combinations of rights. It is only possible to distinguish between access for world and access for the owner, and the actual access rights distinguish only between read access and write access. (Further access controls could however be achieved through the use of password facilities.)

There are some superficial problems with such schemes (e.g. users may need to belong to more than one group). But the fundamental problem is the coarseness of granularity of the access rights. Suppose for example that a user wishes to give another user (not a member of his group) access to a file, then he has no alternative but to provide either all users with the same rights or to make the intended user a member of his group.

In both cases unnecessary privileges have to be given. Even if these are granted only for a short time (giving the user time to make a copy of the file, for example) the system is still more insecure than would be necessary if the policy allowed a full implementation of Lampson's Access Matrix.

Thirty years after Lampson published the idea behind his Access Matrix paper [9], he published a paper which argues that

> "We don't have "real" security that guarantees to stop bad things from happening, and the main reason is that people don't buy it. They don't buy it because the danger is small, and because security is a pain",

and

> "when security flaws cause serious damage, buyers change their priorities and systems become more secure, but unless there's a catastrophe, these changes are slow. Short of that, the best we can do is to drastically simplify the parts of systems that have to do with security:
>
> • Users need to have at most three categories for authorization: me, my group or company, and the world.
>
> • ..." [33].

In the second volume of this book, which describes the SPEEDOS operating system design in some detail, I describe how Lampson's matrix (and many more security features) can be put into practice in a relatively straightforward manner, which users will hopefully not see as a "pain".

7.3 The Authenticity of Logged in Users

A basic assumption of most systems is that a logged in user is really who he claims to be, since he has succeeded in passing through the operating system's initial authentication mechanism. However, there are many reasons why this may not be the case. For example, the user may in fact be a hacker who has successfully deceived the initial authentication procedure. Or he may be somebody taking advantage of the real user's absence from the terminal (e.g. when he has left himself logged in while going for coffee), or he may even be a criminal who has overpowered the genuine user.

For these reasons further checks to re-authenticate a user may be appropriate in an environment which aims at high security. Such checks might be periodic and/or they might take the form of challenges when the user seeks access to a secure resource. The form of the challenge might, but of course need not, be that of the initial authentication procedure. However, with most conventional operating systems such re-authentication procedures are rarely possible in any form.

8 Gathering the Evidence

Given that security mechanisms are generally not strong enough to guarantee the perfect execution of security policies, it becomes important to monitor security sensitive events in order to help discover security breaches, to help track down users responsible for breaches, to help establish the extent of the damage and determine what recovery procedures are appropriate.

Again low priority – if any – is given to these monitoring activities in many security policies, especially those found in discretionary systems. If the system doesn't organise this, it is very difficult for users to learn about security breaches affecting their files.

9 Too Many Cooks

There is an old saying that too many cooks spoil the broth. This is certainly the case when considering security in computer systems. Typically some basic mechanisms are built into the computer architecture. Then the operating system adds its own additional mechanisms. The basic file system adds some more. The database system may add yet other security mechanisms. The network systems may also add more security mechanisms. Sometimes proprietary security software packages are then added to this usually complicated hotch-potch of mechanisms.

If these multiple sets of mechanisms are not well designed to interact with each other correctly, security gaps are often created which the clever hacker can use by playing them off one against the other. This is one of the major forms of weakness in a security system.

It is our contention that what is required is not an ever increasing number of mechanisms to patch up the gaps left by others, but instead a small number of very simple basic mechanisms which have universal application in all levels of the system. These should preferably be publicly known and well understood by all. In the last analysis a security mechanism which relies on secrecy offers no security, because human nature finds it too hard to keep secrets for ever!

In the remainder of this volume we examine the weaknesses of existing system security mechanisms at the hardware, architectural and operating system levels, particularly emphasising how they can harmonise with each other, and we suggest new mechanisms which are intended to be simple, easy to understand and universally applicable through all levels of a system. In defining these it has been our intention that they should be flexible enough to allow the implementation of all the different kinds of security policies which we have so far discussed, and indeed it is our aim that this should be possible in a single system. In other words we aim to produce a relatively small set of mechanisms which will not only allow the implementation of many different policies and styles of security systems, but to show that these can be implemented alongside each other in a single computer system.

Part 2
Basic Computer Architecture and Operating System Principles

Chapter 6
A Brief Introduction to Computer Architecture

This chapter briefly introduces some basic features of computer architecture with the aim of providing non-specialist readers with enough background information to understand the remaining chapters of the book.

1 The Structure of a Modern Computer

Modern computers, structured according to the principles laid down by John von Neumann [34][18], consist of the following main components: at least one *central processing unit* (CPU), a *main memory* (often called *RAM*) and some *input-output* (I/O) devices (see Figure 6.1). The CPU carries out the actual calculations specified in a program by executing its instructions. The main memory stores the program and its data during the computation. The I/O devices are used for long-term storage of information (hard discs, flash memory sticks, CDs, DVDs, etc.) and to provide a means of communication between the computer and the outside world (monitors, keyboards, printers, etc.).

In a very loose way we can think of the CPU as the "brain" of the computer, the main memory as its memory and the I/O devices as its sense organs for communicating with the outside world. The different parts of the computer in a classical von Neumann architecture communicate with each other via a *bus*, which is a set of lines (wires) along which information flows, as is shown in Figure 6.1. This can be thought of as the nerve system of the computer.

The main memory consists primarily of a sequence *bits* (binary digits), each of which can hold the value 0 or 1. The CPU *reads* and *writes* bits in groups of 32 or 64 bits, called *words*. It can interpret these as various kinds of numbers (based on binary arithmetic) or as groups of *bytes* (each consisting of 8

[18] See https://en.wikipedia.org/wiki/Von_Neumann_architecture

bits) which can be interpreted as alphabetic or numeric characters, punctuation marks and other symbols.

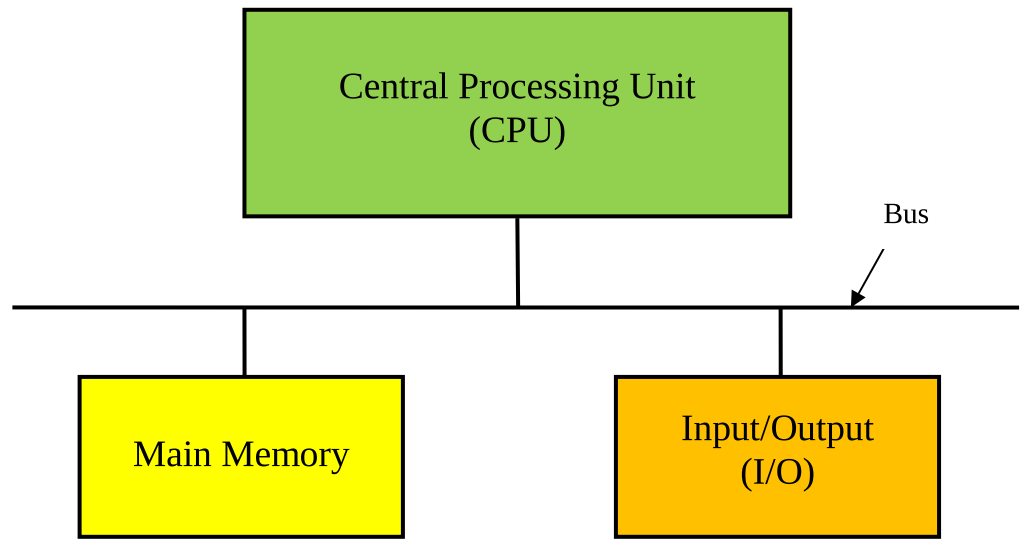


Figure 6.1: A "von Neumann" Computer

Each word in the memory has a unique numeric address (corresponding to its position in the memory, starting at the address 0). This is used by the CPU to identify the intended word when reading from or writing to the main memory.

The use of the bus is normally controlled by the CPU. Typically it has three kinds of lines: address lines, data lines and control lines. The address lines are used to pass memory addresses from the CPU to the main memory, indicating which memory location is involved in a read or write operation. The data lines are used for the actual transfer of data. The control lines indicate what kind of operation is required (e.g. a write operation).

Accessing I/O devices can be, and often is, treated in a similar way to memory accesses, with the address lines indicating not a main memory address but a device buffer address. Not all computers treat I/O devices in quite this way, but the differences are not important in this context.

2 Main Memory

A typical main memory has a number of important properties, some advantageous and some not so advantageous.

- Main memory *access times* (i.e. the time to read or write a word of main memory) are very fast. This enables the CPU to work at high speed.
- Information in the main memory can be accessed *randomly*, which means that you don't have to work through the memory item by item until you can

access the item you are seeking[19]. This random access characteristic of main memory is extremely important, because the instructions and data items needed for a computation are not stored in a single sequence.

- Main memory is a very expensive form of memory, much more expensive per word of memory than a hard disc or a tape or a CD or DVD, etc.
- A further not so happy characteristic of main memory is that it is normally not *persistent*. This means that, like most other electronic memory devices, the information held in the memory is lost if the power is turned off. This also contrasts with hard discs and other magnetic media.

It is necessary to keep track of where in the main memory particular programs and data items are stored. This means that the main memory itself not only holds program instructions, numerical values and characters and other data items, but also memory addresses. Thus a particular word in the main memory might contain the address of another word. For the present we shall assume that such an address is simply a main memory address; however, we shall later encounter other possibilities. Addresses are normally stored in separate 32 or 64 bit words.

Since the main memory holds the currently active data and program instructions in use by the central processing unit it is important from a security viewpoint and is therefore supplied with basic protection mechanisms which will be discussed in more detail in later chapters.

3 The Central Processing Unit

The CPU is the unit which actually carries out instructions, and it therefore needs access to the information held in the main memory. When a word is read from the main memory into the CPU, there must be a small memory in the CPU to receive it. Similarly before a write operation the word to be written back into the main memory must be held in a small memory in the CPU. Usually the same 32 or 64 bit memory in the CPU, known as a *memory buffer register* (MBR for short), is used for both purposes and is connected to the data lines of the bus. Similarly the main memory address to be used for the read or write operation is held in a *memory address register* (MAR), which is connected to the address lines of the bus.

Somewhat simplified, the CPU of a modern computer typically consists of two main parts, an *arithmetic-logic unit* (ALU) and a *control unit* (Figure 6.2).

[19] A device which cannot be directly accessed at any point with the same ease and speed, but is only really fast when items are accessed in sequence, such as a magnetic tape, is called a *sequential access* device. You can understand this difference by comparing the time difference between searching for a song in the middle of a music cassette tape or in the middle of a music CD.

3.1 The Arithmetic-Logic Unit

The ALU contains the logic circuits for performing such operations as additions, multiplications, comparisons, logical operations, etc. There is also a set of "registers" associated with the ALU. An ALU register is a memory location which is actually held with the ALU rather than as part of the main memory of the computer. Most ALU registers contain one word of information, i.e. their size corresponds to the size of a word in the main memory. We have already seen two special examples, the MAR and the MBR.

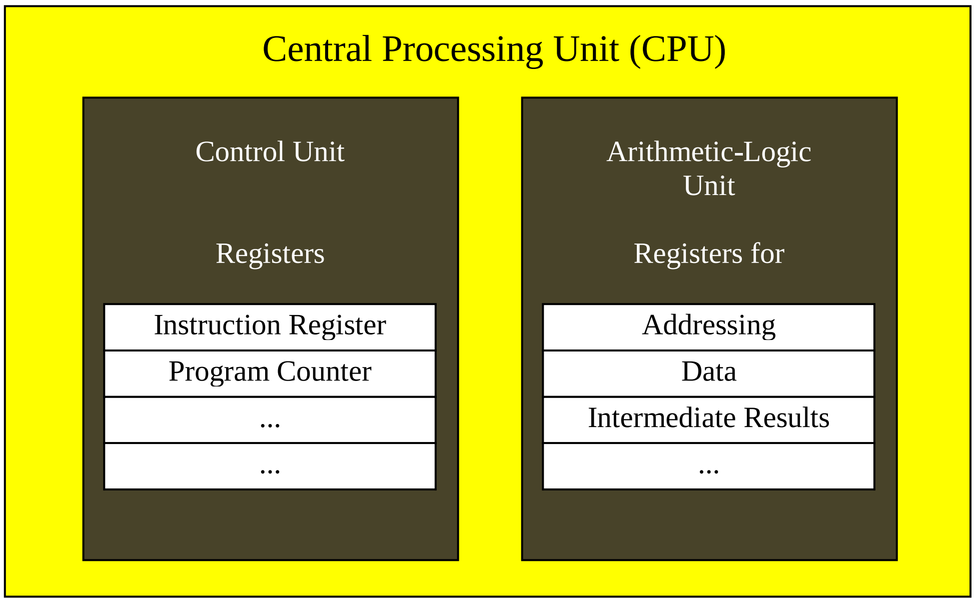


Figure 6.2: The Central Processing Unit of a Computer

An important reason for having ALU registers is that they can be built from components which have much faster access times than equivalent chips used for implementing main memory. But like main memory chips they are not persistent (i.e. they too lose their values when the power is off). ALU registers are considerably more expensive than main memory chips.

The main reason for having ALU registers is that they can store a small amount of information which is needed for immediate use in executing instructions in the CPU, and their access times are fast enough to allow the CPU to execute at full speed. But by using the slower chips for the main memory it is possible to build large main memories at an affordable price.

3.2 ALU Instructions

The ALU registers are used mainly to hold *operands* for instructions. These are the values on which the instructions directly operate. An instruction usually has an operation field and several operand fields, as is illustrated in Figure 6.3, which shows typical instruction formats for a RISC (reduced instruction set computer). The operation type tells the CPU what kind of operation should be executed (e.g. ADD). The remaining fields specify the numbers of the ALU registers which hold the operands for the instruction (or hold an immediate value). For example if the encoding for an ADD instruction is 37, then an instruction in the form <<37, 3, 8, 5>> (using the first structure in the diagram) tells the CPU to add the content of register 8 to the content of register 5 and to store the result in register 3.

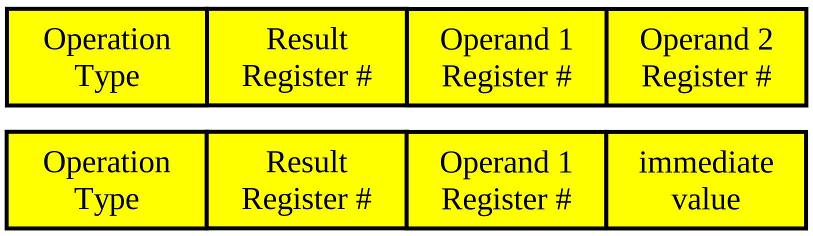


Figure 6.3: Typical RISC ALU Instruction Formats

3.3 Load and Store Instructions

The ALU registers which serve as data operands must from time to time be loaded with values from locations in the main memory. Similarly the results of instructions held in registers must sometimes be stored back into locations of the main memory. The usual way to achieve this on RISC computers is to have separate *load* and *store* instructions. These instructions themselves require operands indicating which register is to be loaded (or stored) and which main memory address is involved. Typical RISC load and store instruction formats are illustrated in Figure 6.4.

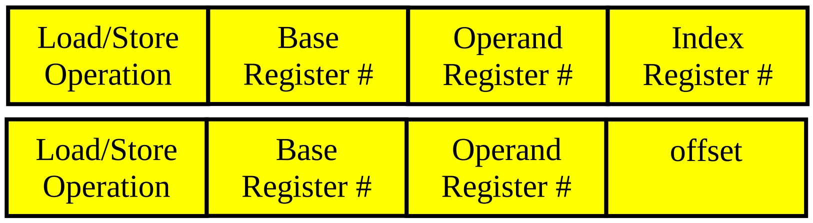


Figure 6.4: Typical RISC Load/Store Instruction Formats

The first format specifies *base* and *index* registers, e.g. for working through a list in the main memory. The base register holds a main memory address which typically refers to the start of a structure such as an array, and the index register holds a variable offset from that point, indicating how far into the array the relevant item is. Using the second format, a literal value (a fixed value which appears in the instruction itself) is added to a base register.

3.4 The Control Unit

As well as having instructions which directly perform arithmetic and logical operations on values in registers, and having instructions for loading and storing registers using main memory accesses, a computer needs a third kind of instruction in order to execute algorithms. To understand this we must now take a look at the Control Unit component of the CPU.

From Figure 6.2 we see that the Control Unit also has at least two registers: the *instruction register* (IR) and the *program counter* (PC). These two registers are used to control the order in which the instructions in a program are executed.

The IR holds a copy of the *content* of the instruction which is currently being executed in the ALU. When an instruction is loaded into the IR the control unit can decode and analyse it, sending appropriate control signals to the ALU to tell it for example which kind of operation is to be carried out and which registers are to be used.

3.5 The Fetch-Execute Cycle

Before an instruction can be decoded it must be fetched from the main memory into IR (via the memory buffer register MBR). Thus there are two basic phases to the execution of an instruction: fetch the instruction, then execute it. This is known as the *fetch-execute cycle*. In order to fetch the instruction the control unit must know where it is located in the main memory. This is where the program counter register PC comes into the picture: it contains the *address* of the next instruction to be executed. While a program is being executed it is the job of the control unit to keep updating this register.

3.6 Program Execution

In the normal case a program's instructions are executed sequentially, i.e. one after the other in the sequence in which they appear in the main memory. Thus as arithmetic and logical instructions or load and store instructions are being executed, the next instruction is found simply by adding a small number, corresponding to the length of the current instruction, to the address currently held in the PC register.

But of course programs do not simply execute in a straight line from start to finish. They contain decisions and they contain repeated sections, both of which require the control unit to have the ability to jump to an instruction which is not the one physically following that just executed. For this purpose computers usually have two kinds of instructions, known as *unconditional jumps* and *conditional jumps*. An unconditional jump instruction contains the address of the next instruction as an operand, which is then loaded into the PC register. A conditional jump also contains a condition to be tested (e.g. by comparing the values in two registers); the result of the test then determines whether the next instruction is executed or whether the destination address provided in the instruction is loaded into the PC register.

3.7 Routine Calls

It is frequently useful in a computer program to include *routines* (sometimes called procedures, functions, subroutines or methods) which carry out a useful subtask or calculation that may be needed several times in the program. The same routine can be *invoked* from different points in a program; when the instructions in the routine have been executed it must *return* to the instruction following the instruction from which it was called. Most computers provide some form of *call* instruction to assist this task. As a minimum the call instruction jumps (as with an unconditional jump instruction) to the first instruction in the subroutine, and at the same time it stores for later use the address of the instruction following the call instruction, called the *return address*.

As was already indicated, the instructions themselves are fetched from the main memory. Recalling the interface which we described between the CPU and the main memory, we can now see how this happens.

To fetch an instruction the control unit copies the value of PC (the program counter) into MAR (the memory address register) and sends a *read* signal to memory. When the read is complete, the next instruction has appeared in MBR (the memory buffer register). This is then copied by the control unit into IR, leaving MAR and MBR free to be used for some other purpose (e.g. for a load or store operation).

4 Cache Memories

Before examining the third main component of von Neumann computers (the input-output subsystem), it is appropriate to consider a technique used on modern computers for optimizing accesses to memory. It provides an effective solution for the problem which we mentioned earlier, namely that a CPU can execute instructions considerably faster than the instructions and their operands can be fetched from the main memory. Without cache memories the CPU would only be able to execute at the rate at which instructions and data could be fetched from main memory.

The idea, which its inventor, Maurice Wilkes, originally called a "slave memory" when he proposed it in 1965 [35], is now generally known as a *cache* memory. The word "cache" comes from the French word *cacher*, which means "to hide". This name emphasizes that a cache memory, unlike the main memory or the CPU-registers, is hidden from the assembler programmer's view. It is (almost) purely a hardware optimization.

The idea is that a memory unit considerably faster and smaller than the main memory is placed close to the CPU (e.g. between the CPU and the bus). A cache memory holds instructions and data which have recently been used (and may be needed again). Once an item has been fetched from the main memory a copy of it is stored into the cache. Thereafter it can be fetched from the cache and access to the main memory becomes necessary in fewer cases.

Because the cache is significantly smaller than the main memory it obviously cannot hold a copy of all items in the main memory, so when it becomes full a decision has to be made which item should be replaced to make space for the new item. The strategy usually used is called "least recently used" (LRU). The item selected for replacement is the item which has not been used for the longest time. The success of this idea depends on the fact that items from the main memory are frequently used several times over a short period of time and also that neighbouring items in the main memory are often used together. It is remarkable how successful this strategy is in practice. Modern computers often achieve a "hit rate" of over 95 %, i.e. more than 95 % of all memory accesses can be satisfied by the cache, so that actual references to main memory are relatively infrequent. The result is that computations can proceed at very much faster rates than in an equivalent system without a cache.

We need not concern ourselves here with the technical aspects of cache memory implementation, but in the next chapter we shall introduce the idea of virtual memory and virtual addresses. At that point it will be necessary to return to the question of caches to discuss some issues which then arise.

5 The Input/Output Subsystem

The third main part of a modern computer is the input-output (I/O) subsystem. This serves as the interface between the computer and the outside world. In modern computers information is typically input into the computer via keyboards (e.g. attached to terminals, or built into laptops, etc.) and more recently via touch screens. Information is provided from the computer as printer output or via monitor displays. There are other kinds of I/O devices which can be attached to computers, such as scanners, graph plotters, fax devices, analogue-to-digital and digital-to-analogue converters, and special equipment and instruments for real-time systems. And of course there is usually an interface to connect to a local area network (LAN) and/or the Internet.

The magnetic media memory devices, such as hard discs, CDs, DVDs, magnetic tapes and on earlier computers magnetic drums, are also usually considered as I/O devices; the way they are controlled and accessed from the computer is in fact very similar to the way other I/O devices are handled. But logically they have a role more analogous to the main memory, in that they are memory storage devices which, in contrast say to printers, keyboards and screens, cannot directly be read or written from the outside world. In this sense the analogy with the human memory is much more appropriate. At this stage we shall include the magnetic memory devices in our discussion of the I/O subsystem, but later we shall draw a clear distinction between their physical affinity to external I/O devices and their logical affinity to the main memory.

The I/O subsystem plays a major role in so-called "mainframe" systems, the large computers which are used mostly by companies and by government departments, mainly because such organisations handle large amounts of data. Consequently in mainframe systems the I/O subsystem can be rather complicated and is usually an expensive part of the computer. In contrast the laptops, personal computers, workstations and hand-held devices such as smartphones, which many people now own, have a very much simplified I/O subsystem.

This difference is not important for our purposes. We do not need to discuss I/O devices and their controllers in great detail, because the important thing from a security viewpoint is not really how these function individually, but how their use is controlled from within the computer. Here the most important point is that it is usual for the operating system to have complete control over the transmission of data between the computer and the I/O devices. Let us now take a look at how this works.

6 Overlapping I/O and CPU Operations

When talking about the I/O devices it is important to realize that there is an enormous speed difference between the CPU and main memory on the one hand and I/O devices on the other. Even with the fastest I/O devices, magnetic discs and even solid state drives (SSDs), we are talking about a significant speed difference in the time it takes to access information. An implication of this speed difference is that if the CPU were to stop processing and wait for each I/O operation to complete, then it would spend almost all its time waiting. For this reason I/O operations are carried out *in parallel with* CPU operations. What normally happens is that the operating system, after starting an I/O operation for a process, puts the process which initiated it into a waiting state until the operation is completed and then makes a *process switch* to a different process, which can carry on using the CPU while the I/O operation for the other process is taking place.

When the I/O operation comes to completion the I/O device (or its controller) informs the CPU, so that the program which requested the I/O operation may resume its work. In nearly all systems this is achieved by the I/O device or controller causing an *interrupt*. After the CPU has been interrupted by an I/O device completion signal, the process scheduler (that part of the operating system which decides when which processes can use the CPU) can allow the process which instigated the I/O operation to continue executing.

6.1 Kernel Calls and Interrupts

All of this is controlled by the operating system and remains invisible to the application program, which treats an instruction for carrying out an I/O operation just like a normal CPU instruction.

But in reality the I/O instruction which appears in the application program is not a normal instruction at all. It is a special instruction for invoking the services of a *device driver* in the operating system. Whenever an application program wishes to invoke any operating system service (not just carry out an I/O operation) it executes an instruction which we shall call a "kernel call" (It is sometimes called a system call, supervisor call, or executive call). Such instructions have a special operation type which causes the CPU to activate a designated operating system routine, usually by making use of the interrupt mechanism.

6.2 Why Application Programs Do Not Have Direct Access to I/O Devices

There are several reasons why normal application programs do not contain instructions that directly activate I/O devices. One of these is that the actual control of I/O devices at the hardware level is usually fairly complex. For this reason the operating system (or the device manufacturer) provides device drivers, which are software modules capable of coping with this complexity but at the same time providing a much simpler interface to application programs. Another reason is that direct access by application processes to I/O devices could lead to security breaches. This is obviously the case for shared devices such as hard discs containing information belonging to different users. If any user could simply access any part of any disc at will, then confidentiality, integrity and availability of information belonging to others would all be at risk.

But also in the case of normally unshared devices, such as a printer, the operating system has to maintain control. Without it, users could write to the printer at any time, interfering with each other's output. Furthermore if the use of devices such as printers were not controlled, then solving the confinement problem, a security problem mentioned in Chapter 3, would become more difficult.

Consequently when an application program wishes to have an I/O operation carried out it always does this by invoking the operating system, using a kernel call. The appropriate operating system service routine carries out checks which determine whether the application process (or the user on whose behalf it is executing) is permitted to perform the required operation (e.g. use the printer) and if so whether it is appropriate at this time (e.g. whether it is currently allocated for use by another application process).

Having performed all the necessary checks and adjustments necessary, the device driver eventually activates the appropriate I/O device. The form of an actual I/O instruction to achieve this at the hardware level varies considerably between different computers. One difference is apparent in the way the actual device required is addressed by the CPU. In some cases devices use pseudo-"main memory" addresses on the address bus to address the different devices.

This method, known as *memory-mapped* I/O, makes sense in that information is transferred to devices via hardware registers or buffers, which can be viewed as memory that can be read and written. Other systems use a separate set of numbers, transmitted across the control lines of the bus, to signify which device is intended and which operation is required.

These different approaches are often reflected in the form of the hardware instruction used to activate I/O devices at the hardware level. In one case I/O instructions appear as normal read and write operations on defined (device buffer) addresses. In the other case there is a special "start I/O" instruction. There is a corresponding difference in the way control is exercised over the use of such instructions. If the I/O instruction reflects memory mapped I/O by using special addresses, then the address checking hardware must recognize that an application process is attempting to use an address which it is not allowed to use. On the other hand if there is a special "start I/O" instruction then this has to be classified as a privileged instruction. In either case an illegitimate attempt to directly use an I/O device will result in an interrupt, and the operating system interrupt routine can then take appropriate action (e.g. by forcibly ending the application process).

7 Magnetic Media Devices

The earliest computers were built primarily to carry out calculations. There was little or no data stored permanently in the computers, there was no Internet and nobody was producing proprietary software which needed to be protected. Consequently there was no serious security problem. The primary aim of computer designers was to build computers capable of carrying out ever faster scientific and mathematical computations.

However, it was not long before computers began to be equipped with devices which were capable of storing information and programs on a long term basis. Magnetic drum stores and then magnetic tape devices were developed. The drums had only a small data storage capacity and the tapes were very slow if they were not used sequentially.

With the invention of magnetic discs it became possible not only to store large amounts of data internally in the computer, but also to access such information rapidly without a sequential search.

There are many differences between magnetic discs and main memory. First, access to main memory is many orders of magnitude faster than access to even the fastest discs, so that disc memory cannot be regarded as an alternative to main memory. (This point is not affected by the use of cache memory.) Second, disc memory is much cheaper per word than main memory, so to store bulk information on disc is much cheaper. Third, unlike conventional main memory, disc memory is persistent (i.e. the information is not lost when the power is turned off).

These differences, together with the fact that magnetic media memories have mechanical components and are physically much more like I/O devices than they are like main memory, led to what initially appeared to be a natural division of information in the computer.

Since the late 1950s it seemed natural to use magnetic media as long term storage devices and main memories as the vehicle for holding programs and their data temporarily, while computations are being carried out. This in turn led to what appeared to be an equally natural division in operating systems. One part of the operating system is responsible for the execution of programs, and this has control of the main memory, while another quite different (and usually rather larger) part, known as the *file system*, manages the long term storage of bulk information and has control of the magnetic media.

However, this clear and simple division did not survive for very long. Because of the high costs of main memory in the 1960s, the idea of *virtual memory* was introduced. This is a technique for allowing more and/or larger programs to execute at the same time than would be possible in the main memory alone. However, the division between computational (now virtual) memory and file memory did. In the next chapter we shall take up this story, as it is very important for the implementation of security measures in computers.

Chapter 7
Virtual Memory

The two primary kinds of memory unit relevant to a discussion of virtual are the *main memory* (also known as RAM) and *magnetic memory* devices (e.g. internal and external hard discs). These have a number of different properties.

- Modern main memory is built from logic circuits which have very fast access times, because they do not involve moving parts. Hard discs, by contrast, are very much slower because the information is usually stored on rotating surfaces and usually also involves reading and writing heads which have to be physically positioned to the right place.[20]
- Main memory is very much more expensive per byte of storage capacity than magnetic devices.
- Magnetic storage devices are persistent, which means that the information stored on them does not disappear when the power is turned off. On the other hand main memory is usually not persistent.

These differences have strongly affected the way they are used in computer systems.

1 Memory in Early Computer Systems

In early systems of the late 1950s and 1960s vintage the main memory was used as a *computational memory*, i.e. the memory in which data and programs for active processes[21] were temporarily stored during actual program execution. On the other hand the magnetic media devices were used as a file memory, i.e. a memory in which information and programs could be stored on a long term basis. This difference is illustrated schematically (but not to scale) in Figure 7.1.

---

[20] Although modern SSD devices are electronic and do not rotate, they fall into the "disc" category because they are slower than RAM and are organised to be used like discs.

[21] At this stage a *process* should simply be considered as an entity which defines a particular execution of a program. In the next chapter we discuss the concept of processes in greater detail.

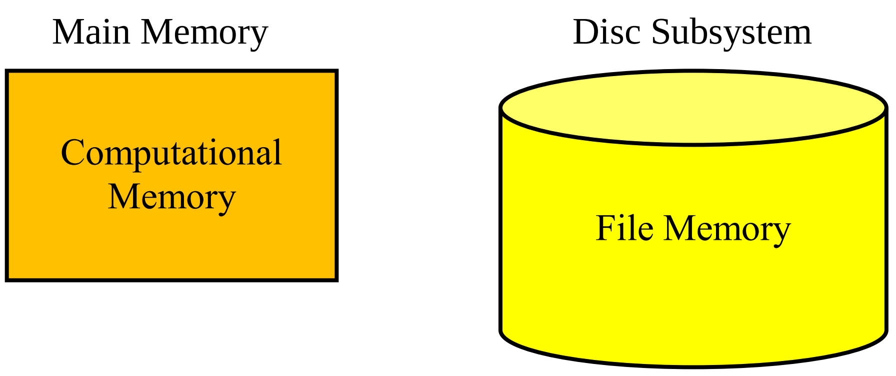


Figure 7.1: Computational and File Memory in Early Systems

This use of the memory devices took advantage of their different properties. The high speed of the main memory (assisted by cache memory) was important for keeping the CPU supplied with the program instructions and data which it needed while executing processes. But the lower cost and persistence properties of disc devices allowed much more storage to be attached to the system at a reasonable cost, and it also catered for the long term survival of data and program files even when the power was switched off.

Of course when data and programs became active, these had to be copied into the main memory. The data was brought into main memory by file system mechanisms; usually only the active parts of a file were transferred into main memory "buffers". Programs were typically copied in full into the main memory by a program loader when the user activated a process.

2 The Transition to Virtual Memory

In the 1960s, when the commercial exploitation of computer had become a serious business proposition, it did not take long for user demands to stretch the limits of the early model of memory use. This meant that as the possibilities for executing multiple processes in parallel on a single computer were improved, users wanted ever more processes to be concurrently active. This led in principle to the need for more main memory. To keep prices affordable, operating system designers experimented with the idea of allowing users to partition their programs in such a way that individual partitions (known for example as "overlays") could be separately loaded into the memory as they were needed. However, this was by no means an ideal solution, because it put a substantial burden on user programmers to organise their programs carefully, and with the simple memory protection mechanisms then available it could easily lead to errors. It also considerably increased the complexity of operating systems.

By the mid to late 1960s it was evident that a radical change was needed, and one after another manufacturers began to adopt an idea which had already been around since the very beginning of the 1960s, *virtual memory*. In 1962 Kilburn and his colleagues at Manchester University in the U.K. had published a description of the first *paged* virtual memory system, which they had implemented in the Atlas computer [36]. At about the same time the Burroughs B5000 computer [37] was introduced in the U.S. with a rudimentary form of *segmented* virtual memory. We shall discuss the differences between virtual memories based on segments and on pages as the unit of organisation later in this chapter, but before we do so it will help to get a more general feeling for the basic idea underlying virtual memory.

3 Program Locality

The overlaying technique (which was replaced by virtual memory) was based on the idea that a program could be decomposed into sections which need not all be loaded into the main memory at the same time. The problem lay not in the basic idea, but in its implementation.

It is not at all a serious problem that only parts of a program are available in the main memory. This is because executing processes typically display an important property called *locality* [38, 39]. When a process is executing it generally works through its algorithm in phases. During any particular phase it tends to make memory references which are clustered together both in time and in memory.

For example when a loop in the program code is being executed, the same sequence of instructions is repeatedly used. If a loop of 500 instructions is executed say 100,000 times then for a substantial time span (in terms of CPU and main memory speeds) instructions need only be available in the main memory from that section of program (which might be only a fraction of the size of the entire program). There is no efficiency loss if the rest of the program code is only held on disc during this phase of the process execution.

A similar consideration applies to data. As an extreme example, consider a loop counter (the variable which is used to count how many times a loop is executed). This is accessed each time round the loop. There will of course be other data references, which are also often repeatedly used over time.

Similarly data elements which are accessed, even if they are not individually used more than once, tend to be clustered in the same area of a program. Consider for example a loop which accesses successive elements of an array or list in each iteration. During the execution of the loop all the accesses to memory will be concentrated around the code segment containing the loop, the data segment containing the array and a few other auxiliary variables such as a loop counter. Provided that these are in the main memory during the phase corresponding to the execution of the loop, it doesn't really matter that the rest of the program is not in the main memory, provided that when this phase comes to an end and another begins, the relevant segments for the new phase can be loaded into the main memory.

In this sense the idea of overlaying was on the right track. It set aside an area of main memory into which parts of a program could be loaded in succession. Those parts of the program which were not needed remained on disc, whence they could be loaded into the main memory when needed.

But there were some serious problems with overlaying. For example the fixed length of an overlay area might not be the appropriate length for accommodating the various program segments at different phases of the program execution. An even more important problem was the addressing difficulties which it created. Application programmer, compiler, linker and operating system all had to be careful to get the addresses right and to ensure that nothing went wrong when switching between overlays, because the same memory addresses were used to address different overlays. Another problem was that addresses must not be used as cross references between different overlays which might not concurrently be loaded into the main memory.

What was needed, and what virtual memory systems generally achieve, is on the one hand a capability of allowing programs to be partly in the main memory and partly on disc, but on the other hand to provide a technique which solves the addressing problems of overlaying. We look at these issues in turn.

4 The Basic Idea behind Virtual Memory

The most fundamental difference between conventional virtual memory and the earlier non-virtual memory systems is that the *computational memory*, i.e. the memory in which computations take place, is no longer viewed as being identical with the main memory. The computational memory is extended by "stealing" some of the disc space from the file system, as is shown in Figure 7.2 (not to scale). This is normally used to hold a version of executing programs copied from the file store by a loader. The operating system's virtual memory manager transfers parts of these programs into the main memory as they are needed. When a section is no longer needed in main memory it is copied back into the extended computational area. Hence the extended computational area holds a (partially) up-to-date copy of each process's program image as it is being executed.

In modern computer systems this extended computational area is usually held as part of a disc which is permanently on-line (i.e. an internal hard disc). The rest of this disc and other discs associated with the system hold the file system space.

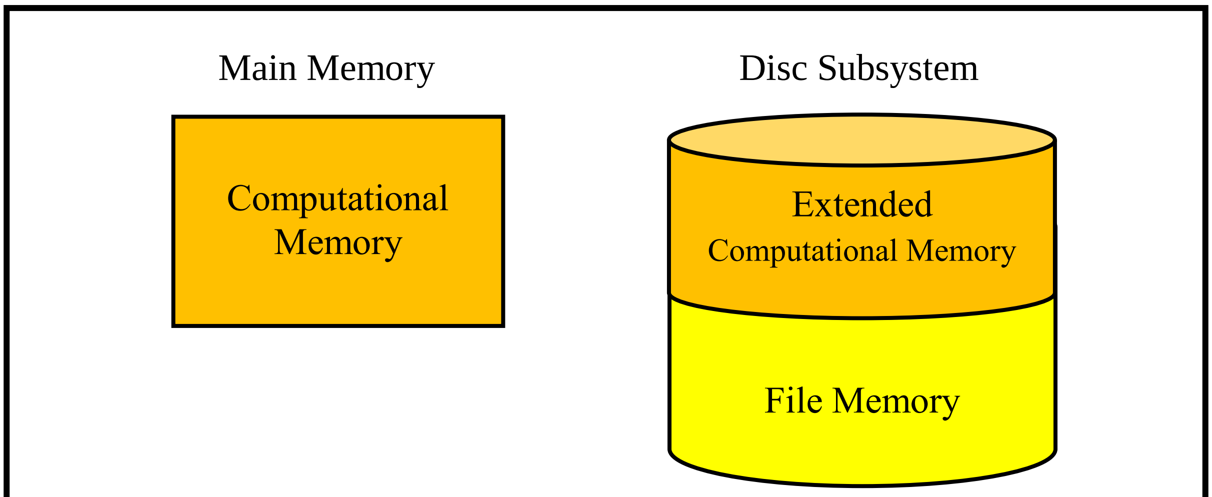


Figure 7.2: Conventional Virtual Memory Organisation

In most systems a clear *logical* distinction is drawn between the file system memory and the extended computational memory, i.e. information on disc is viewed as being in the file memory *or* in the virtual memory but not both at the same time. But there are systems which allow program code to be viewed as being both in the virtual memory and in the file system memory at the same time, in that a file is *mapped* into the virtual memory.

5 Virtual Memory Management

The idea of virtual memory changes nothing in terms of the CPU's need to have rapid access to the main memory. The enormous difference in speed between accesses to main memory and accesses to disc still means that it is quite infeasible for the CPU to fetch instructions and/or data directly from disc. But the least it must achieve is to recognise when an instruction or data word which it is trying to access is not currently available in the main memory and to organise that it be brought into the main memory.

To achieve this almost all computers use a technique called *virtual addressing*[22]. Instead of using main memory addresses as cross references within programs, *virtual addresses* are used. These support a larger range of addresses than can be accommodated by the main memory. They are used to address both instructions and data in programs, but the hardware provides an *address translation unit* (ATU) which rapidly converts these into main memory addresses.

When the ATU detects an attempt to access a virtual address which is not currently in the main memory, it raises an interrupt, i.e. a signal from the CPU to

[22] Exceptionally, the Burroughs B5000 system and its successors (which first invented segmented virtual memory), used a quite different technique, which was not very successful and is of no further interest for our theme.

the operating system which causes the currently executing process to be temporarily halted; this is called a *virtual memory fault* interrupt. It provides the operating system's interrupt routine with details of the problem, in particular with the address which caused the fault. The interrupt routine then analyses this fault and arranges that the missing code or data be brought from virtual memory on disc into the main memory. It then restarts the process at the point where the fault occurred.

In reality the handling of virtual memory fault interrupts is rather more complicated. The operating system's virtual memory manager must find space in the main memory for the missing program unit, if necessary discarding some other program unit (possibly from another process) by writing this back to the extended computational memory. It then reads the missing unit from the extended computational memory into the free main memory space. After that the process can continue execution.

It is crucial that the algorithm which selects a program unit for discarding makes a good choice. The best choice would be the program unit which is not going to be needed for the longest period into the future, but as this would involve crystal ball gazing, most systems settle for a good approximation, namely the program unit which has not been used for the longest period in the past. An efficient implementation of this algorithm, called the *least recently used* (LRU) algorithm, requires some hardware assistance, usually provided in the form of a *used bit* which we shall encounter later.

If a poor algorithm is used to select a victim program unit for discarding, a condition known as *thrashing* can occur. This happens when units which will soon be needed again are chosen for discarding. Then the computer begins to chase its tail, managing to achieve nothing except handle virtual memory faults.

As disc I/O operations are "expensive" in terms of CPU speeds, a further optimisation is often made. Since the extended computational memory contains an image of the entire program, it already has an image of a victim program unit. Consequently victim program units need only be copied back to the extended computational memory if they have been modified since they were last loaded into the main memory. In order that the virtual memory manager can check whether this is the case there is often hardware assistance, this time in the form of a *changed bit* (sometimes called a *dirty bit*). We shall also encounter this later.

6 What form of Virtual address?

In conventional computer systems three different forms of virtual addresses have been used.

– *Paged virtual addresses* simply decompose a program into units of fixed length, called *pages*. These are typically all the same length (but with sizes varying between about 256 bytes and 16 KB in different computer systems).

– *Segmented virtual addresses* decompose a program into *segments*, which are logical units corresponding to items in the program.

– *Segmented and paged virtual addresses* decompose a program into logical units, which are then further decomposed into pages.

In the following sections we discuss each of these in turn, discussing their advantages and disadvantages, and then present an alternative, orthogonal segmentation and paging, which eliminates all the disadvantages and introduces new advantages.

7 Paged Virtual Memory

In a paged virtual memory, programs are decomposed into units of the same fixed length, called *pages*, which are loaded into the main memory on demand. The main memory is similarly divided up into *page frames* which have the same length as pages. Thus when a virtual memory fault (which we can in this context call a *page fault*) interrupt occurs, the virtual memory manager finds an empty page frame (or makes one free by discarding a page already in the main memory) and writes a copy of the requested page image from the extended computational memory into the free page frame. This is a relatively straightforward procedure. Because all pages have the same size, any victim page (i.e. page removed from the memory to make space for another) will do just as well from the viewpoint of space availability, so the discard algorithm can concentrate entirely on other criteria, such as the length of time since pages were last used and/or whether a victim page needs to be written back to disc or not.

In a paging system the programmer does not need to think about how his program has to be composed into overlays and the compiler also just compiles the program as for a non-virtual memory system, starting with an address of 0 for the first word of the program and continuing to allocate addresses in a single linear sequence. In fact the compiler does not even have to be concerned whether the program is longer than the main memory.

A paged virtual address in a program looks just like a main memory address in a non-virtual memory system, except that the virtual address may be larger than a main memory address.

Let us now look at a virtual address in more detail. Suppose it is 32 bits long and the page size of the system is 4 KB (= 2^{12} bytes), then it is possible to regard the 12 least significant bits of the address as an offset within page, and the most significant 20 bits as a page number (see Figure 7.3).

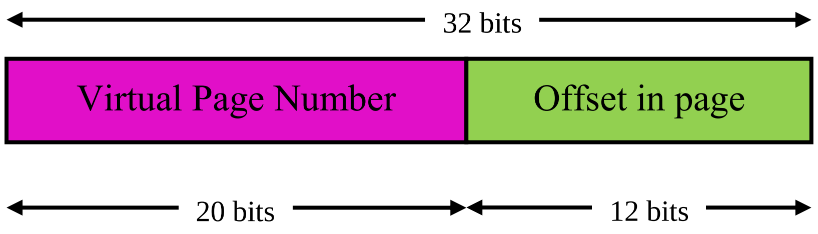


Figure 7.3: A Paged Virtual Address

Such addresses are translated into main memory addresses by an *address translation unit* (ATU). This is situated close to the CPU and is used by the CPU to translate every virtual address which it uses.

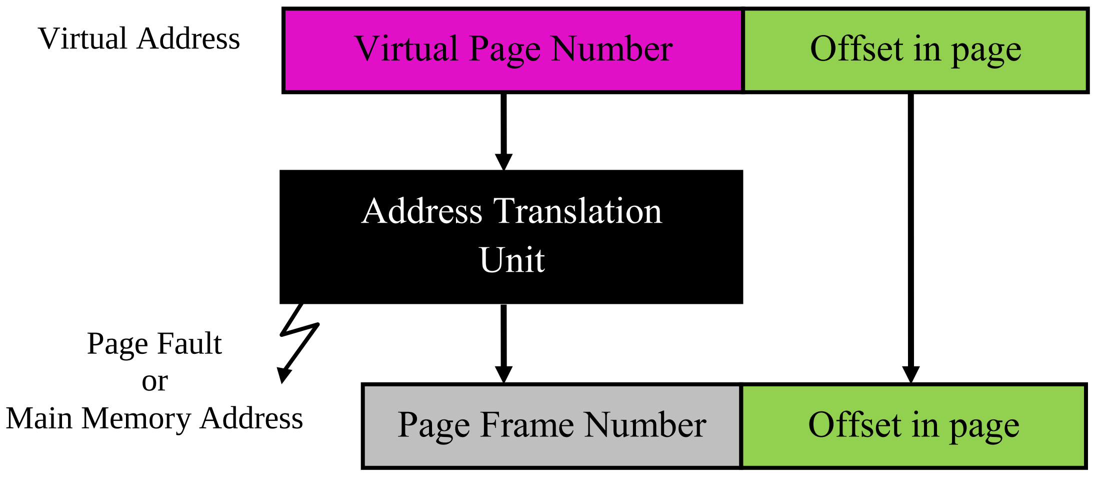


Figure 7.4: The Address Translation Unit as a Black Box

The ATU for a paged virtual memory system in fact needs to translate only the virtual page number part of a virtual address into a main memory page frame number, since the offset in page remains the same. Figure 7.4 shows as a black box what the ATU does. Notice that the ATU cannot always produce a valid translation, because the virtual page number is larger than the main memory page frame number. When an address cannot be translated the ATU causes a page fault interrupt. Then the operating system takes over in order to bring the required page into a page frame of the main memory, as was already described.

7.1 Inverted Page Tables

There is more than one way to implement the black box. The Atlas system [36] used an *inverted page table*. This can be thought of as a table with one entry for *each page frame of the main memory*. Hence the length of an inverted page table is proportional to the length of the main memory. Each entry contains the virtual page number of the page currently resident in the page frame corresponding to the entry (see Figure 7.5). Since there is not always a valid page in each page frame, a *valid bit* is included with each entry.

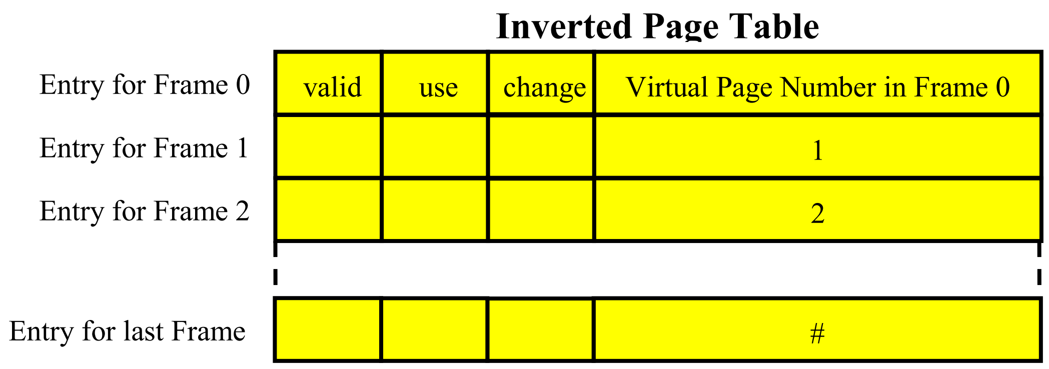


Figure 7.5: An Inverted Page Table

The *use bit* is set by the hardware whenever a byte or word of the corresponding page is accessed (read, write or execute access). The *change bit* is set by the hardware when a process writes into the page.

A page need not always reside in the same main memory page frame. If it is selected as a victim to be discarded, but is later required again, the virtual memory manager can place it in any suitable page frame without consideration for its previous location. Thus pages are completely relocatable in the main memory.

An important point about inverted page tables in a multiprogramming system is that they may contain entries for more than one process. If a virtual address is an effective program address starting at 0 for each process, as is usual in modern systems, then virtual page numbers are not unique! This means either that addresses must in some way be made unique or that the valid bit must be changed for many entries on a process switch, which adds an overhead to the process switch operation. In a later research system developed at the University of Manchester, the MU6-G, for example, the addresses in the inverted page table were made unique by the addition of a process number [40].

Inverted page tables are the wrong way round to be indexed, yet it would be far too slow to carry out a sequential search of each entry. In the Atlas system an associative memory was used. This is a memory in which all entries are searched by the hardware in parallel. This is a very expensive technique in terms of hardware, since hardware for the comparisons must be duplicated for each entry. Hence as memory prices reduced and main memories became larger, the number of entries in such a table increased proportionally. This made the use of associative memories far too expensive and so an alternative implementation of the ATU became widely used.

7.2 Conventional Page Tables

A conventional page table, hereafter simply called a *page table*, can be directly indexed, using a virtual page number as its index. Hence the length of a page table is in principle proportional to the size of the virtual memory. In practice however, there is usually a separate page table for each process; consequently the size of an individual page table is proportional to the length of the program. An entry in a typical page table is illustrated in Figure 7.6.

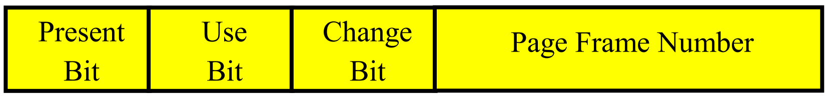


Figure 7.6: A Typical Page Table Entry

With this model the ATU forms a main memory address from a virtual address by using the virtual page number part of the address as an index to select an entry in the page table. If the page is present in the main memory, this holds a page frame number. The offset in page is concatenated with this page frame number to produce a main memory address. This is illustrated in Figure 7.7.

An entry in a page table always has a *present bit* to indicate whether the page described by the entry (i.e. the page which has a virtual page number indexing the entry) is actually in the main memory or not. If it is not present in the main memory, a page fault interrupt is caused. As with inverted page tables there is usually also a *use bit* and a *change bit*.

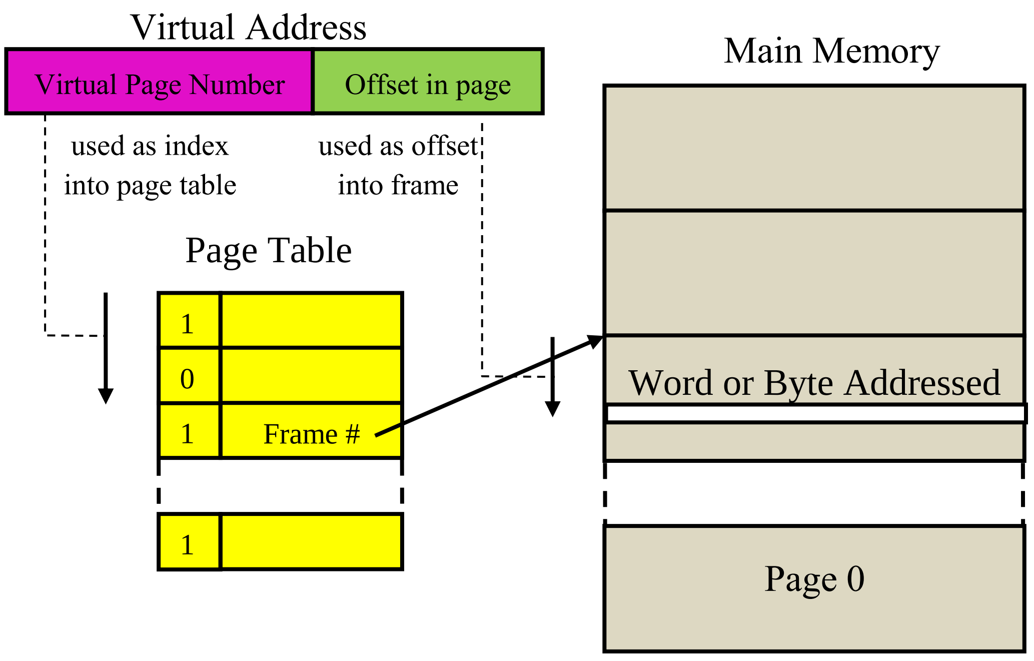


Figure 7.7: Using a Page Table to Translate a Virtual Address

7.3 Making Memory Accesses Efficient

With conventional paging systems the page tables cannot be efficiently implemented in hardware, because they are too large, so they are usually placed in main memory. In principle this means that for every useful main memory access a further memory access is necessary in order to translate the address needed to make the useful access. (This assumes that the page tables themselves can be directly addressed using absolute main memory addresses. If the page tables get too large to be permanently held in main memory, as happened in some systems, they too had to be addressed using virtual addresses, i.e. further main memory accesses may be necessary to address the page tables!)

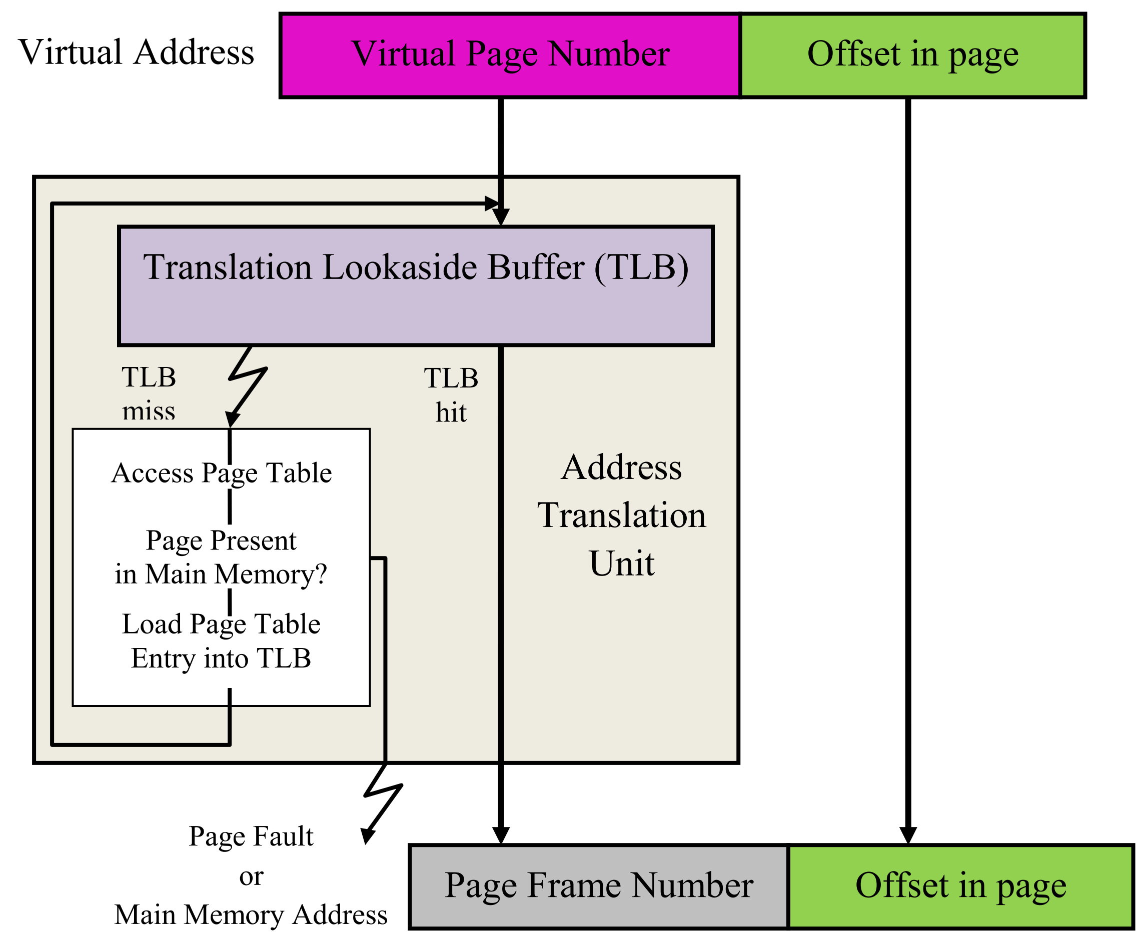


Figure 7.8: The ATU with a TLB using Conventional Page Tables

To have two or more main memory accesses for each useful main memory access would slow the computer down by a very serious amount, so that some additional technique had to be used. The problem is solved in a manner analogous to the way normal accesses to data and instruction accesses can be speeded up, by the use of a cache memory. Since this is needed at a different point in the execution of instructions and serves a different purpose to normal caches, a special address translation cache, usually known as a Translation Lookaside Buffer (or TLB for short), is placed in the CPU's Address Translation Unit. The techniques used for implementing the data cache can also be used to implement a TLB. An overview of the TLB's role in the ATU is shown in Figure 7.8.

The TLB caches entries from page tables, i.e. it provides a rapid mapping from virtual page numbers to page table entries. The present bit need not be cached, since the only entries in the TLB are for pages actually in the main memory. But a *valid bit* is needed to indicate whether the entry in the TLB is currently in use. Thus a TLB is remarkably similar to an inverted page table!

The main difference is that it is incomplete. In other words there is not an entry in a TLB for every page frame. For this reason the functionality is less. If an address cannot be translated by an inverted page table, the corresponding page is not in the main memory. But if the TLB cannot translate an address, this does not necessarily imply a page fault. It often simply means that the required address mapping has to be placed into the TLB, even though the corresponding page might already be in the main memory. But it *might* also imply a page fault.

In earlier computer designs the hardware or microcode was usually responsible for managing the TLB. In some later systems (including RISC systems) this responsibility has been moved into the software.

Finally, because each active process typically has its own page table, virtual addresses (and therefore virtual page numbers) are not unique in conventional paging systems. Hence entries in the TLB are ambiguous and can therefore only be used in the context of the right process. This means that on each process switch all the entries in the TLB (except for those of the selected process) must be invalidated by the operating system.

7.4 Protecting Processes from Errors

Most paging schemes provide a process with some internal protection from errors which might exist in the program code. The aim is to detect errors as soon as possible to prevent unnecessary damage being done internally and to help the programmer to debug (find and correct errors in) his program.

Internal protection against program errors involves the use of three additional bits in each page table entry: a read permission bit, a write permission bit and an execution permission bit (see Figure 7.9). With each memory access the kind of access requested is compared with the appropriate permission bit, and the access is only permitted to proceed if it is of the appropriate kind. If an error is detected, a memory protection interrupt is raised, causing the executing process immediately to be halted.

With this approach it is possible to organize the object code of a program into three groups: program code segments which need an execute permission bit (and possibly a read only bit), normal data segments which need both the read and write permission bits, and constant (non-changeable) data segments protected by a read only bit.

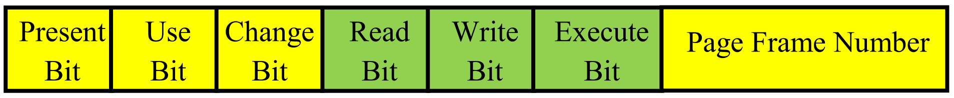


Figure 7.9: A Page Table Entry with Access Permission Bits

8 Segmented Virtual Memory

The first computer system to support the idea of a segmented virtual memory was the Burroughs B5000 system [37], designed in 1961, and its better known successor, the B6700 [41]. Although the Burroughs systems included many innovative ideas, they had little direct impact on later computer systems. One reason was undoubtedly that the underlying memory management model, based on a segmented virtual memory *without virtual addresses*, led to complications which could easily have been avoided by the use of virtual addresses. Furthermore it was not a very secure system, because its security depended on the correctness of approved compilers, and the decision to approve compilers rested in the hands of the computer operators. We therefore describe a simpler model for a segmented virtual memory which uses virtual addresses.

8.1 A Segmented Virtual Memory Model

In this simple model a program is decomposed into *segments* which correspond to logical elements in a program's structure (e.g. individual code routines and data structures). Using the analogy of paging (see Figure 7.3), 32 bit virtual addresses consist of the pair «segment number, offset in segment», cf. Figure 7.10.

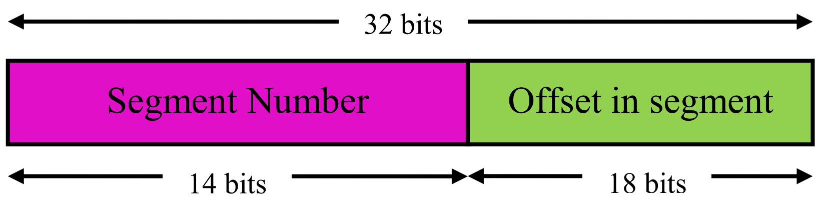


Figure 7.10: A Segmented Virtual Address

Segmentation has a marginal advantage over paging for compilers, because they do not have to linearize programs into a single sequence of virtual addresses. As they encounter a logical structure in the program being compiled they can allocate a segment number for it and produce offsets from that segment number to allow individual parts of the segment to be addressed.

Because the number of bits used to implement a segment offset in a segmented virtual address determines the maximum length of a segment, this field must be longer than the page offset field for a paged virtual address, since segments can be longer than typical page sizes. Some segmented systems decomposed 32 bit virtual addresses into a 14 bit segment number and an 18 bit segment offset. This allows a program to have a maximum of 2^{14} = 16,384 segments, each of which can have a maximum length of 2^{18} = 262,144 bytes or words. Such a division results in a rather unhappy compromise. On the one hand it leads to a quite restricted length for an individual segment. On the other hand the number of segments is also quite restricted. The problem is that some programs are likely to have lots of small segments while others may have a few large segments[23].

In the segmented model each variable length segment can in principle be loaded to start at any address in the main memory. This may at first sight appear to be a very flexible approach, but in practice it creates a difficult problem for managing the use of the main memory. As a result of segments having different lengths, the memory becomes fragmented and difficult to manage. The gaps between segments in the main memory tend to become ever smaller (and therefore less usable).

8.2 Segment Tables

The mapping of virtual addresses onto main memory addresses can be implemented in a similar way to conventional page tables. Figure 7.11 shows how entries in a segment table might look. These entries are considerably wider than page table entries (cf. Figure 7.6) because of the need to store a *length field* in addition to the *full* main memory address at which the segment starts (rather than just the page frame number part of the address).

Present Bit	Use Bit	Change Bit	Read Bit	Write Bit	Execute Bit	Segment Length	Start Address in Segment

Figure 7.11: A Segment Table Entry

Figure 7.12 shows how a segmented virtual address is translated into a main memory address (cf. Figure 7.7 for the equivalent paging diagram). As in paging schemes the overhead of having to make a main memory access to translate each virtual address can be avoided by means of a translation lookaside buffer containing copies of the most recently used segment table entries. The entries in the TLB, as in the segment table, are wider than those needed in the

23 I am not aware that since the development of 64 bit computers any purely segmented system has been suggested, which is scarcely surprising since pure segmentation is not regarded as a good model for virtual memory in view of the memory management problem which it creates (see below).

paging model, making the TLB more expensive to implement.

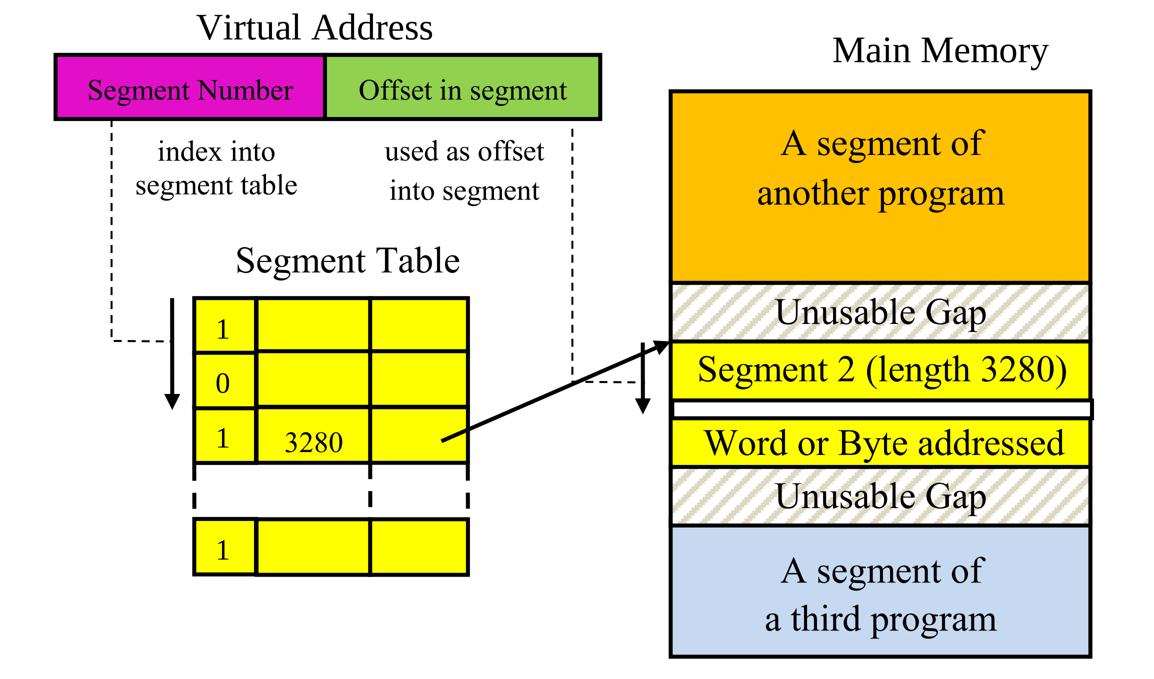


Figure 7.12: Using a Segment Table to Translate a Virtual Address

In contrast with a pure paging system, the protection bits in a segment table entry (read, write and execute bits) map directly onto the properties of the logical segments which the entry describes, so that the compiler does not have to be concerned about clustering segments with related properties together for protection reasons, as may happen in a pure paging scheme.

The appearance of a length field in segment table entries allows the hardware to check that an effective program address (which includes an offset from the start of a segment, Figure 7.10) is within the bounds of the segment. This is essential to guarantee protection between different concurrently active processes, because the main memory word following the end of a segment may contain another segment, possibly from some other program. This bounds check has the further practical advantage that it helps to discover internal program errors which involve attempts to address outside the range of individual segments. This is an error which cannot be detected by hardware in pure paging systems.

9 Comparing Segmentation and Paging

There is widespread agreement that the biggest disadvantage of segmentation lies in the difficulty of managing the underlying segmented main memory. This is a much more difficult task for the operating system than it is in a scheme which supports paging. For example, in choosing a victim segment to be discarded when space has to be found for a new segment, it is not enough to consider which segments have not been used for the longest time or which have not been changed. It is equally important to consider which potential victim segment will leave enough space for the new segment, and what effects choosing a victim have on the main memory fragmentation problem. Consequently no modern computers use a purely segmented virtual memory scheme.

A further disadvantage of segmentation in earlier systems was that a segment cannot be longer than the main memory, and in fact it may not be longer than that part of the main memory available to user programs. (In a paging system longer segments, which are invisible at the hardware level, are automatically decomposed into pages, so the problem does not arise there.) Even today, with main memory as plentiful as it is, it would be a disadvantage to have to place very large segments in the main memory without decomposing them into pages, because the likelihood is that accesses concentrate around a particular part of the segment, making it unnecessary for the rest of the segment to use up main memory.

On the other hand segmentation was the preference of compiler writers in earlier systems, because it nicely reflects the logical structure of programs and it provides a better hardware framework for detecting program errors at an early stage. (This is the reason why the B6700 designers implemented a segmented virtual memory.) A further advantage of segmentation is that it is easier to delete individual segments and create new ones.

It is therefore not surprising that researchers began to look for a scheme which could effectively combine the advantages of both while avoiding their disadvantages.

10 Combining Segmentation and Paging

There have been several attempts to combine segmentation with paging. How researchers have approached this issue has usually been strongly influenced by their understanding of what a segment should be. So far we have followed the Burroughs philosophy (but not their implementation) in assuming that a segment corresponds to a logical element in a program, such as an array or a procedure. This approach leads to the view that segments will usually be very small, typically smaller than a sensible page size.

An alternative view was developed by the Multics designers [42], who regarded segments as an architectural vehicle for implementing files in the context of their aim of achieving direct addressability (a theme to which we will return in the Chapter 12). Their idea was that files in the file system should be mapped into the main/virtual memory as segments. In this case many segments can be expected to be considerably larger than a page. As they demonstrated, it is relatively easy to treat a segment as an entity which can be composed into multiple pages. This has become a common approach and the next section describes it as a model which extends the paging and the segmentation models presented so far.

10.1 Paged Segments

The starting point for understanding the conventional way of combining segmentation and paging is the virtual address structure which it typically uses (see Figure 7.13). The basic new idea compared with the models which we have previously described is that a segment can be decomposed into pages. So we have a segment number, as in the segmented model, a page number within the segment, and an offset within the page. The page number and offset parts are in fact viewed by the compiler simply as an offset within the segment, since paging is invisible to it. Thus the compiler in principle has all the advantages of segmentation, except that if he uses this scheme to implement individual small program units in separate segments then much memory space will be lost to internal fragmentation!

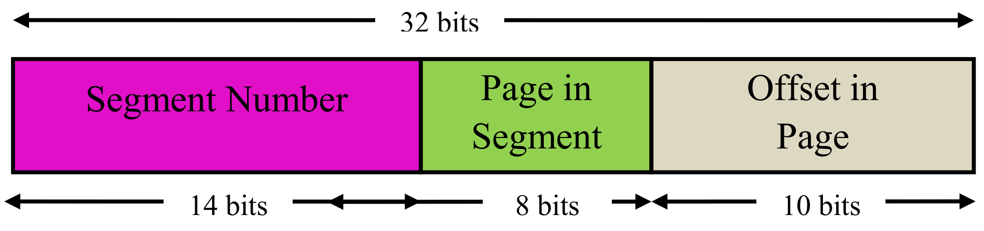


Figure 7.13: A Segmented Virtual Address

All three parts of the virtual address are visible to the operating system and to the address translation unit. This is because the virtual memory translation tables are organized in two parts. For each program there is a segment table, which is indexed by the segment number part of the virtual address. In contrast with the pure segmentation model this segment table does not contain the address of the segment in main memory; this now becomes the address of the start of the page table for that segment. The page table is then indexed by the page-in-segment part of the virtual address. As in the conventional paging model this page table holds the page frame number in the main memory holding the page. The address translation procedure is illustrated in Figure 7.14.

One advantage of this structure is that the logical properties of segments can be held in segment table entries (where they logically belong) while the memory management properties can be held in the page table entries (where they logically belong). Figure 7.15 illustrates what these entries might look like in terms of our previous models.

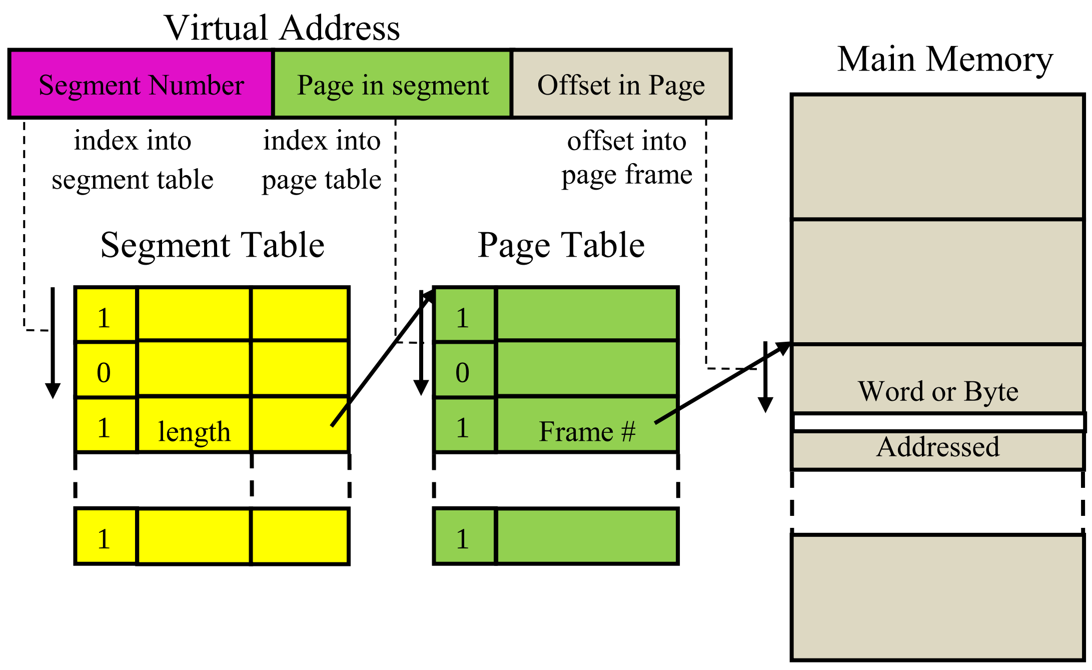


Figure 7.14: Segment and Page Tables to Translate a Virtual Address

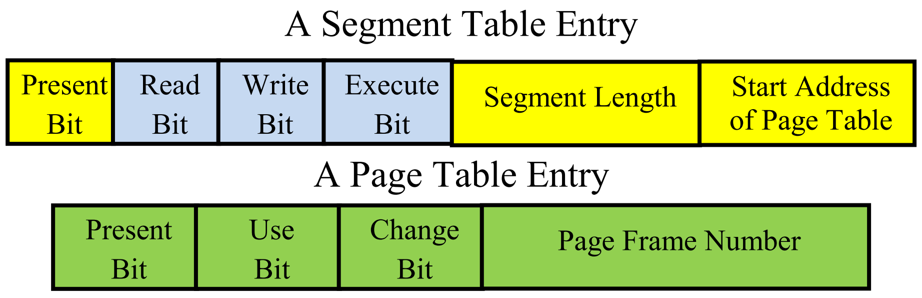


Figure 7.15: Segment and Page Table Entries
in a Segmented and Paged Model

Each kind of table entry can have its own present bit. In the case of the page table entry this indicates whether the page in question is in the main memory or not. But there are two possible interpretations for a present bit in the segment table entry. The most obvious one is that it indicates whether the page table for the segment is present in the main memory. This would allow page tables themselves to be discarded from the main memory.

But the present bit in the segment table can also be used for a different purpose, more in the sense of a valid bit, indicating whether the entry in the segment table is actually in use. This interpretation allows segments to be bound dynamically into a process's address space.

The segment length field in the segment table entry can also be used in two ways. The most obvious use is to hold the actual full segment length in words or bytes. In that case it serves exactly the same purpose as the length field in a purely segmented system. But it can be kept shorter if it is regarded as a count of the number of pages in the segment. This second possibility ensures that a process cannot access beyond the last page of the segment, which is important to guarantee protection between processes. But it does not ensure that an effective address remains within the logical bounds of a segment, since the latter may not occupy the entire last page, and so it loses one of the advantages of pure segmentation.

10.2 Making Memory Accesses Efficient

Logically the translation of a virtual address into a main memory address requires that the address translation unit make two additional main memory accesses, one to the segment table entry and one to the page table entry. (If, as is found in some systems of this kind, the virtual memory tables are themselves addressed by virtual addresses, even more main memory accesses may be required to carry out an address translation.) This might appear to be a disadvantage compared with both of the simpler models, but in reality it is not very significant, because most address translation operations are fully handled by the TLB. Furthermore support for combined segmentation and paging does not substantially complicate the TLB, because the latter does not need to have a three part view of virtual addresses. Its task is simply to determine whether a page is in the main memory. From its viewpoint the pair «segment number, page in segment» can be viewed as a single virtual page number, just as in a paging system, provided that it maintains the logical protection bits and the memory management bits with each entry. Thus the TLB for a combined segmentation and paging scheme turns out to be exactly like that for a simple paging scheme. If this has a high hit rate (in modern systems the hit rate reaches about 98 %), the occasional extra reference to the main memory is not very significant.

10.3 Evaluation of Paged Segments

It seems at first sight that this model effectively combines the advantages of both the paging and the simple segmentation models while avoiding their disadvantages. Memory management is based on paging, which is much more effective than segmentation. Thus the main memory can be divided into fixed length page frames and these can be matched easily to fixed length sectors of disc in the extended computational memory. Similarly it is possible to take advantage of the logical properties of segments, so that internal protection works well in terms both of the use of bounds checks on segment lengths and the checking of basic access modes, without explicitly having to cluster similar segments together, as was necessary with the simple paging scheme.

The combined scheme even introduces a new advantage. In a purely segmented memory it is not easy dynamically to extend the length of a segment, because of the problems this causes in the main memory, but in a system in which segments are paged, there is no problem in adding new pages at the end, because these are separately paged in, exactly like any other page.

However, our discussion has so far made an important assumption, viz. that segments are typically large entities, following the Multics idea that they can be used to map files from the file system directly into the virtual memory. Henceforth we refer to such segments as *architectural segments*. But studies of programs segmented for the Burroughs B5500 have shown that typical segments, in the sense of logical program units, are in fact very small on average. In one study, Batson, Ju and Wood [43] measured segment sizes in a collection of sample programs and found that 60 % of all segments use less than 40 words, and that the largest average segment size for the various classes of segment which they considered was only 181 words. A similar study a few years later by Batson and Brundage [44] on a different program sample confirmed that segment sizes on the B5500 are very small.

If we apply such figures to the combined segmentation and paging scheme just described, then we find that the result is disastrous in terms of internal fragmentation. To put this into perspective let us first consider the loss to internal fragmentation in the conventional paging model. This is on average a half page per program (since the final page of the program is usefully used up to an arbitrary point). Even if the compiler clusters segments with similar properties together to take advantage of protection bits, the loss through internal fragmentation is only half a page per property group (altogether one and a half pages if executable code, constants and writeable data appear in a single object program). But in the combined scheme just described it is *at the very best* an average of a half page per segment (which assumes that segments are much larger than page size), and if segments are on average less than half a page long, as the Burroughs studies suggest, then internal fragmentation can lead to programs which are more than double their natural length! What is equally unfortunate in this case is that the page tables are almost entirely wasted, since almost all of them contain only a single entry! For this reason researchers continued to seek for alternative ways of combining the advantages of segmentation and paging in ways which eliminate this problem.

11 Conclusion

In this chapter we have introduced the most commonly used techniques for managing virtual memory. As described, the situation is not particularly satisfactory from the viewpoint of security. There are at least two reasons for this.

First, the conventional virtual memory approach, whereby a part of the disc memory is "stolen" by the virtual memory management system to serve as an extension of the main memory, results in considerable duplication of mechanisms. Disc space is handled in two quite separate ways. One the one hand part of the disc space serves as a basis for a file system which takes substantial responsibility for security issues at the "higher" level of a system. On the other hand part of the disc space is managed in a quite different way by lower level software, which is equally responsible for security, but at the level of executing programs. This duplication is part of the reason why systems are extremely insecure, because, like the Berlin Wall, it offers hackers with opportunities to play off one part of the security mechanism against another, and it is an understatement to assert that the end result is complicated. In fact it is *very* complicated.

This leads to even further duplications. For example, programming languages are forced to handle access to data which "belongs" to a program in one way, while resorting to quite different mechanisms to handle data which is stored in file systems. We will take up such issues in a later chapter, where we will also show that a much simpler, more efficient and, most important, a more secure way of organising virtual memory is possible.

The second major problem with conventional virtual memory is that it is not capable of effectively handling small segments. At first sight this might not seem to be important, but as will become clear in a later chapter, this deficiency eliminates the possibility of efficiently implementing an important class of systems which are capable of enormously enhancing the security of operating systems. This is also an important theme which is taken up in a later chapter.

Chapter 8
Processes

Support for parallel activities within a computer system is one of the key functions of any operating system, because

– computer installations are often required to support more than one user in parallel,
– even in single user systems the user expects to carry out activities in parallel,
– the hardware of the computer can carry out activities in parallel.

This means that the operating system must not only be able to manage and control parallel activities, but it must also be able to react to parallel events (e.g. hardware interrupts indicating that an input-output operation has completed).

The hardware of a computer system consists of various components which can operate in parallel with each other (e.g. CPU, disc and printer activities can overlap in time) and these operate at quite different speeds. The speed differences can be very significant, e.g. a modern CPU can carry out billions of instructions in a single second, a modern disc (HDD) cannot carry out a read or write operation in less than about 3 to 10 milliseconds (i.e. at most ca. 330 to 100 operations per second)[24], while a printer may take several seconds to print a page. Consequently if a CPU were to execute one program at a time, waiting for its I/O operations to complete, most of its available processing time would be lost in the idle state. Hence multiprogramming, i.e. allowing many programs to be executed quasi in parallel on a single processor, is absolutely essential. To achieve this, a central component of the operating system, called a *process scheduler* or *thread scheduler*), maintains a pool of *processes*, from which it selects individual processes to be executed. Ideally this pool contains a mixture of

24 SSD devices can operate about 100 times as fast as traditional discs, but this is still very slow in relation to processor speed. For a comparison between HDD and SSD devices see https://www.storagereview.com/ssd_vs_hdd

I/O intensive processes (which require a substantial amount of I/O activity) and CPU intensive processes (which spend most of their time performing CPU calculations). Given such a mix the process scheduler will normally give priority to the I/O intensive processes, which will only need a small amount of CPU time between activating and waiting for I/O devices, allowing the CPU intensive processes to execute while the I/O intensive processes are waiting for their input-output operations to terminate.

1 Scheduling Algorithms

To determine which process can actively use the CPU the process scheduler uses a process scheduling algorithm. Here we describe by way of example some aspects of a scheduling algorithm on a general purpose system where a wide variety of user applications might co-exist. The likelihood that all the applications described would coexist in practice in a single node is not particularly high, but is not impossible. We use this application mix to illustrate that even extreme examples can be combined into a single scheduling algorithm.

1.1 High priority real time processes

It is essential that some applications processes have absolute priority over other activities at a node. Suppose for example that a process is responsible for checking and controlling the temperature of the key units in an atomic power station. It is self-evident that this process should be able to gain control of a CPU in order to carry out its checks at regular short intervals, probably measured at the millisecond level[25]. The only thing that such a process needs to do in the normal case is to scan the readings of all its measuring devices and then go to sleep for a short interval. Of course it is a different matter if the readings are not within the expected range, and the process will then need to activate alarms, etc.

In other words there are some processes which mostly need only a little CPU time but which should be immediately scheduled at regular (short) intervals and should be permitted to keep control of the CPU as long as they need it. When they complete a regular short activity they will normally wish to go dormant for a short time and then be re-awakened when their next activity should be started.

From this it is obvious that a process should be able to deactivate itself (we refer to this as `short-suspend`) and in this example specify a time interval before it is re-activated.

[25] I am not an expert in nuclear power station control. This example is simply used to illustrate that it is really important for some applications to be favoured by a process scheduling algorithm over all others.

1.2 Medium priority I/O intensive processes

Some processes carry out many input-output (I/O) operations and need to be activated when each such I/O operation completes, in order to initiate the next I/O operation. A common example is a spooler process, which has the task of maximising the use of a printer device. Many applications must print from time to time, and the relationship between their processing time (to produce a printable result) and the frequency of printing can vary enormously, depending on their purpose. For example a weather forecasting application must carry out an enormous amount of processing before it produces printable results, but a request for a data base application to print information from the database needs very little processing time before it can print the next line.

The idea behind spooling is that the use of a printer can be optimised by preventing applications from directly printing to a printer device (which in the case of weather forecasting might result in a printer sitting idle for minutes or even hours at a time) and instead for them to "print" their results to a file. Put simply, instead of writing its results to a printer, an application stores them in a file and when the file is complete it sends a request to a spooler module to print them. This queues such requests and prints them when a printer is available.

Hence when an application's results file reaches the top of the spooler's queue, these are read from the module in which they are stored and are written at full speed to the printer. In this way the printer is used optimally.

Of course the spooler can organise its printing queue according to priorities, taking into account users who need rapid results, but such details need not concern us here. The important point is that from the viewpoint of process scheduling a spooler uses a minimal amount of CPU time (e.g. to read information from a print file and pass this to the printer); it must then wait until the information has been printed before it can print more.

Thus spooler processes (usually one per printer) and other processes which have similar CPU and I/O characteristics need a relatively high priority (to keep the printer or other I/O device working at full speed), but of course they should have a lesser priority than high priority real time processes. Because the CPU time needed by them is very short, giving them a quite high priority scarcely affects processes which have a lower priority.

1.3 Interactive Processes

In real-time transaction processing systems such as airline reservation systems, where for example travel agents sit at terminals attached via a network to a central computer in order to make bookings for their clients, this style of interaction usually requires a small amount of CPU time to handle a transaction and then, from the viewpoint of CPU time, a huge waiting period until the next transaction is entered from the same user. The transaction itself typically requires a very small amount of CPU time and a few database accesses. From the user's viewpoint the important issue is that he is not kept waiting for more than a second or two between inputting a request and receiving his answer.

1.4 CPU-intensive processes

Some computations (e.g. many scientific applications, including for example weather forecasting) need enormous amounts of CPU time and only occasionally (in terms of CPU time) need to output a result before continuing their calculation, and as indicated above, their output operations are converted into virtual memory accesses.

1.5 Combining the above Requirements into a Single Process-Scheduling Algorithm

While the likelihood that all the above requirements would occur in a single computer system is relatively small, to design a CPU scheduling algorithm which caters for them all is not particularly difficult.

The first step is to organise the processes into different priority levels. Here we envisage four such priority levels, numbered say from 0 to 3. Priority 0 (the highest priority) is used for *high priority real time processes*, priority 1 for *medium priority I/O intensive processes*, priority 2 for *interactive processes*, and priority 3 (the lowest priority) for *CPU-intensive processes*.

We then apply the rule that at all times the CPU is allocated to the process of the highest priority which is in the ready state (i.e. is able to execute, see next section).

It is obvious why high priority real time processes should have the highest priority, but why for example should spooling processes have the next highest priority? The first part of the answer is that it is important for such processes (e.g. spoolers) to keep their I/O devices (e.g. printers) running at full speed. But it is also important that they usually require only a trivial amount of CPU time, so that they scarcely affect the CPU usage of lower priority processes, while at the same time can keep their I/O devices active.

The interactive processes are placed below the medium priority I/O intensive processes because they sometimes require more than just momentary use of the CPU (normally in contrast with the first two categories), and so can make good use of the CPU for somewhat longer time intervals. However the issue of fairness is especially evident here. If one transaction monopolises the use of the CPU for too long then this will cause other transactions (and therefore users at other terminals) to wait for a possibly intolerably long period without receiving a response. To avoid this, the priority 2 queue can be organised on the basis of time slicing, i.e. a CPU time slice of a few milliseconds is determined as the maximum time limit for a process to hold the CPU. If this time slice expires the CPU is forcibly taken from the process and the next process at this level is allowed to run. If there is no priority 2 (or higher) process in the ready state, the first ready process in the priority 3 queue, i.e. a CPU-intensive process, is selected.

The above process scheduling algorithm is just an example of a scheduling algorithm. Computers used for different purposes may need more specialised algorithms, so it would be a mistake to think that the same algorithm will suit all computer systems.

2 Process Scheduling States

From the viewpoint of process scheduling, a process can be classified as being in one of four states: *inactive* (when the process is doing nothing, which in conventional systems usually means that the process does not exist), *ready* (when it can execute but has not yet been assigned to a CPU by the scheduler), *running* (when it is currently active on a CPU) and *blocked* (when it is waiting for some condition to be fulfilled, e.g. completion of an I/O operation). Figure 8.1 illustrates these states and possible the transitions between them.

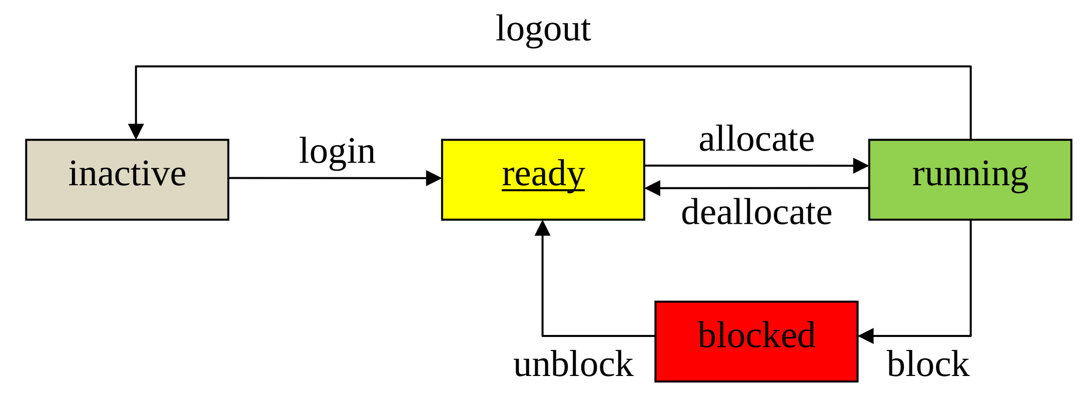


Figure 8.1: Process/Thread States and Transitions between States

The *ready* state indicates that the process can be executed (i.e. be placed into the *running* state) when the scheduling algorithm selects it. In this case the scheduler allocates a CPU to it.

A process in the *running* state can forcibly lose the CPU as a result of an interrupt (e.g. because its time slice has expired). In this case it is returned to the ready state. Alternatively it can voluntarily relinquish control of the CPU (e.g. in order to wait for the completion of an I/O operation which it has initiated), in which case it enters the *blocked* state. It is released from the blocked state, returning to the ready state, when an interrupt indicates that the reason for it being blocked has been lifted.

The important purpose of the process scheduler is to carry out *reschedule* operations, i.e. to select a process from the ready state and put it into the running state. This can occur as a result of a process being added to the ready queue or as a result of a process being unblocked following an interrupt.

3 Process State[26]

A process should not be confused with a program. It is not a program as such, but is the execution of a program on a CPU. A program is merely an algorithm, i.e. a recipe for carrying out a computation. When a program is executed, it has a process state, which from the viewpoint of the process scheduler defines those aspects of the process's activity which must be preserved when the process is not actually executing on the CPU. The most important part of a process's state, from the process scheduling viewpoint, is the values in its CPU registers.

When the process scheduler makes a *process switch*, it must store the current register values in a data structure associated with the process which has previously been running and then reload the registers with the values associated with the process now selected to run on the CPU.

Thus the process scheduler must maintain a list of processes, containing the saved register values for each process along with other relevant items of information (e.g. the time consumed so far by each process as it runs, its priority, etc.).

4 Program Structure

In preparation for the following discussion about processes we must now make a short digression to describe an important aspect of program structures.

Normally a program, unless it is very simple, is decomposed into smaller units which we call routines. A routine, variously called a procedure, function, subroutine or, in object oriented programming, a method, contains instructions for carrying out a part of the program's task. This not only helps the programmer to break down his program into smaller, easily intelligible parts, but it can define a subtask which may have to be performed several times during the execution of the program. We then speak of a program *calling* or *invoking* a routine. Such a routine might be defined along the following lines (using a pseudo programming language which is much simpler than real programming languages).

[26] The term *process state* as used in this section should not be confused with a process's *scheduling state*, described in the previous section.

```
routine identifier (parameters)
begin
     programming statements
end
```

The identifier is the name of the routine. The parameters allow different values to be passed each time the routine is called (invoked), thus allowing the routine to perform different variants on the same task. Statements are the instructions which define how the routine's task is to be carried out. These may, but need not, contain a **return** statement. The routine returns to the caller either when it has executed all the statements to the end, or when it encounters a **return** statement.

A routine can be called several times in a program, and each invocation causes the same code to be executed. When it has completed it returns to the calling code at the point following the call.

Suppose a graphics program contains a routine called **draw_circle** which has three parameters. The first two define the position of the centre of the circle as integer coordinates, and the third defines the radius, this could be invoked by the following statement:

```
draw_circle (13, 17, 5)
```

which would draw a circle at the coordinates 13, 17 with a radius of 5 (using whatever unit size is assumed, e.g. centimetres).

Here is an example of how a program using this routine might look.

```
program my_drawing_program
begin
     initial statements
     draw_circle (13, 17, 5)
     more statements
     draw_circle (24, 37, 10)
     final statements
end
```

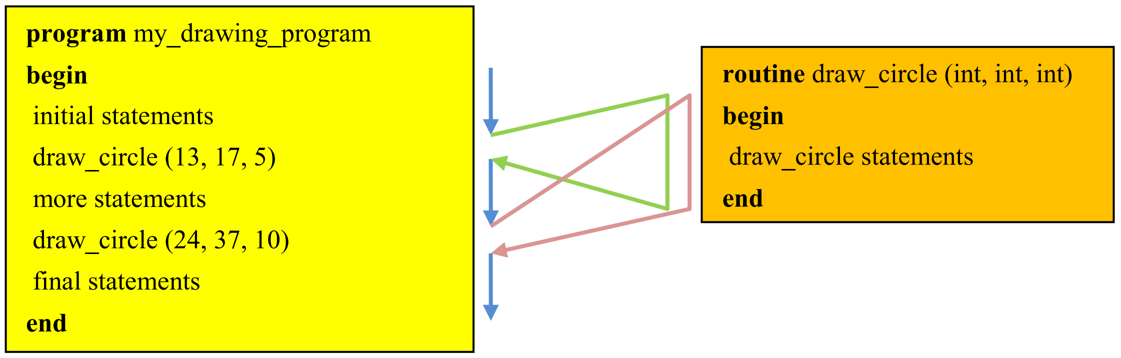


Figure 8.2: A Program Invoking the Same Routine Twice

Figure 8.2 shows how this would execute in a running process. The blue arrows represent the progress of execution in the main program. The green and red lines represent the invocation and execution of the routine for the first and second times respectively.

Each time the routine returns, it must jump back to the instruction following the invocation, which is different for each invocation of the program. Hence at the time of each invocation the return address must be stored in such a way that it can be recovered when the routine is ready to return.

5 Process (or Thread) Stacks

Compilers are programs which translate a computer program defined in a high level language (e.g. C++ or Java) into the low level instructions which the CPU can understand and execute. Most compilers organise the calling of routines by means of a data structure called a *process stack*. A stack, in general terms, is a data structure which can grow and shrink at one end (the top), just as a stack of trays in a self service restaurant grows and shrinks only at the top. In computing circles we talk about *pushing* items onto a stack and *popping* them off. In the case of a *process stack* each time a routine is called a *stack frame* is pushed onto the stack and each time a routine returns, its stack frame is popped. Process stacks can support a number of useful facilities, as we now describe.

5.1 Routine Linkage

As is evident from Figure 8.1, each time a routine returns back to the calling program, it returns to a different place, namely to the instruction after the call instruction. Consequently the compiler must add code to the compiled program (known as run-time code) which, each time the routine is called, records the address to which it must return. It records this information at the base of the new stack frame which it creates for the routine call. Hence when the routine returns, the address in the program at which the calling program must be re-activated is available on the stack.

The routine linkage segment on the stack must also hold information about where the previous stack frame begins. Usually this is addressed via a register, so that the address in this register (which we will call F for frame[27]) is also stored in the linkage segment. Other information may also be stored in the linkage segment to help with the return operation.

5.2 Parameters and Local Variables

Since the parameters passed to the called routine can change with each call, they must be recorded at a point on the stack where the called routine can locate them. Similarly the called routine will need its own set of working values (called local variables) which can change with each call.

[27] In some computers this register is a special register, but it can also be one of the general purpose registers.

The compiler therefore provides run-time code which places a parameter segment above the linkage segment on the stack. Before the call this area can be addressed by the calling routine to set up the parameter values, and when the call has taken place the called routine can address the same segment to access its parameters. Then immediately above the parameters the run-time code also provides space for the routine's own local variables.

5.3 Expression Evaluation

Sometimes a calculation, as expressed in a high level language, requires some intermediate results which the programmer has not explicitly named and which therefore no local variable has been declared which would provide space to store the value until it is needed. For example if the programmer writes:

```
  a = (b + c) * (d - e)
```

the run time code will provide space for the local variables a, b, c, d and e in the stack frame, but when the calculation is carried out there is no space for the result of evaluating (b + c) or (d - e).

This problem is also solved by using a process stack, such that these results, as they are calculated, can be pushed onto the top of the current stack frame and once they are used they can be popped from the stack.

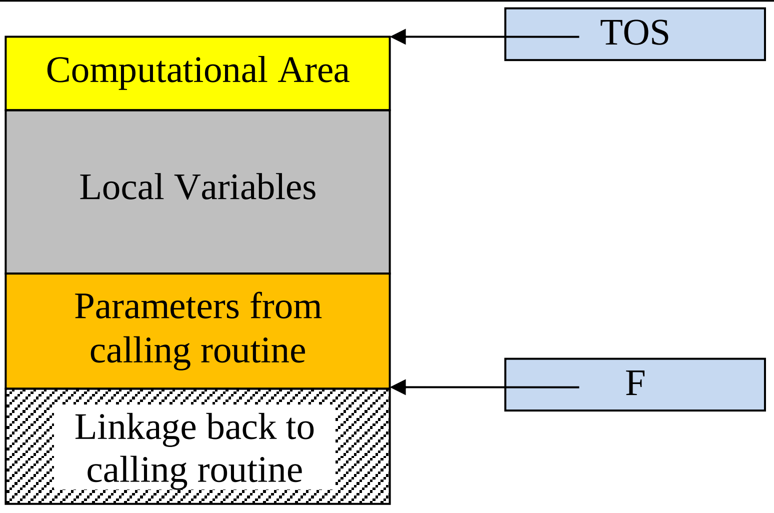

TOS
Computational Area
Local Variables
Parameters from calling routine
F
Linkage back to calling routine

Figure 8.3: A Stack Frame

This implies that the top of the stack is not always at the same point in a stack frame, which means that in order to keep track of this another register is needed, which we call TOS (top of stack)[28]. Like F, TOS must also be stored in the linkage segment.

We now see that a stack frame can basically be viewed as consisting of four

[28] Like F, TOS might be implemented as a special register or it might just be a general purpose register maintained by run-time code.

parts: a linkage segment, a segment for parameters, a segment for local variables and an expression evaluation area (see Figure 8.3).

5.4 The Stack Structure

So far the impression may have been given that the stack has only two levels: a main program and a routine called from the main program. However, that is not the case. Routines can be called not only from a main program but also from within other active routines. Thus a process stack can have many levels of stack frames. Each stack frame, together with its linkage segment, provides enough information for the corresponding routine to carry out its own task, possibly call other routines and when it has completed its task, to return to its caller (see Figure 8.4).

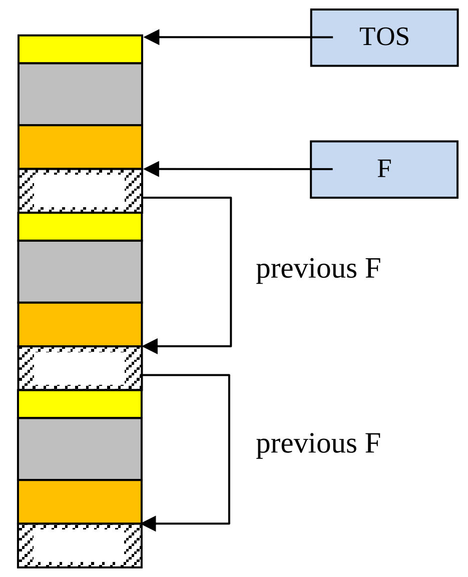


Figure 8.4: A Process Stack

This structure also supports *recursive routines*, i.e. routines which call themselves in the middle of their computation. Programmers must take care, when defining recursive routines, that the recursion has a termination condition, otherwise the stack would grow "endlessly".

The same structure also supports *re-entrant code*, i.e. code which can be executed simultaneously by multiple processes, provided that each process has a separate process stack, and that the program code is not modified.

6 Global Variables and Parameters

So far the impression may have been given that a routine uses only its own local variables and the parameters directly passed to it. In practice this is often not the case. Many high level programming languages are defined in such a way that a routine can address both its own locally declared variables and a set of variables which have been globally declared at the start of the program. This means that it has to be possible to use addresses which are not relative to the current stack frame register F. A simple way to achieve this is to have a similar register, which we shall call G, for addressing the global variables of a program. G addresses the variables in the first stack frame of a dynamically executing program. At any given point the currently executing routine can then use addresses relative to F or to G. The linkage at the bottom of the stack indicates that the end of the program execution has been reached.

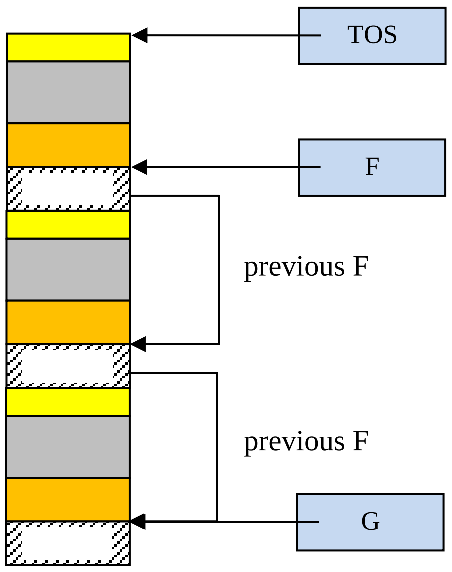


Figure 8.5: A Process Stack with Global Parameters and Variables

7 Calling the Operating System

Most conventional systems have a special mechanism which allows a process to call the operating system. However, another interesting idea which was used in some systems (e.g. Burroughs B6700 [41], Multics [42], ICL2900 [45, 46][29]) was to implement calls from application programs to the routines of the operating system more or less in the same way as routine calls within the user's own program. In other words the invocation of an operating system service routine takes place on the process stack of the application process.

Although this is in principle very similar to the stack technique which we have already seen, there is one major difference. The operating system is compiled separately from its application programs and therefore should not need to address the global variables of an application program. Instead it possibly has its own global variables, which must become addressable when one of its routines is invoked.

When the operating system comes into the picture we realize that there are actually two different kinds of global variables which may need to be addressed.

[29] The second reference for the ICL2900 is a reprint of the first.

The operating system has some data structures which must be persistent and which need to be addressable from *all* processes as long as the system is running. (As an example of such a data structure, consider the queues of files which a printer spooler has to manage.) Such data structures must survive at least as long as the system is running, and they cannot be put onto the stack of an application process. For this reason it is useful to have a further addressing base register. Let us call this the P register, where P is short for "persistent". This will point to data items which are not on an application's process stack.

An operating system routine can receive its parameters from the application program using the same parameter passing mechanism as is used for passing parameters between local routines of an application process. However, the call to an operating system service routine and the linkage at that point on the stack have to be rather special, because the P register has to be set up and because the return mechanism must – for protection reasons – invalidate this.

8 Handling Interrupts

Not all the routines of an operating system are invoked directly by application processes. In particular, interrupt routines are activated directly by the hardware at arbitrary points in the execution of other processes when an interrupt pending signal is detected (e.g. to indicate that an I/O operation has completed). Yet symmetry (for example in order to keep the compiler of the operating system routines simple) suggests that these should also be handled using stack frames. The B6700 designers were consistent in this respect and used a slight variation of the same routine calling mechanism to implement interrupt routines.

What they did was logically very simple. They treated an interrupt as if it were a *forced* routine call. The currently executing process (whatever process that might be) was stopped, its program counter value and other contextual information were stored (as normal linkage) on its stack at the location above the address in the TOS register, so it appeared that the application process had called the interrupt routine. The interrupt routine then used a new stack frame on the application process stack. When the interrupt routine completed and exited, this then appeared like a normal routine return, so the interrupted application process could continue as if it had never been interrupted.

But there is a catch in treating interrupts as forced routine calls. What happens if the interrupt has been caused because there is no more space on the stack? In the B6700 there was in fact another ALU register, called SL (stack limit), which in principle defined the last memory address for the current stack. Each push operation on a process stack was checked to see if the resulting TOS value would go beyond SL. If it did, a stack overflow interrupt was signalled. This is where the danger lay, because the stack has no more space on which to execute this interrupt.

Not to be deterred by this problem the B6700 designers worked out the maximum number of words needed by this interrupt routine, and actually set the SL value to where it should have been minus this amount. In the later B7700 system this was one of the few things changed in the architectural design. In that system a separate stack was activated whenever an interrupt occurred. (This had the advantage that higher priority interrupts can interrupt lower priority ones and still have stack space, providing this is kept to a reasonable level.)

9 Processes and the Operating System

There are some interesting implications of handling operating system routines on application process stacks. The first of these is that at least in theory no operating system processes are needed. However, in most practical systems, even in those which support the execution of operating system routines on an application process stack, there are usually some additional operating system processes for carrying out activities which perform tasks that are independent of a particular application process.

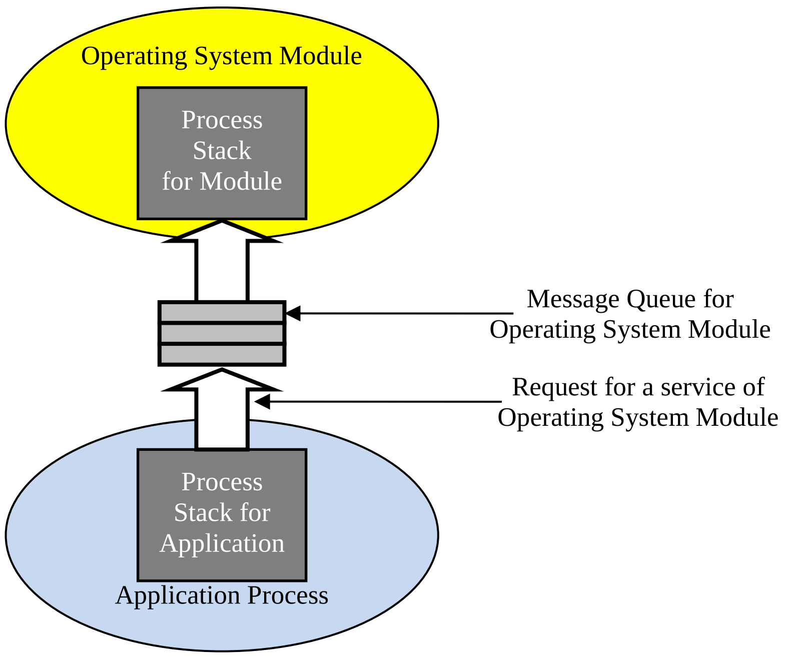

Operating System Module
Process Stack for Module
Message Queue for Operating System Module
Request for a service of Operating System Module
Process Stack for Application
Application Process

Figure 8.6: Communication in the Out-of-Process Model

There are in fact two models for decomposing an operating system into processes. The simplest involves having a separate process for carrying out each operating system activity; this is a standard technique used in very many operating system designs. We refer to this kind of design as *out-of-process*, because operating system services are provided for an application *out of* the application's *process*, in a separate process. The technique is sometimes called *message-oriented*, because the application process must pass its parameters as a message from one process to another. This is illustrated in Figure 8.6.

The alternative is the technique implemented in the B6700, where operating system services are provided in the process belonging to the application. We refer to this kind of design as *in-process*; it is sometimes also called *procedure-oriented*, because the operating system routine is implemented as a routine (see Figure 8.7).

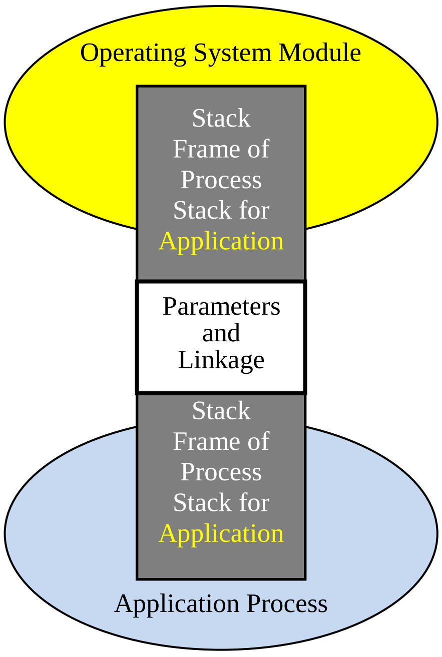

Operating System Module
Stack Frame of Process Stack for Application
Parameters and Linkage
Stack Frame of Process Stack for Application
Application Process

Figure 8.7: Communication in the In-Process Model

In an interesting paper by Lauer and Needham [47] these two techniques were compared. The authors reached the conclusion that they were duals of each other. However, they overlooked two significant points, which are discussed in detail in my former student Kotigiri Ramamohanrao's PhD thesis [48]. First, as we shall see later, there are some fundamental differences when it comes to the issues of protection and security. Second, the two models are not equivalent in terms of their dynamic properties, e.g. with respect to the level of parallelism attainable in each.

Since a process is the unit of execution on a computer it is easy to fall into the trap of thinking that the fewer the number of operating system processes, the less concurrent or parallel activity can be achieved in carrying our operating system tasks. In fact this is quite the opposite of the truth. In a system in which there is a process statically assigned to provide a particular service (i.e. the out-of-process model), this service must be used serially by different applications. This is because the server process examines its input message queue, selects a request, and then services it. When it has finished it selects another request, etc. Meanwhile all the application processes which have requested this service are usually blocked waiting for the service[30].

In contrast, in the in-process model no application processes are automatically blocked (also less work for the process scheduler!) since they can all concurrently call the same routine of the operating system and execute in parallel with each other, using their own process stacks. So there are fewer processes but at the same time there is more potential for parallelism.

This does not imply that it is always possible to achieve full parallelism in an in-process system. The difficulty comes when these processes have to access the same data structure. This would happen for example when two operating system routines address shared operating system data using the P register which we introduced. If they are both only *reading* the data there is no problem, but if they need to *modify* it, then the data structure can become inconsistent, leading to wrong results. We discuss this and other synchronisation problems at the end of the chapter.

The important point here is that operating system processes executing in the same module in an in-process system do not always have to take turns to use the same module. First, not all data structures accessed by these processes are shared. Even if they are executing in the same routine, they may only need to access data via the F and G registers, which is not shared and therefore causes no problems. But even if they are accessing shared data (in our model via the P register) they may not be modifying it but only reading it. So in practice more parallelism can be achieved in an in-process design.

A final point is that when processes cooperate with each other, such as when operating system routines are active on different stacks, they may want to communicate not only implicitly via shared variables but also explicitly by sending signals to each other. Most operating systems have an interface which allows signals to be sent between processes to allow them to cooperate. This is an area which can easily lead to security breaches if it is not handled properly. The problem is that cooperating processes have to trust each other and rely on each other sending the right signals at the right time. If the system does not have a mechanism for ensuring that only the right processes are allowed to communicate with each other, then for example a process receiving a signal on trust, but sent by a malicious process, might take some actions which it would not otherwise have taken.

10 Multiple Processes

The discussion has so far assumed that within a single program the only parallel-

[30] It is possible to associate more than one process with each operating system module, but that creates new problems which are not important for the present discussion.

ism which can occur is when users activate different processes. However, there are some programming activities where it makes sense for a single activation of a process to split into multiple tasks (hereafter called *threads*), which can be carried out in parallel. This means that an application program can be written as a *parallel program* (using a *parallel algorithm*), which can be decomposed dynamically into several threads that can be multiprogrammed with each other. The form which this sometimes takes is that a routine declared in a program can be invoked either by using a normal routine call (as we have been discussing in the last few pages) or as an activation of a separate thread.

Different programming languages often define different semantics for parallel programming. For example some programming languages (e.g. Burroughs Extended Algol) allow an initial thread to create child threads which share the data of their parent thread. Each new thread has a separate stack which is created when the thread is activated. The first stack frame of this "child" stack contains the parameters and local data of the routine which has just been invoked as a thread. Its linkage points back to the creating ("parent") stack. As a routine can access not only its own variables (for which space is created on the new stack in the usual way) but also more global variables, it must be possible for the child thread also to address variables on the parent stack. In other words, the F register of the child thread points to locations on the new stack while its G register points to locations on the parent stack. Both threads can proceed to execute as normal. Both can call new routines, they can call the operating system, and they can even create further new threads.

The threads can communicate with each other either explicitly (by sending signals using an operating system interface) or implicitly (by changing values in the globally shared data on the parent stack). If shared data structures are used, access to these must be synchronised in a manner similar to the way just described for the case where threads synchronise access to data accessed via the P register.

A problem arises if the parent thread terminates before the child thread completes its task, because this results in the destruction of the stack frame containing the global data which should still be accessible to the child thread. In the B6700 this problem is solved by destroying the child thread. This is not as bad as it sounds, because the threads can cooperate with each other via signals and/or shared data, and so they can arrange to terminate in the right order. If necessary the creating thread can ask the thread scheduler to block it until all the threads which it has created have terminated.

11 Synchronisation: Mutual Exclusion

The above discussion of the merits of in-process vs. out-of-process systems raised the issue of process synchronisation. This is an important issue, because a failure to understand synchronisation issues can often lead to errors in the design of operating systems and that in turn can create security loopholes.

11.1 The Basic Problem

At the hardware level the modification of variables (e.g. additions and subtractions) normally take place in the ALU registers. Values are brought from memory into registers (as operands) and the results of operations are written back to registers. The results are later written back to the main memory.

Suppose that two processes (or threads) each wish to execute the same piece of code to increment a counter by one. At the assembler/machine level this might be achieved using three instructions:

```
LOAD  R1, COUNTER    {load the value of the memory variable
                      called counter into register 1}
ADD   R1, 1          {add 1 to the value in register 1}
STORE R1, COUNTER    {store the value of register 1 back into counter}
```

where R1 is an ALU register which is available to programs and counter is a shared operating system variable addressable via P. This may at first sight seem quite harmless, but consider what happens when two processes execute it in parallel. Each has its own register set, so each process's R1 is a different version of R1. Suppose the counter starts off with the value 3, and the actual timing of the sequence of code for the two processes turns out as follows:

```
Process 1  Process 2
LOAD  R1, COUNTER          {R1 for process 1 now contains 3}
      LOAD R1, COUNTER     {R1 for process 2 now contains 3}
      ADD  R1, 1           {R1 for process 2 now contains 4}
      STORE R1, COUNTER    {COUNTER now contains 4)
ADD   R1, 1                      {R1 for process 1 now contains 4)
STORE R1, COUNTER          {COUNTER now contains 4}
```

The surprising result is that each of two processes intended to add one to a counter which had an initial value of 3, and the result turns out to be 4, not 5! This is a simple form of synchronisation problem which can arise when several processes attempt to modify the same shared variable. Such a situation can arise in practice in a single processor system, if the first process is interrupted after executing the LOAD instruction and the process scheduler then selects process 2 to run before process 1. In a multiprocessor system it can happen anyway, if two processors execute different processes using the same shared data.

There is of course a solution. Such a section of code is called a *critical section*. Critical sections where processes write to the same variables (whether using the same or different code sections) can only sensibly be executed in sequence; the first must finish before the second starts.

11.2 Mutual Exclusion

The synchronisation problem just described requires mutually exclusive access to the variables in question (here COUNTER), i.e. when one process is accessing the critical section, all other processes must be excluded from that section.

To achieve this, a process, when it attempts to enter a critical section, must execute an entry protocol indicating that it wishes to enter the section. If the instructions executed in the entry protocol allow this, then the process enters the section, changes the variable(s) and then uses an exit protocol to say that it is no longer in the critical section, thus allowing some other process to enter the section, i.e. it executes the following sequence:

```
entry protocol
critical section
exit protocol
```

In order to implement the entry/exit protocols some hardware support must be provided.

11.3 Dekker's Algorithm

The absolute minimal hardware support necessary is that when a word of memory is modified by a CPU instruction this must be achieved indivisibly, i.e. nothing must be allowed to interrupt the hardware writing process. The Dutch mathematician Th. J. Dekker is accredited (by Edsgar W. Dijkstra [49]) with the first correct solution for this problem, using only the indivisibility of write operations to a single word. However, his solution is only of academic interest, because

– it only works for two processes,
– the entry and exit protocols are different for the two processes, and
– it involves *busy waiting*.

Busy waiting describes an entry protocol where a process must continuously loop, consuming CPU time and making continuous memory accesses, until the other process exits from the critical section. It is also not a *fair* scheduling technique, since an unlucky process may be starved of useful use of the CPU over a long period. Furthermore a risk of *deadlock*[31] arises when a priority scheduling algorithm is used, if a low priority process successfully enters a critical section, but loses the CPU to a process of higher priority which is executing a busy wait loop. Hence more support is needed from the hardware in real systems.

[31] A deadlock arises when two or more processes are waiting for each other to release some resource, with the result that they will wait forever.

11.4 Turning Off Interrupts

In the past some systems solved the problem by turning off interrupts. This means that in a single CPU system a process can proceed without interruption to its completion on the CPU. Other processes cannot be scheduled on the CPU because the scheduler cannot be activated. But this simple technique has several problems, e.g.

- Turning off interrupts affects a particular active CPU, but does not have a global effect on other CPUs. Thus in a genuine multiprocessing (i.e. multi-CPU) system this solution is ineffective.
- If interrupts are turned off for too long, information can be lost (for example – but not only – in real time systems).
- In almost all systems the "turn off/on interrupts" instruction is a privileged instruction (because it can lead to loss of data, etc.). Consequently this method is not available for use by non-privileged processes, i.e. user level cooperating processes.

11.5 Busy Wait Instructions

Some CPUs provide a *test-and-set* instruction, which is easier to understand and use than Dekker's algorithm, but this also results in processes having to busy wait until the critical section is free. In more modern systems this instruction has generally been replaced (e.g. in IBM mainframe systems) with a *compare-and-swap* instruction[32], which also relies on busy waiting.

11.6 Semaphores

In 1965 Dijkstra developed a new idea for solving synchronisation problems, called a *semaphore*. It is based on the idea that counting variables should be implemented indivisibly.

A semaphore *sem* is a structured abstract variable on which two special operations can be indivisibly carried out. It consists of an integer (*sem.counter*) and an associated queue of waiting processes (*sem.queue*).

The first operation is a P operation (the entry protocol). (P is the initial letter of the Dutch word "passerem" (to pass); the P operation determines whether a process may pass a point in the code or should wait.) The second is a V operation, the exit protocol. (V is the initial letter of the Dutch word "vrygeven" (to release); the V operation releases the critical section when completed.)

For practical purposes the operations (on a semaphore *sem*) are defined as follows:

[32] see for example http://en.wikipedia.org/wiki/Compare-and-swap

```
P (sem) =>
      [[sem.counter = sem.counter - 1;
      if sem.counter < 0 then suspend (sem.queue)]]
V (sem) =>
      [[sem.counter = sem.counter + 1;
      if sem.counter ≤ 0 then activate (sem.queue)]]
```

The bracket pair [[...]] is used to indicate that the bracketed instructions are carried out indivisibly

Put simply, the P operation decrements the counter and if the result is negative the process is placed into the related queue of waiting processes. The V operation increments the counter and if the result is not positive a waiting process is activated (made ready).

Using these operations the solution to the mutual exclusion problem is trivial:

```
Semaphore sem = 1 {a semaphore variable sem initialised to the value 1}
P(sem)
critical section
V(sem)
```

This solution avoids the problems of the other attempts to solve the mutual exclusion problem: there is (apparently) no busy waiting, it can be used for any number of processes, and the solution is the same for all processes.

11.7 Implementing Semaphores

An important issue is how semaphores can be implemented *indivisibly*. We consider two possibilities.

a) The Process Scheduler implements semaphores using techniques described earlier, or

b) Special instructions are provided to implement semaphores.

Process Scheduler Implementation of Semaphores. In this case application and/or operating system processes invoke operations of the process scheduler to carry out P and V operations. The process scheduler uses one of the techniques described above (e.g. *compare-and-swap* operations or turning off interrupts) to ensure that indivisibility is guaranteed. This represents an improvement because

– the time required in the mutual exclusion state (e.g. busy waiting, interrupts off) is only for the duration of the P and V operations, not for the entire period of the application's critical section, and

– the process scheduler retains control over the turning on and off of interrupts (if that is how the process scheduler achieves mutual exclusion).

But the disadvantage is that each time a process enters and exits a critical section the process scheduler must be called. This is quite costly, since on many occasions there may be no clash between processes wishing to use the same critical section (i.e. no queuing operation may be necessary).

Semaphore Instructions. It is not normally feasible to implement the entire code of a semaphore operation (especially the queuing part) as an ALU instruction, but it is feasible to implement the counter manipulation part. In the ICL2900 Series [46], for example, two indivisible ALU instructions were provided, along the following lines[33].

a) Decrement & Test (counter, local) is equivalent to the first part of the P operation, and is defined as follows:

```
DECT (counter, local) =>
     [[ counter := counter - 1; local := counter ]]
```

This instruction indivisibly decrements a shared counter variable and copies the result to a process-local variable.

b) Test & Increment (counter, local) is equivalent to the first part of the V operation, and is defined as follows:

```
TINC (counter, local) =>
     [[ local := counter; counter := counter + 1 ]]
```

This instruction indivisibly copies the value of counter to local then increments the shared counter variable.

The user code can then combine these machine instructions with normal process scheduler operations (here *suspend* and *activate*) as follows to implement P and V operations:

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

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

Suspend and activate must be indivisible operations (as is usual in the process scheduler).

Although both the instructions and the scheduling operations are indivisible, an interrupt can occur between these two parts of a P or V operation. This is not a problem provided that the process scheduler's suspend and activate operations are commutative, i.e. the order in which suspend and activate operations occurs has no effect on the final result, e.g. if the following scheduler calls are made in sequence and the process involved is already active

```
activate   - process continues
activate   - process continues
suspend    - process continues
suspend    - process continues
```

[33] The actual ICL2900 implementation reverses the counter values, i.e. a mutual exclusion semaphore is initialised to -1 and is incremented by the equivalent of the P operation, while the V equivalent operation decrements the counter. This is logically equivalent to Dijkstra's approach, but we follow Dijkstra's suspend and activate instructions convention to avoid confusion. For a more general description see [131].

```
  suspend   - process is suspended
```

The advantages of this implementation are as follows:

– DECT and TINC are not privileged instructions. They can be used in user programs.

– If the appropriate condition is satisfied, then there is no call to the operation system (suspend/activate); this is very efficient.

– The process scheduler routines are short and simple: no extensions are needed for the basic process scheduler model discussed earlier.

Nevertheless the fact remains that the actual scheduler operations must be synchronised using one of the more primitive methods discussed earlier, i.e. turning off interrupts or busy waiting, assuming that the hardware does not have a special instruction for this (which may be necessary in a multi-CPU system).

12 Further Synchronisation Problems

One aspect of semaphores was not considered in the last section, where it was assumed that the semaphore's value is initialised to 1. Such a semaphore is called a *binary* semaphore. But it is possible, and often sensible, to initialise semaphores to other values. In general the current value of a semaphore can be understood as follows:

```
  > 0: the number of resources currently free
  = 0: no resources free and no waiting processes
  < 0: the number of processes waiting for a resource.
```

The mutual exclusion problem is a particular example, where the semaphore is set to 1 because processes are competing for a single resource, the critical section. We now consider briefly some important problems which can be solved by semaphores.

12.1 Bounded Buffers

When parallel processes cooperate with each other they must communicate either by sharing variables in memory (e.g. in an in-process system), or by sending messages to each other (e.g. in an out-of-process system).

If the operating system provides processes with a message passing facility (such as that illustrated in Figure 8.6) this must be implemented using shared memory. We now consider how such a message passing facility can be implemented (from the viewpoint of synchronisation).

The problem to be solved is in fact a more general problem, which can appear in many aspects of operating systems, database systems, etc. It is caller the *producer/consumer* problem and the shared memory is called a *bounded buffer*. We begin with the simple case of *one producer* process (which adds new entries to the buffer) and *one consumer* process (which removes entries from the buffer). Since the buffer cannot be infinitely long, it is called a *bounded* buffer. Suppose such a buffer has space for eight entries (see Figure 8.8). The first is entry 0, the second entry 1, etc.

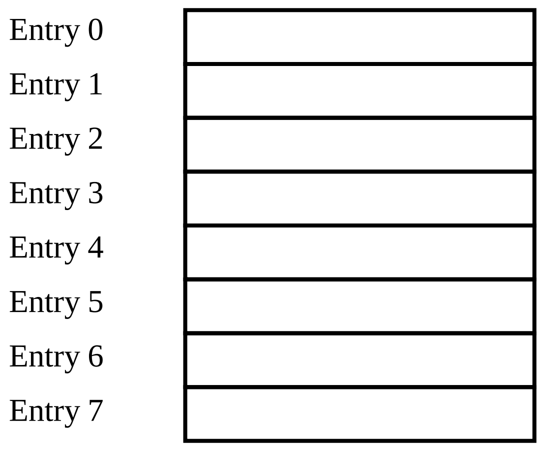


Figure 8.8: A Bounded Buffer with Eight Entries

The producer process can fill all these entries in turn, starting at entry 0, but when he attempts to add an entry after entry 7, he must wait until the consumer has removed entry 0, freeing it up to be used as entry 8, etc. The two processes work at their own speeds and so it is possible that a producer attempts to add a new entry to a full buffer, or a consumer attempts to remove an entry from an empty buffer. The synchronisation problem is to ensure that the producer must wait to add an entry to a full buffer and that a consumer process must wait if it attempts to remove an entry from an empty buffer.

The solution is quite straightforward if *general semaphores*, i.e. semaphores which can be initialised to any value, are used. It requires two semaphores. We call the first *empty*, and the second *full*. Assuming that the buffer is empty when the processes start, the semaphore empty is initialised to 8 (despite the numbering of the buffers!), because there are 8 empty buffer slots (i.e. 8 empty resources); the second semaphore (*full*) is initialised to 0, because there are no full slots at the start.

Apart from the buffer itself and the two semaphores, 2 further shared variables are needed. We call the first of these integers *nextfree*, because it holds the index value (i.e. the entry number as it appears in Figure 8.8) of the next free buffer (so that the producer knows which entry to use). The second is called *nextfull*, as this tells the consumer where to find the next full slot.

The solution of the problem for a bounded buffer with eight entries is as follows. Initially both *nextfree* and *nextfull* are set to 0.

The producer code, which is designed as a loop to be executed as many times as there is something to produce, is as follows:

Producer algorithm:

```
produce an entry
use the semaphore operation P(empty) to claim an empty slot
insert the new entry into the slot indicated by nextfree
add one to nextfree modulo 8, i.e. if the result of adding 1 causes
     nextfree to exceed the value 7, divide the result by 8 and
     store the remainder in nextfree
use the semaphore operation V(full) to release a full slot
```

Repeat these steps as often as necessary, each time filling a slot.

Consumer algorithm:

```
use the semaphore operation P(full) to claim a full slot
remove the entry from the slot indicated by nextfull
add one to nextfull modulo 8, storing the remainder in nextfull
use the semaphore operation V(empty) to release an empty slot
consume the entry
Repeat these steps as often as necessary, each time emptying a slot.
```

Notice that whenever the buffer is full (i.e. the semaphore *empty* reaches the value 0) the producer process will wait. It will be activated as a result of the consumer executing the V(*empty*) instruction, i.e. releasing a full slot.

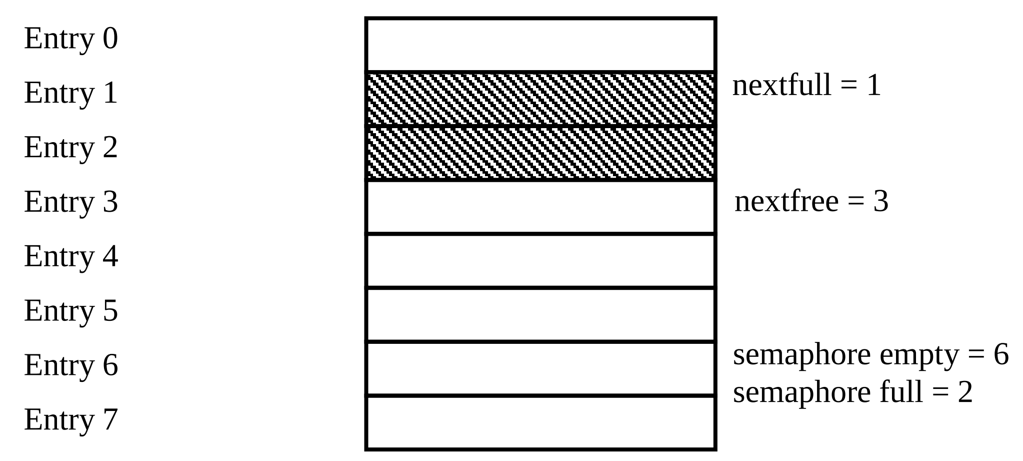


Figure 8.9: A Bounded Buffer with Two Full Entries

Similarly whenever the buffer is empty (i.e. the semaphore *full* reaches the value 0) the consumer process will wait. It will be activated as a result of the producer executing the V(empty) instruction, i.e. releasing an empty slot.

While the buffer is neither full nor empty both process can work in parallel on different slots, without having to wait. Figure 8.9 shows the state of the buffer after 3 entries have been produced and one of these has been consumed.

The reason why this solution is only guaranteed to work with a single producer and a single consumer is that if for example there are two or more producers they must share the use of the variable *nextfree*. To guarantee that they do not try to fill the same slot (which is determined by the value of *nextfree*) they must have mutually exclusive access to this variable. To achieve this, a binary semaphore (initial value 1) can be shared by the producers. We call this *pmutex* (indicating **p**roducer **mut**ual **ex**clusion). In this case a P(*pmutex*) instruction must be inserted between steps b) and c) of the producer algorithm and a V(*pmutex*) after step d), thus causing the steps c) and d), which use the *nextfree* variable, to be treated as a critical section for the producer processes. (Notice that step b) must be outside the critical section, otherwise a deadlock situation could arise, in which the producers wait for each other forever. In contrast the order of the V operations is not so important, because a V operation does not cause processes to wait.)

A similar situation arises for multiple consumers, this time with the variable *nextfull*, This can be solved in the same way, this time by introducing a semaphore *cmutex* (for **c**onsumer **mut**ual **ex**clusion), with a P(*cmutex*) instruction after step a) and a V(*cmutex*) after step c). A similar deadlock situation would be possible if the P(*cmutex*) instruction is placed before step a).

Notice that *pmutex* is not required if there is a single producer and multiple consumers, nor is *cmutex* required if there is a single consumer but multiple producers.

Finally it is worth pointing out that the bounded buffer problem arises in many situations in the design of operating systems where processes cooperate with each other by passing messages.

12.2 Readers and Writers

A common problem occurs when some processes wish to read from a database (i.e. group of variables) while others wish to modify it. Processes in the first group are called "readers" and in the second group "writers".

If only readers are present there is no consistency problem (and therefore no exclusion problem). Readers can share access to the database without creating problems. However, writers must exclude not only other writers, but also readers.

The first and simplest solution to this problem was published by Courtois et al. [50]; it gives readers priority over writers, i.e. readers must only wait when a writer is writing, but writers must wait until no reader is reading. The solution requires a variable shared by readers, *readcount*. This holds a count of the number of readers which are reading in parallel, and is initialised to 0. Two binary semaphores are also required. The first, *wmutex* (writer mutual exclusion), prevents readers and other writers from accessing the database while a writer is active. The second, *rmutex* (reader mutual exclusion) is claimed by the first reader and released by the last reader and allows shared reading to be coordinated.

Writer algorithm:

```
use the semaphore operation P(wmutex) to claim access to the database
use the database in writer mode
use the semaphore operation V(wmutex) to release the database
```

This is of course the simple mutual exclusion algorithm.

Reader algorithm:

```
use the semaphore operation P(rmutex) to gain exclusive access
     to the variable readcount.
add 1 to readcount.
if the current value of readcount = 1 (i.e. if this process
                                             is the first reader)
then use the semaphore operation P(wmutex) to see if a writer is active.
     (If so the reader waits, otherwise it continues.)
Each further reader also adds 1 to readcount and then continues
     without the P(wmutex) operation.
use the semaphore operation V(rmutex) to release access to readcount.
use the database in reader mode.
use the semaphore operation P(rmutex) to regain exclusive access
     to the variable readcount.
reduce the count of readers by 1.
if the result is 0 (i.e. this process was the last reader)
     use the semaphore operation V(wmutex) to release the database
     for readers or a writer
use the semaphore operation V(rmutex) to release access to readcount.
```

If the aim of this algorithm is always to guarantee that readers are given priority over writers and if (as is normally the case) no priority mechanism is built into the semaphore queues, then a small modification of the algorithm is necessary (as my former students and I have noted [51]).

The problem arises because when a writer issues a V(*wmutex*) operation, the queue of processes which are waiting to access the database can contain both a reader and multiple writers. Since there is no guarantee that a reader process will be selected, reader priority is not guaranteed.

To avoid this problem we proposed that an extra binary semaphore should be introduced (which we call *extra*). This is used only by writers and nests the writer algorithm described above within a P(*extra*)-V(*extra*) pair, i.e. the P operation precedes the first step and the V operation follows the last step. This ensures that when a writer is writing, other writers are queued not on *wmutex* but on *extra*. Hence the next writer can only be scheduled after the writer has not only released *wmutex* but also *extra*.

This illustrates how difficult it can be to use semaphores to solve apparently simple problems. Curtois et al. also presented a writer priority solution in the same paper. This is somewhat more complex and need not be presented here.

12.3 Private Semaphores

A semaphore has a related queue. For many problems it is not important in what sequence the waiting processes are reactivated. Mostly a FIFO (first-in first-out) queue is used in practice. But Dijkstra's definition of semaphores does not define a specific ordering, so that an arbitrary order of waiting processes must be assumed.

However, in some cases the order in which processes are reactivated is important, as we already saw for the reader-writer problem. This problem can be solved in a general way using Dijkstra's semaphores.

To achieve this each process has a *private semaphore*. A private semaphore is a normal semaphore, which is typically initialised to 0. Thus if a process issues a P operation on its own private semaphore, this gets the value -1 and the process waits. The process is activated when another process issues a V operation on its private semaphore.

Suppose for example that a computer program, written as a parallel program, is designed to simulate a simple board game such as ludo, a game with its own board and special rules (which need not concern us here – the result of a dice throw distinguishes the individual turns). The important point is that each of four players (or two players each playing two opposite colours, which we ignore for the sake of simplicity), has four coloured tokens of the same colour (blue, red, green or yellow), and each plays in turn.

In the program there would be a representation of the board, with the current positions of the four tokens of each player on the board, together with the following semaphores:

– a mutual exclusion semaphore, which we call *mutex*, initialised to 1
– an array of four private semaphores, each initialised to 0

After the initialisation each player would have a separate process which continuously executes the following instructions (until the game is won):

```
throw dice
select and move counter on board according to rules
change board display accordingly
if game won, finish;
     otherwise issue V operation on next player's semaphore
issue P operation on own private semaphore
```

This is just one of many examples of the use of private semaphores.

Finally, semaphores and their implementation are further discussed in chapter 21, where further research results are described in the context of the SPEEDOS system.

13 Scheduling Resources

Finally, we consider how the scheduling of resources (e.g. determining which process can next use a printer) can be organised, since many readers who are only familiar with out-of-process systems may at first find this puzzling. For such readers it seems natural to have a separate process which decides when another process is given the go-ahead to use the resource.

The solution is actually straightforward and is based on the use of private semaphores. Each process that wishes to claim the resource in question invokes a scheduling module (in his own process) which has the task of determining the order in which each process can use the resource. If the resource is free, then a requesting process can proceed immediately. Otherwise the process must wait (on its own private semaphore) in a pool of waiting processes until the scheduling module determines that it can use the resource. When the process using the resource returns to the scheduling module (in its own process) this indicates that it has finished using the resource. The scheduling module (still executing in the process of the resource which has just freed the resource) then determines according to its algorithm[34] which process waiting in the pool can proceed next. It then releases the private semaphore of the chosen process, removes its entry from the pool of waiting processes and exits from the scheduling module. The chosen process, now activated, can now use the resource. When it has finished using the resource it returns to the scheduler module and releases the resource as described above.[35]

14 Conclusion

This chapter has not directly discussed security issues, but it has laid down an important foundation for understanding SPEEDOS and for solving some very significant security issues which will be discussed in later chapters.

[34] In the case of a printer scheduler, for example, the algorithm might adopt a policy of giving priority to short print jobs, or it might select print jobs based on the seniority of the requester or simply on a first in first out basis, etc.

[35] For an example from SPEEDOS see chapter 33.

Chapter 9
Protection and Sharing in Conventional Systems

Having considered how virtual memory can be organised and also how executing processes can be implemented, we are now in a position to discuss the important issues of protection and sharing. In this chapter we review attempts to achieve these aims in the context of the various memory management models previously presented.

1 Protecting Processes from Each Other

In a multiprogramming system which uses conventional page tables or segment tables each process has its own range of virtual addresses, always beginning at 0. Thus virtual addresses in such systems are not unique. However, each process has its own page or segment table. Since different page tables are held in different parts of the main (or virtual) memory a mechanism is needed which enables the appropriate table for the currently active process to be located. This is usually held in a CPU register, which, in the case of paging, we call the Page Table Base Register (PTBR) or in the case of segment tables the Segment Table Base Register (STBR).

The PTBR/STBR contains not only the address of the beginning of the table for the currently active process, but also its length. When translating a virtual page/segment number the ATU uses this length field to check that the selected unit is within the bounds of the table. If not, this means that the process is trying to address a (non-existent) memory unit (page or segment) which is beyond the program's last real page or segment. Such an error results in a memory violation interrupt.

Like other CPU registers, the PTBR/STBR is part of the state of a process, but this is nevertheless a system register which cannot be accessed by the application program. Whenever a process switch is made by the process scheduler, the corresponding values for the old process are saved and the register is loaded with the appropriate values for the new process. This is illustrated in Figure 9.1 for page tables, on the assumption that Program B is currently active. A corresponding illustration for segment tables would show that the table entries would point to variable length units in the main memory, and in the case of paged segments the segment tables point to page tables which in turn point to different pages in the main memory.

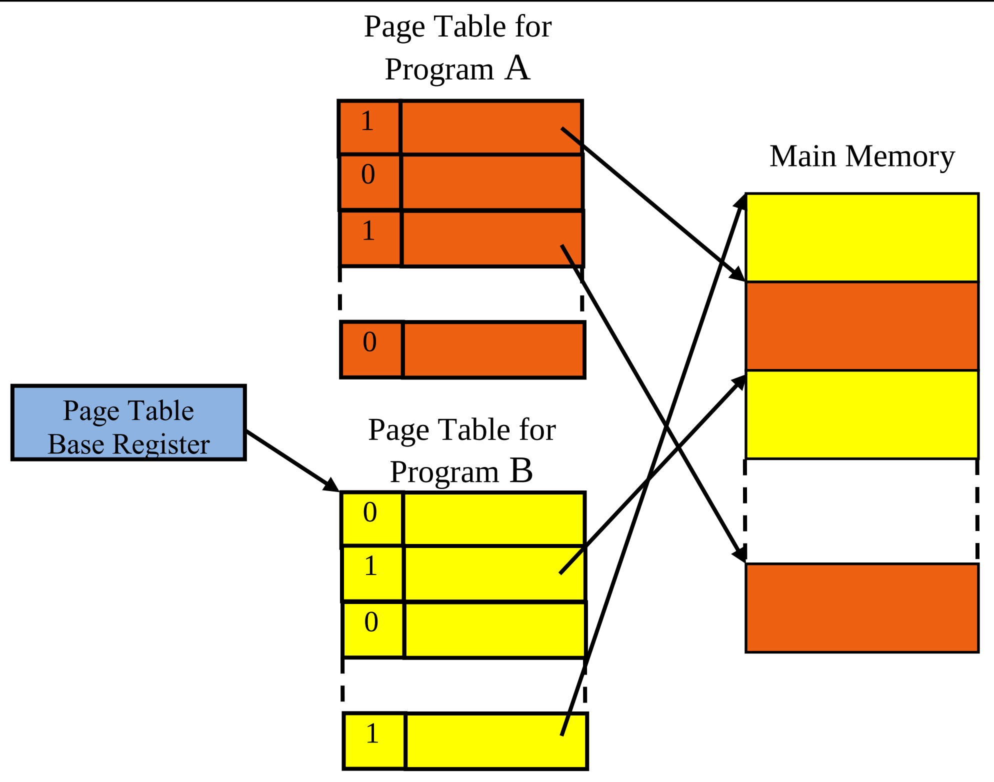


Figure 9.1: Page Tables in a Multiprogramming System

An advantage of this scheme is that the pages of different processes are automatically protected from each other without needing a further protection mechanism, such as existed for example in the IBM S/360 storage key/ protection key scheme [52, 53][36]. This is because any virtual address which a process tries to access is interpreted as being one of its own addresses (using its own table). It cannot address beyond its own range of addresses, because of the length field in the PTBR/STBR. So there is no way it can formulate addresses associated with other processes.

2 Protecting the Operating System

If the operating system were to have its own page or segment table then it too would be protected from user processes. Unfortunately this simple solution has some practical disadvantages. The most important of these is that operating sys-

[36] The second citation is a reprint of the first.

tem designers like to be able to address not only their own pages, but also those of user processes – at the same time as they can also address their own pages! This is necessary for example when the operating system wishes to access the parameters supplied by an application process in a request for an operating system service, since the application can store these only in its own memory space. Consequently another scheme was sometimes adopted for protecting the operating system.

First we have to look at how the addressing works. Since the aim is to allow the operating system to address information stored in a user process, it is clear that the applications process's page or segment table must be active. To address the operating system means that a page table for this must also exist and be addressable at the same time. The first step towards achieving this is to have *two* PTBRs or STBRs, one each for the operating system and the application. There now remains only one problem: how do we choose between them? A solution which is sometimes adopted is to "steal" the most significant bit of all virtual addresses for the purpose. If the top bit of the virtual address is set to 0 then it is a virtual address of the application process, so the application's register is used to translate such an address. But if it is set to 1, then the address is regarded as an operating system address and the operating system's PTBR/STBR is used to find the page table. This effectively means that the maximum size of programs has been halved. Figures 9.2 and 9.3 illustrate how this works for a paged system. For segmentation and paged segmentation the same principle applies.

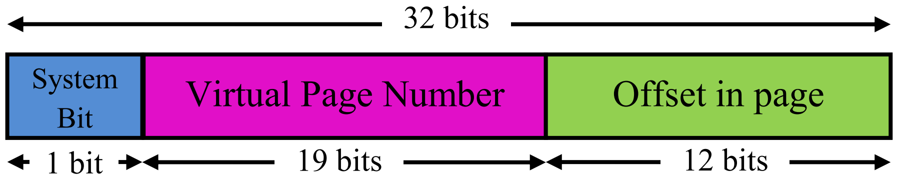


Figure 9.2: A Paged Virtual Address with Operating System Addressing

However, an application should not be permitted arbitrarily to address operating system pages or segments. This can be prevented, for example, if addresses beginning with a 1-bit can only be used only in privileged mode. Another possibility is to invalidate the system's PTBR/STBR when an application process is active. However, a different kind of solution is sometimes used in practice, involving a hierarchical protection scheme.

3 Protection Rings

This idea was first implemented in the Multics system, and has since been implemented in other systems (e.g. the ICL2900 Series). Basic protection between separate processes is achieved, as in the conventional paging model and in the simpler segmentation model, by each process having a separate address space which is controlled by the operating system switching the value in a segment table base register whenever a process switch occurs. Also as in the simpler models the operating system shares the address space of each process by using addresses with the top bit set.

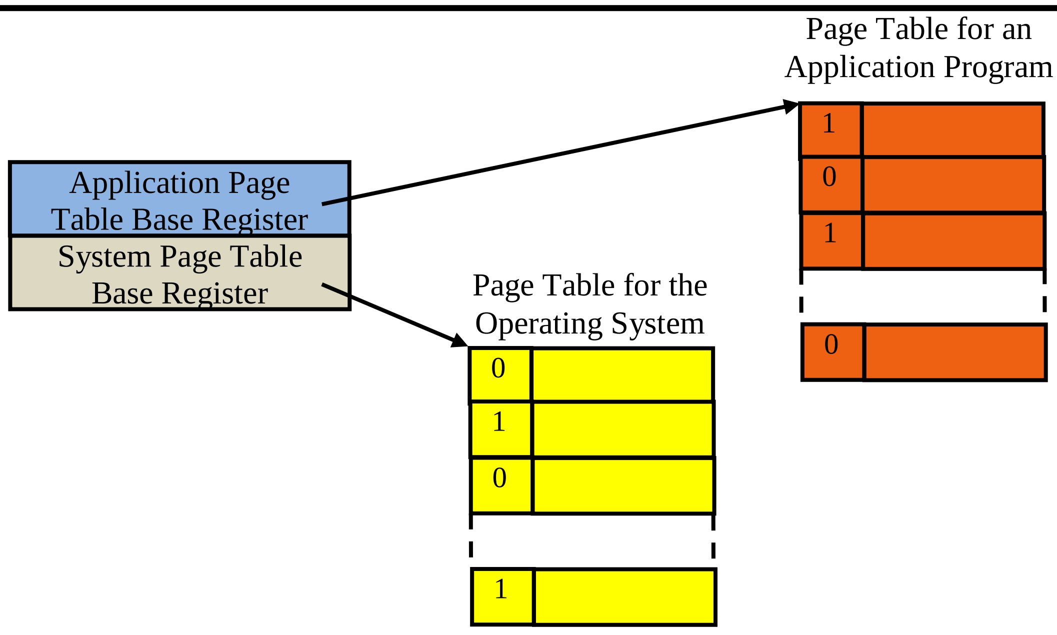

Page Table for an Application Program
Application Page Table Base Register
System Page Table Base Register
Page Table for the Operating System
1
0
1
0
0
1
0
1

Figure 9.3: Addressing the Operating System

What makes protection interesting in systems which have been influenced by Multics is the way the operating system is protected. It is based on the idea of hierarchical privilege, which can be seen as a reflection of a software system design philosophy that came into fashion in the late 1960s as a result of a paper by E. W. Dijkstra [54], sometimes known as layered design.

The ring protection mechanism assumes that an operating system is structured as a series of layers, and that a lower numbered layer is more privileged than a higher numbered layer. If we consider the hardware as layer 0 (which will shortly turn out to be advantageous in practice) the first software layer, often called the kernel, is layer 1. The number of layers varies in different systems. In the ICL2900 Series, for example, four bits are set aside for this purpose, so there may be up to 15 software layers (plus the hardware). The higher numbered layers (from 8 to 15) are not needed for the operating system and can be used to structure an application program into several layers.

The basic hierarchical protection rule is that a process executing software at layer *n* may access segments defined to belong to layer *n* and to all outer layers (i.e. those with higher numbers), but not those belonging to layer *n-1* or less. This means that the operating system need no longer be privileged as a monolithic entity. The inner level modules (i.e. those with the lower numbers) are the most crucial modules, controlling the hardware resources and managing the virtual memory tables. These are protected from accesses by the outer layers of the software. However, the outer layers are not similarly protected from the inner layers. The rationale for this lies with the rule for invoking software at different layers.

In keeping with the idea of layers as abstract machines, an outer layer may invoke the interface of inner layers, i.e. a layer may make routine calls to software at layers with lower numbers. These are sometimes called "inward calls". They must be accompanied by a change of privilege to reflect that a lower layer has been entered. The later return from the procedure back to its caller in a higher layer (an "outward return") must restore the old situation, i.e. reduce privileges.

The implementation of this ring protection scheme in the ICL2900 Series involves having a field called the *access control register* (ACR) in a protected system register. This indicates the layer at which the current process is currently executing, and information in each segment table entry shows the layer to which the segment belongs.

Entries in the segment tables in fact contain two such fields (see Figure 9.4), which replace the read permission and write permission bits found in the simpler models. The *read access key* (RAK) holds the layer number valid for read accesses to the segment. When a read operation on a word in this segment occurs, the hardware compares RAK with ACR. If RAK is greater than or equal to ACR the read access is permitted, otherwise a protection violation interrupt is caused. Similarly there is a *write access key* (WAK) which determines by a similar test whether a write access is permitted. Usually RAK and WAK have the same value for data segments which are writable, or RAK contains a layer number and WAK is set to zero (remember that the first software layer is 1), if the segment contains constants. However, other values are possible.

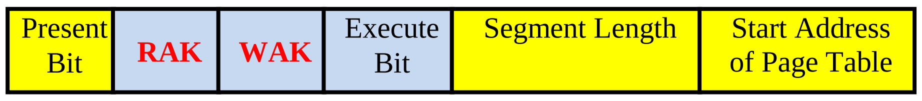


Figure 9.4: A Segment Table Entry with Ring Protection

There is also an *execute permission bit* in a segment table entry which indicates, as in simpler models, whether a segment may be executed as code. The value in ACR, which determines the current level of a process, is set by the operating system by reference to its internal tables when a system call is made.

The main software advantage of hierarchical ring protection is that it allows the operating system (and application programs) to be structured using the layering technique. However, this design methodology, like many other software designs which are based on a hierarchical concept, has some problems (see e.g. [55]) and is largely being replaced by object-oriented design techniques, which we shall discuss later. These do not map onto ring protection particularly well.

The main hardware advantage is that it allows a measure of protection for the operating system against itself, in that less privileged parts of the operating system cannot accidentally or deliberately corrupt data belonging to more privileged parts, and the application process cannot corrupt operating system segments at all. Similarly the application program can be decomposed into parts which are hierarchically protected.

But why should protection be hierarchically organised? In principle it would appear at least as sensible to organise protection so that each layer is protected fully from each other layer. One answer can be found once again in the issue of how parameters can be passed. The inward system calls, from the application to operating system routines (or from outer layers of the operating system to inner layers), are only useful if parameters can be passed, often by reference, i.e. by passing addresses of information to be accessed by the inner layer. This would not work if the layers were regarded as absolutely self-contained. In other words an operating system service routine executing with an ACR value of say 3 has to be able to access parameters in segments at levels 4 and above. We see once again that parameter passing strongly influences the protection mechanism. You will recall that this was exactly the same reason why the operating system shares the same virtual address space as user processes (using the top bit of the address to distinguish its segment table). The effect of this is that in fact an inner layer has access not only to its parameters, but to all the information held in segments of outer layers!

Apart from this logical weakness in the mechanism there are further practical problems[37] which we do not discuss in detail here.

4 Sharing

So far we have looked at memory management models which by and large assume that information in the computational memory is not shared between processes, except at the operating system level. The advantage of this is that protection between concurrently active processes in the virtual memory can in practice be achieved by placing firewalls between the processes which are extremely difficult to cross. The usual way of implementing such firewalls, as we have seen,

[37] For example, to which layer does a stack frame belong? The mechanism also opens up some problems which hackers can exploit. And it is difficult to assign an ACL level to library routines needed at several levels.

is to use context dependent virtual addresses, which are meaningful only while the corresponding process is active. With this kind of scheme it is impossible to address the virtual memory of some other process.

However, in reality such firewalls are far too restrictive, because in practice sharing in the computational memory is necessary. Consequently practical schemes usually include some tricks to allow a limited form of sharing. The most obvious example, as we have seen above, is the need for the operating system to be able to address parameters passed to it by application processes. The trick which we saw used in this case is to have two page or segment table base registers, one for the application process and one for the operating system, and using a bit in the virtual address to determine which is used. The practical effect is that the operating system and the application share the same virtual address space, thus allowing parameters to be passed without difficulty.

This trick works for the purpose intended, but it brings with it two new problems:

— How can the application process be prevented from addressing and changing operating system segments, since these are now within its range of manufacturable virtual addresses? The usual solution is the hierarchical ring protection model.

— How can the operating system be prevented from accessing sensitive information of the application process? This problem is left unsolved by the ring mechanism.

But the approach has a further disadvantage. It is a trick used only to solve a special problem. It does not solve the general problem – how software entities can be shared in a general way in the computational memory.

We now establish a need for a general model which allows software entities to cooperate by sharing segments, and we shall then see to what extent this need has been met in both conventional systems and in systems designed especially to allow controlled sharing of information.

5 Shareable Segments

The appropriate units for sharing memory are segments. If we reject the special solution which places the operating system in the virtual address space of each process, then the problem which it was designed to solve still remains. Segments containing parameters to be passed from an application process to the operating system must in the general case be possible. These are segments which are typically created by the application but must be addressable from within the operating system. Such segments, or at least references to them, should be on the application's thread stack if the in-process model is used.

A different form of shared data which we encountered in connection with thread stacks is that declared by a program in more global stack frames, which – depending on programming language scope rules – is often accessible to a procedure executing with a higher stack frame. In the more general model we supported this by means of a register which we called the G register.

On the other hand, not all information on a thread stack should be addressable to the currently active routine. For example, if an operating system's command language interpreter (CLI), or a graphical equivalent to this, invokes an application program on the same thread stack on which its own stack frames are held, these are exposed to danger in systems which give the currently active routine free access to stack frames lower down the stack.

These three issues (passing parameters to the operating system, access to more global stack frames of the current program and protection of stack frames such as those of the CLI), taken together, suggest that the decision to regard a process stack as a single segment – a decision which is found in several in-process systems – is unsatisfactory. This suggests rather that a stack should be viewed as a physical entity which contains a number of logical segments that need separate protection (the CLI problem) but which also possibly have to be shared in special ways (the parameter problem and the global frame problem).

The process stack is not the only example of a need for sharing segments in the computational memory. We have already seen that the code segments making up a program or algorithm, provided they have been compiled as re-entrant code, can be used concurrently – and therefore need to be shared concurrently – by different threads in different processes. In this case the issue even involves threads which are not explicitly cooperating or need even be aware of each other's existence!

We have also encountered a further need for sharing access to segments. The operating system, executing concurrently on different stacks, needs access to its own persistent data structures (e.g. process tables and queues, directories of files). To handle this in our general model we introduced the P register. This is another example of segments which should be shared concurrently by different processes.

It appears then that there is a general need for sharing segments (a) on the stack, (b) as off-stack code segments and (c) as off-stack data segments. The aim of this chapter is to look for a general model which flexibly allows segments to be shared where this is appropriate but which at the same time provides adequate protection where sharing is not necessary. As a starting point for this it is instructive to consider the consequences of trying to use the conventional segmentation models in a more general way to achieve sharing. We begin with the simple segmentation model.

6 Addressing Shared Segments

Figure 9.5 illustrates a possibility for sharing segments in the simple segmented virtual memory model. With this approach the operating system organizes the segment tables in such a way that segment table entries for different processes can refer to the same real segment in the main memory.

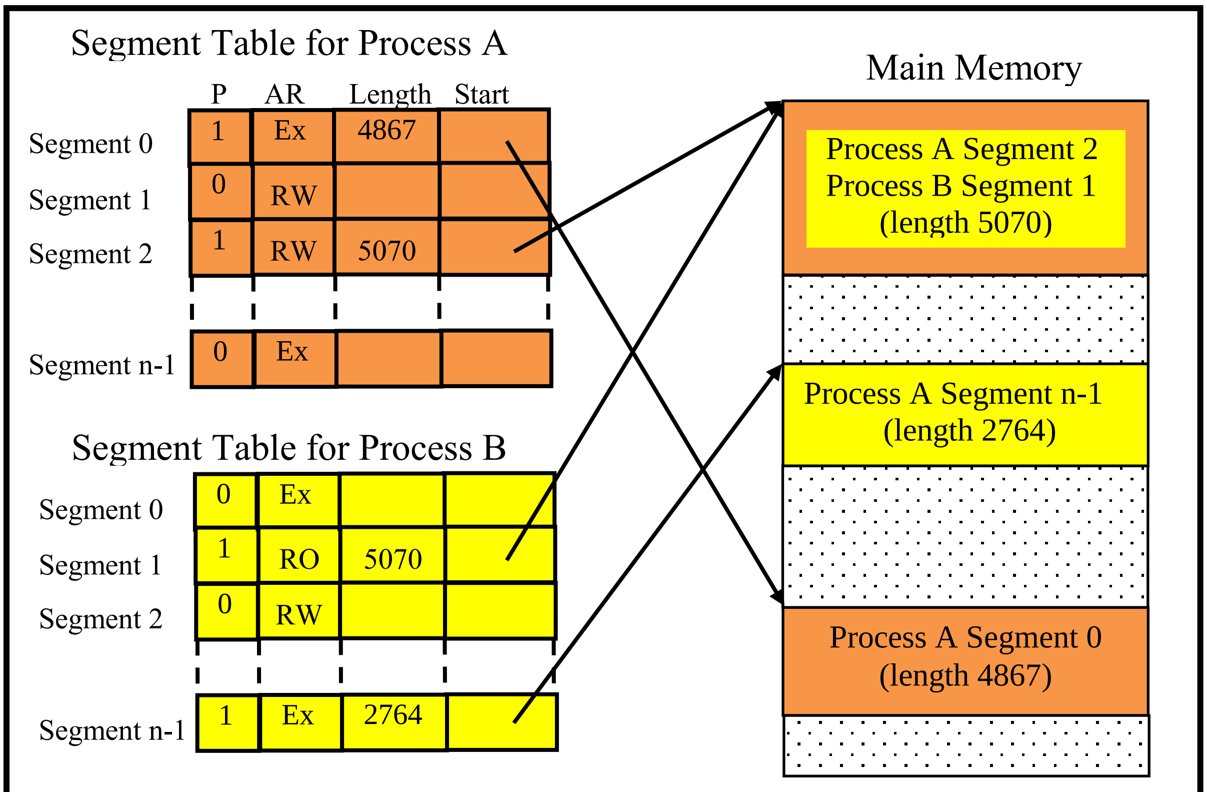


Figure 9.5: Sharing in a Segmented Virtual Memory

This approach works in simple cases, and it has one advantage. It allows different processes to have different views of a segment, for example in the form of different access rights for the shared segment. But its severe organisational disadvantages more than outweigh this advantage.

The first problem is that details of the current memory management status of a shared segment (its present bit, its address in main memory, its length) are held at the same time in several segment table entries. These must be kept consistent with each other when changes are made, which brings an undesirable organisational and run-time overhead. One possibility for reducing the consistency problem is to introduce an indirection, so that instead of the process segment table entries for shared segments containing pointers to the segment in main memory they point to a master segment table entry, perhaps held in a special shared segment table (see Figure 9.6). However this obviously creates other organisational problems, for example by making it necessary to manage the shared segment table, including allocating entries. Furthermore it involves an additional main memory access to bring an entry into the TLB.

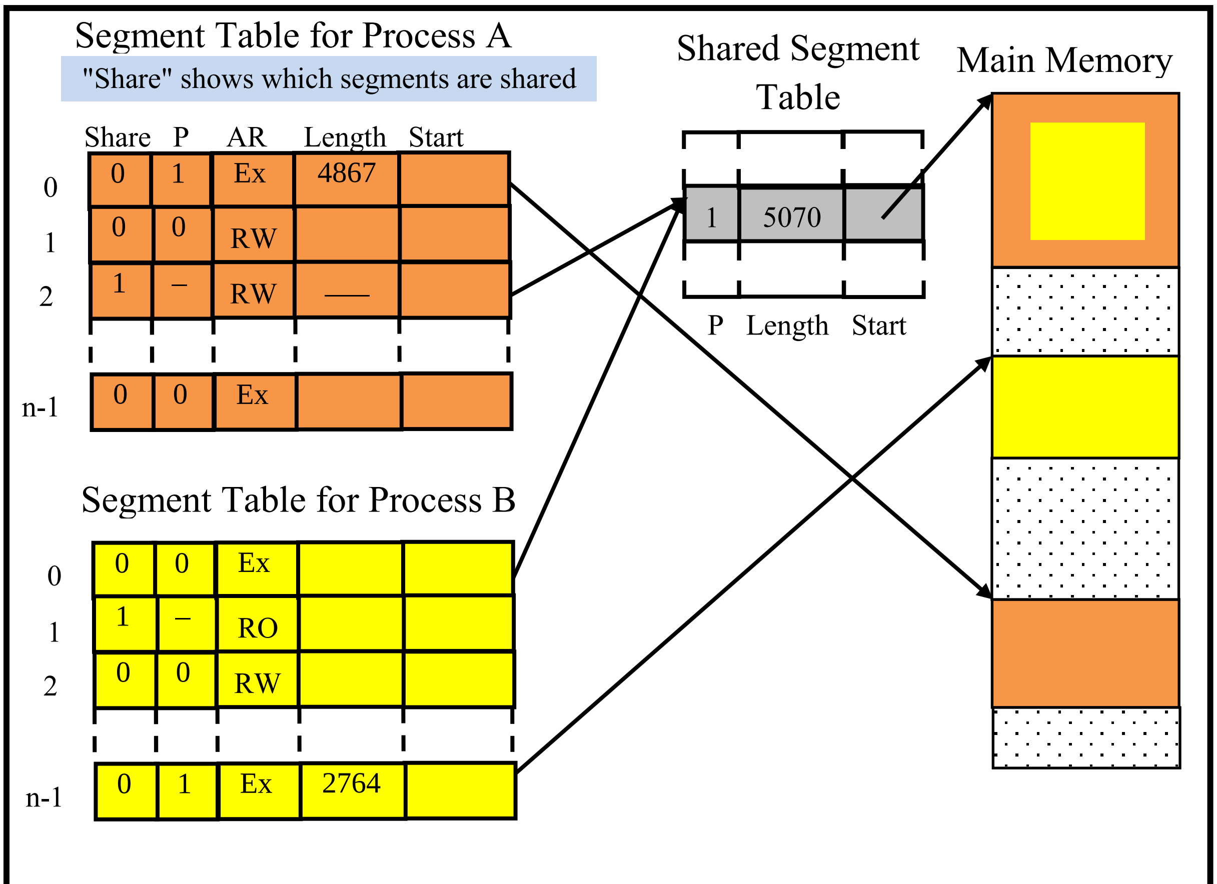


Figure 9.6: Indirection via a Shared Segment Table Entry

But there is a more serious problem than these. The difficulty arises when a shared segment itself contains a reference to another shared segment, e.g. in the implementation of a linked list. An address in one logical segment which refers to another logical segment has the usual form of a virtual address, i.e. a segment number and offset (with the offset possibly zero). What segment number do we use in such a case? Remember that each virtual address is translated in the context of the segment table of the currently active process. The problem is that the entry for a shared segment can appear at a different position in each segment table. For examples of this kind of problem, see Fabry's discussion of capability based addressing [56].

It is tempting to suggest that the problem can be avoided by organizing the segment tables for those processes which are sharing segments in such a way that they always use the same segment numbers. This would be a difficult undertaking if several independent processes are involved, but it is especially difficult if logical segments can be created and deleted dynamically by different processes. This means that they would not only have to synchronize with each other but that they also have to arrange to use the same segment table entries dynamically in each process. What happens if a segment table entry is already in use?

And there is another problem, this time involving the copying of segments, even in a single process. Suppose that an application wishes to make a copy of a complex data structure, say a linked list (consisting of two or more logical segments that are linked to each other) with the intention of producing a similar list. The linking addresses will consist of virtual addresses of the form «segment number, offset in segment». Although all the segments in the list may be copied, they must be allocated new segment numbers. But a general copy operation cannot know which parts of the segments contain references to other segments, so the result is that the old linking addresses get copied. Now the first *copied* segment is linked to the second *old* segment, and so on!

We have seen enough of this approach to the issue of sharing in segmentation schemes to recognise that it is neither efficient nor organisationally easy. This is probably one of the reasons why conventional computer architectures no longer attempt to support (and therefore protect) small segments. In the hope of finding a better general solution, we now consider how things look in systems which combine segmentation and paging in the conventional way.

7 Sharing Paged Segments

Intuitively one might think that it makes no difference to the issue of shared segments if the conventional segmentation and paging model is used rather than just the simple segmentation model. But surprisingly there is a difference.

Sharing of small logical segments is not realistic in systems such as Multics and the ICL2900 Series, because a container holding a segment at the architectural level is physically one or more pages long. Consequently a segment which is much smaller than a page (as in Burroughs systems) leaves most of the page unused. Hence architectural segments are used as a kind of container for holding collections of logical segments, with the consequence that the individual small segments are not protected from each other. Nevertheless this approach opens up the possibility of solving the last two problems which we have just been considering.

One of two situations can arise when a logical segment contains a reference to another logical segment. The linked logical segments may either be located in the same architectural segment, or they may appear in different architectural segments. In the former case, which is probably the normal case, the address linking the two logical segments need not be a full virtual address at all. It is sufficient to store the offset part of the address, with an implicit agreement between those sharing the segment that all such short addresses are interpreted as offsets within the *current segment*, whatever its segment number might be for the current thread.

This scheme has several advantages. First, it allows logical segments being shared to contain references to other logical segments, provided that these are held in the same architectural segment. This is because the different threads use their own current segment number and an internal offset to follow the pointers. Second, it allows an architectural segment to be copied (as an entire unit), leaving it still usable, without the addresses linking its internal logical segments having to be located and modified, provided of course that these linking addresses are only offsets within the architectural segment. Third, it makes such addresses shorter. (This final point is perhaps not very important in the present context, but we shall later see that it turns out to be an important advantage later.)

These advantages arise only if offsets are used as inter-segment addresses for linking logical segments which appear in the same architectural segment, and only if entire architectural segments are copied. How likely is this?

First, let us consider code segments. The compiler typically produces a file containing compiled object code. This is normally held in the file memory. It is then loaded from there into the virtual memory when it is needed. This requires an address space in the virtual memory into which the code segments can be loaded, and which can be shared by all the processes wanting to execute the same program. Usually the code of an entire program is shared, so there is no disadvantage in the individual code segments being located together in a single architectural segment, corresponding to the code file. With regard to copying, this usually takes place at the file system level, e.g. for archival purposes or to transfer the code to another computer. So the use of within-segment offsets is also appropriate.

Although in a segmented and paging scheme it is normal to regard a process (or thread) stack as a single segment, it contains many different logical units (e.g. stack frames) which can be regarded as logical segments. These are of course to a large extent interrelated.

Similarly a heap, which is an area set aside by the compiler to allocate segments dynamically, consists of related collections of logical segments which can take advantage of short pointers. Stacks and heaps, if they are copied at all, will usually also be copied as an entirety, typically for check-pointing purposes. (Check-pointing is a mechanism which involves making a copy of a process at predetermined points, known as checkpoints, so that after an error has occurred the computation can be resumed at a checkpoint rather than having to be repeated from the beginning.)

We shall return to these issues later. But meanwhile it is clear that it is quite realistic to find systems placing collections of related logical segments into architectural segments, linked by short pointers. However, this does not mean that there are no sharing problems with the conventional segmentation and paging model.

There remains a mapping problem at a higher level with respect to inter-segment links, this time at the level of architectural segments. Suppose for example that a program has been compiled and placed into an architectural segment of a process. While the process is executing this code, it decides to call a separately compiled subroutine library which is sitting in a different architectural segment of the same process. Ideally it should be possible to embed a static reference in the program to the subroutine library, but this would require that the subroutine library is loaded at the same architectural segment number in each process. This can be very difficult to arrange in a system which, for example, allows users to change their environment interactively. Instead a more costly dynamic linking mechanism has to be provided by the operating system. (This would be a much greater problem with respect to operating system segments if the operating system were not shared in each application process's virtual address space, using the addressing trick which we have seen.)

Another sharing issue with the conventional segmentation and paging scheme is the management of the page tables for shared segments. Clearly there should only be one page table for a shared segment, to avoid consistency problems. But this raises the issue how the shared page table is addressed. Should the segment table entries of the separate processes sharing an architectural segment each hold a pointer to this page table, or to a shared segment table which further points to the page table? If each process segment table entry refers directly to the page table, each has its own length field, which becomes a problem when one of the processes wishes to change the length of the segment. On the other hand if there is a shared segment table this itself becomes a management problem (e.g. with respect to the allocation of entries) and a run-time overhead (complicating the address translation logic needed for resolving TLB misses).

Furthermore all the problems discussed earlier with regard to the conventional segmentation and paging scheme, for example adequate protection for logical segments and the addressing of the operating system, still remain. In addition the one advantage has been lost which the pure segmentation model offers with respect to sharing: the support for different views of a logical segment via the different segment table entries for a shared segment.

Thus we see that neither of the conventional segmentation models (with or without paging) provide a satisfactory solution to the sharing problems, although we have clearly seen that in this respect the embedding of logical segments into architectural segments has some clear advantages over pure segmentation.

8 Conclusion

The conventional memory management models are very effective at protecting running processes from each other. By providing processes with separate paging/segmentation tables which are used to interpret virtual addresses they can very effectively ensure that an executing process cannot access the pages/segments of other executing processes. In fact this technique is so effective that it is difficult to allow processes to share information. This is like building a house without doors or windows to ensure that thieves cannot break in. But the problem then is that it is also difficult to allow friends into the house, or leaving the house oneself.

This chapter has reviewed some of the main attempts to solve this problem. We have seen how computer architects have resorted to tricks to allow sharing where it is absolutely necessary (e.g. to allow the operating system to access parameters from application processes), but it is quite evident that these tricks do not achieve the aim of providing sharing in a general way. For example the solutions which allow the operating system to access parameters from an application process in fact provide total access to the application's memory space, making it easy for an operating system to spy on the business secrets of its customers.

In the next chapter we continue the search for providing a model which allows both protection and sharing to be organised in a more satisfactory manner.

Chapter 10
Protection and Sharing in Capability Systems

The concept of a *capability* was first suggested by Dennis and van Horn in 1966 [57]. In Chapter 2 a capability was described as a unique identifier for an object together with an associated set of access rights for that object. An important feature of the concept is that it is the *possession* of a capability which implies the right to access the object in the ways defined by the access rights. This makes a capability rather like a bunch of keys which will open some of the doors in a building. The building is the object; the doors which can be opened by the keys are defined by the access rights.

Chapter 2 described the capability idea primarily in terms of file system concepts, but it is in fact a more general idea which can also be applied to addressing at the level of computer architectures, cf. especially [56, 58]. It was one of the leading ideas in the development of several operating systems in the 1970s, the most well-known of which were Hydra [59, 60] and CAL [61]. The first hardware supported capability based system was the Plessey System 250 [62], which was developed as a special purpose computer for supporting telephone networks. A further important hardware development was the CAP system, developed at Cambridge University [63, 64].

The announcement at the beginning of the 1980s of the Intel iAPX432 processor [65, 66] was greeted as the high point of hardware-based capability system development. This system was inspired both by the software implementation of capabilities in the Hydra kernel and by the aim of supporting the ADA programming language, which at that time was being very strongly pushed by the U.S. Department of Defense [67].

This combination of operating system and programming language concepts was typical of the background of capability research. For at least a decade there had been a strong association between the capability idea and high level language concepts for supporting modularity and an emerging form of what has since become known as object-oriented languages. One of the reasons for this connection was the idea that the access rights in a capability could be nicely formulated in terms of the operations on an object, as was already illustrated in Chapter 2 (see especially Figures 2.7 and 2.8). However, the emphasis on providing direct support for programming languages in the hardware resulted in poor performance and led to the virtual abandonment of the capability idea in the late 1980s.

Its demise was largely due to two related factors, the rise of the RISC (Reduced Instruction Set Computer) design philosophy [68, 69] and the breakthrough with very large scale integration of computer circuitry. IBM had been working since the mid-1970s under the leadership of John Cocke on the design of a prototype RISC computer, the IBM 801 [70], which was built using older technology. But it was the prospect of building a complete commercially viable CPU on a single chip that provided the real fuel which led to the subsequent dominance of the RISC movement in computer hardware design.

From the 1960s up to the 1980s the design of computer instruction sets had become increasingly complex. This in turn had led to unnecessary complexity in the design of the CPUs which implemented these instruction sets. There were a few exceptions, such as the CDC 6600 and its successors [71, 72], which served as a source of inspiration for the RISC movement.

The complex computer designs were in retrospect dubbed CISC (Complex Instruction Set Computers). These included most general purpose computer architectures, such as the IBM S/360 and its successors and the DEC VAX 11 series, but also computers which were specifically designed to support high level languages, such as the B6700 and Symbol [73, 74]. In fact high level computer architectures became a special target of criticism from RISC advocates [75]. In this climate the idea of capability based computers, with their links to programming language concepts and their poor performance achievements – as was amply confirmed by Intel's iAPX 432 – had no chance of surviving. In the 1980s the validity of the claims of the RISC proponents, that higher performance could be achieved by using simple and orthogonal instruction sets, were demonstrated beyond all question. A new generation of computers appeared, bringing hitherto unthinkable performance improvements. In doing so they all but destroyed the capability idea. And computer architects all but forgot protection as a research theme, which is scarcely mentioned in publications of the RISC advocates.

Nevertheless the ideas behind capability based computer systems are not unimportant for us, as these are the systems which have taken the problems of protection and controlled sharing in computers more seriously than most other systems. In the following sections we shall therefore review their most important ideas but at the same time it will be necessary to establish the reasons for their poor performance. We begin by considering how capability systems solved the sharing issue.

1 Capabilities and Sharing

In its most general form a capability consists of a unique object identifier and a set of access rights (Figure 10.1).

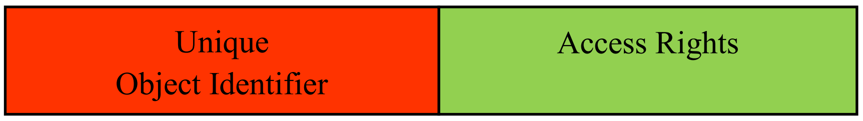


Figure 10:1 A Capability

The object identifier must contain sufficient information to locate the object, and the access rights determine which operations the possessor of the capability may carry out on the object. In later sections we shall look into such issues as protecting capabilities, locating objects and making object identifiers unique. At this point the crucial issue for us is that the object identifier *is* unique. Here "unique" means that it is non-ambiguous and context independent. This is what most distinguishes them from the other schemes which we have so far considered. The access rights determine the operations that the possessor of the capability may carry out on the object.

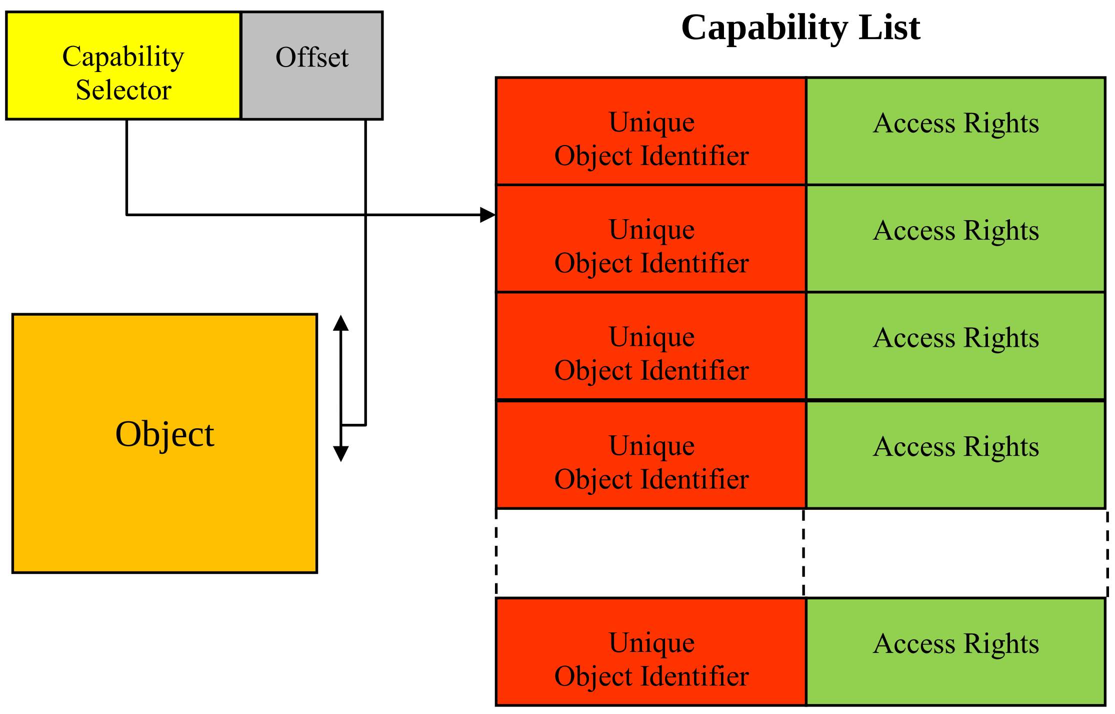


Figure 10.2: Selecting a Capability from a C-List

An object reference in a capability based addressing system sometimes consists in principle of a pair «capability selector, offset». A capability selector is normally an index into a capability list (C-List), as is shown in Figure 10.2. In its most flexible form the C-List is held not in a system table but in the application's addressing space. How such a C-List can be protected will be considered later.

The object identifier which appears in the C-List entry (i.e. in the capability) *uniquely* identifies the object. In fact there are capability systems which allow capabilities to be stored as single entities rather than in C-Lists, in which case the capability selection takes place in some other way, i.e. using a different kind of addressing mode. These are more flexible, since capabilities can then be used as freely as simple pointers. Hence the use of a capability selector as an index into a C-List is not the important thing. What is important is the capability itself, because the object identifier which it contains is unique.

The standard way of converting an object identifier into a main memory address is to have an indirection through a *central object table*, which is shared by all processes in the system. This translation process is illustrated in Figure 10.3. (As unique identifiers are usually very large the central object table is usually implemented as a hash table.) As we shall see shortly, not all objects are simply logical segments, but those relevant to machine level addressing are, and the object table entries for them are analogous to segment table entries.

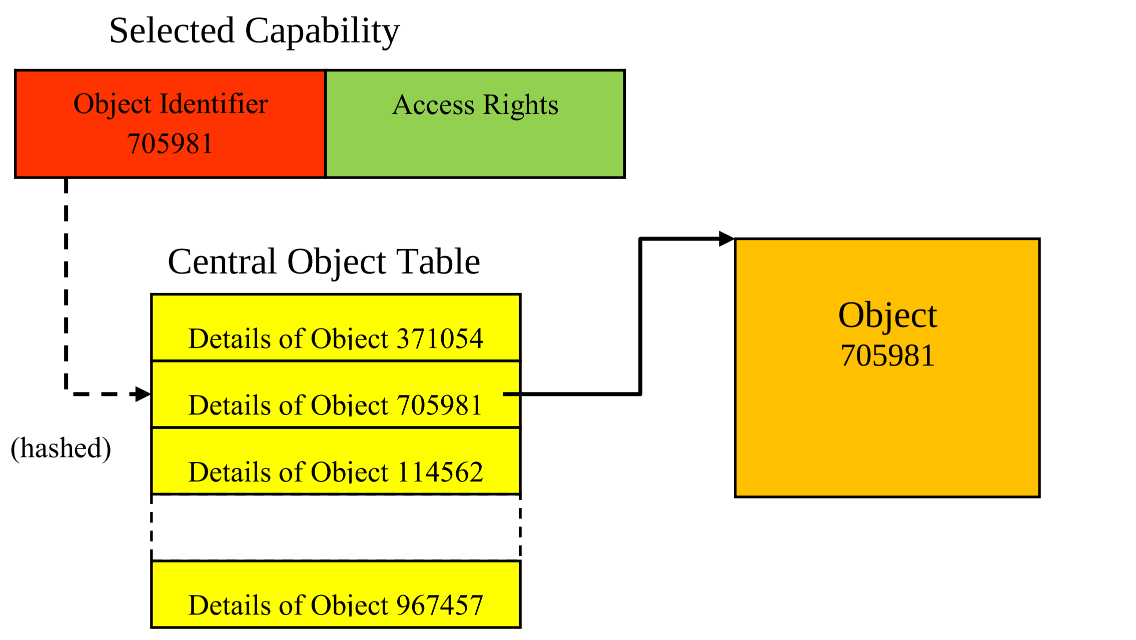


Figure 10.3: Locating an Object via a Central Object Table

In this sense the unique object identifier in a capability for a segment is equivalent to a conventional segment number, in that it is used to select a segment table entry – but remember that this object number is a *context independent* segment number. This means that it is interpreted independently of a particular process context and that it has the same meaning in every process. The consequence of this is that – unlike the conventional virtual memory models which we have considered – protection between processes/threads is not achieved by giving different meanings to the same addresses when they are used in different process contexts. Instead inter-process protection is achieved by allowing addresses to be formed only from those capabilities which the application can access. In other words, if a process or one of its threads can select a capability it can access the segment which the capability describes

This means that the context of a process/thread is defined by the capabilities which it possesses, not by its virtual memory translation table, i.e. not by the global central object table. An important advantage of this is that the sharing of segments can be naturally achieved, simply by providing each process sharing the segment with a capability for it (see Figure 10.4).

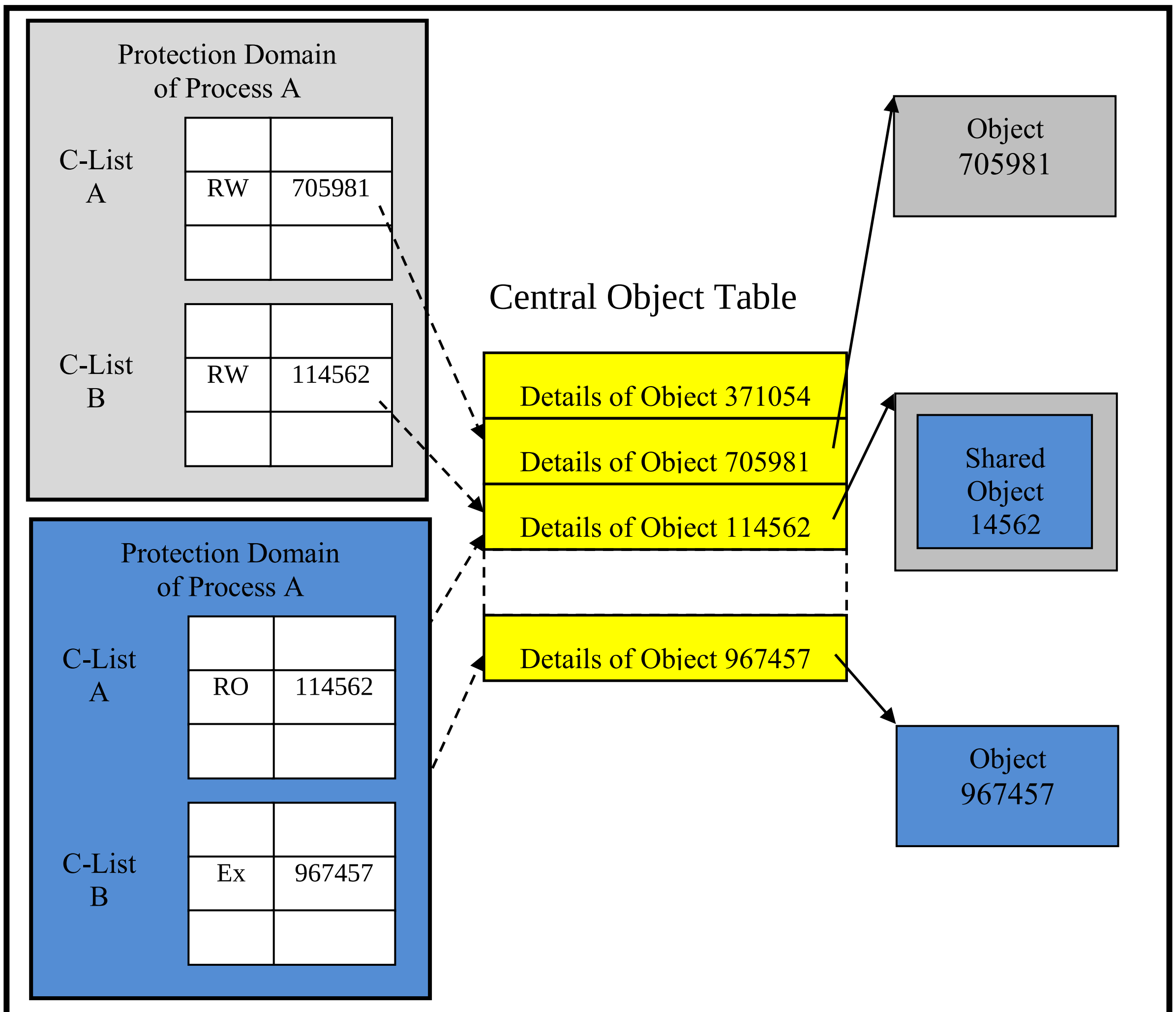


Figure 10.4: Protecting Processes in Capability Systems

This arrangement makes it possible to store inter-segment references as capabilities containing unique object identifiers. Hence the problem described in Chapter 9 (where a reference from one segment to another means different things in different processes) no longer exists in a capability based system. As was mentioned earlier, that problem is in fact just the tip of an iceberg of such problems, as Fabry has well illustrated in a paper which advocates the use of capability based addressing to solve them [56].

However this is not the whole story. There is a downside which is often not mentioned. If inter-segment addresses are always unique, which is otherwise the main strength of capability systems, then the copying problem described in Chapter 9 section 6 appears again!

2 Protecting Capabilities

One key issue in any capability based system is how the capabilities, or the C-Lists in which they are usually held, are protected. The system must in some way guarantee both that existing capabilities cannot be modified in arbitrary ways and that new capabilities cannot be arbitrarily manufactured or forged.

This implies that the system must provide special operations for creating new capabilities and for modifying existing capabilities in controlled ways, and it must oversee the right to execute such operations. But at a more basic level the system must also provide a mechanism which ensures that the normal read and write operations used for manipulating other data structures cannot be used to modify or forge capabilities. How it protects the special operations depends to a large extent on how it ensures that normal instructions cannot be used on capabilities. Various capability protection techniques have been employed in different capability based systems. We now consider the main options.

2.1 Protection in the Operating System Space

Perhaps the simplest solution is to protect capabilities in the same way as segment and page table entries are protected in conventional systems, by storing them in C-Lists in the operating system's private data space. In this case the creation and modification of capabilities can only be carried out by the operating system, at the request of users making system calls which can be checked for validity. An application indicates which capabilities it wishes to use by means of parameters in the system calls. In effect these are capability selectors indexing into the C-Lists.

The main disadvantages of this solution are a performance overhead when the user accesses his capabilities and a lack of flexibility in the way he can organize them into lists. It is rather like having a bunch of keys which is compulsorily held by a porter who will open and close your room for you whenever you ask him, but you always have to go to him to get things done.

2.2 Password Protection

A technique known as *password capabilities*, implemented in the Monash Capability System[38], allows capabilities to be stored in the user's address space while using conventional computer hardware [76]. Rather than holding a set of access rights the capability in this case contains a password, which is a large integer value (see Figure 10.5).

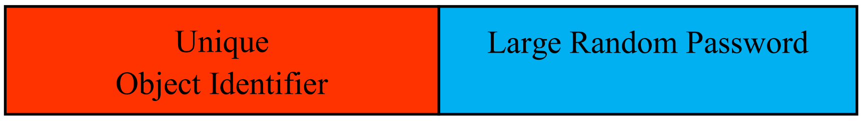


Figure 10:5 A Password Capability

This password, which in the Monash Capability System was 64 bits long, is a system generated random number. The idea is that the number must be large enough that it cannot easily be guessed or systematically generated. The capability is stored in user space and is not protected from being modified using normal instructions. (That is why it works using conventional hardware.) However, it can only be used as a capability in calls to the operating system, and at that point its validity is checked.

The operating system has an internal table in which the object name, the password and the permitted access rights are stored. When it receives a request to carry out an operation the operating system checks the object name and password fields against entries in its table. If it finds a match, the capability is valid for the access rights stored in the table. If the requested operation conforms to these access rights the operation may proceed (see Figure 10.6). Different sets of access rights have different passwords.

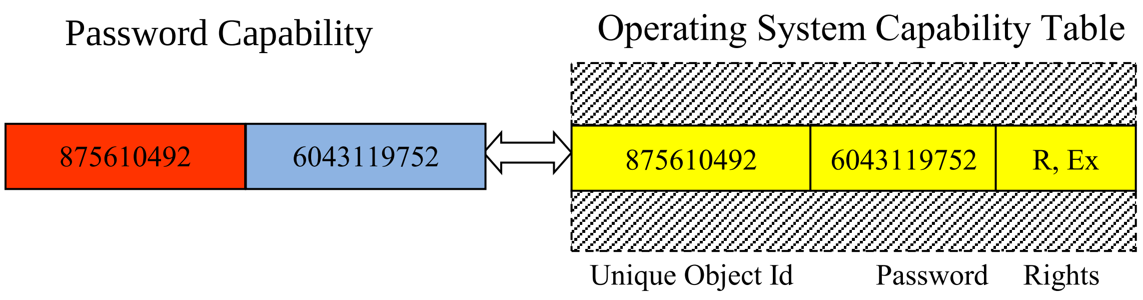


Figure 10:6 Validating a Password Capability

This solution has the advantage that capabilities may be stored flexibly in user address space. But it implies that each use of a capability must be made via an operating system call – which makes it unsuitable as an addressing mecha-

[38] Not to be confused with the Monads System, which is the forerunner of Speedos.

nism – and that there are additional tables in the operating system. (This makes it difficult for example to store capabilities on an external disc and use them later on a different computer.) It is like having a piece of paper with a password on it, which the porter checks before taking his key to open a door for you.

2.3 Protection by Tags

Another approach, which allows capabilities to be stored freely in the application address space and which avoids the problems associated with password protection, is to use tag bits to identify capabilities. This was the solution adopted in the IBM S/38 computer system [77, 78], which was a latecomer into the capability scene, as we shall see later. The tagging solution involves having an additional bit or bits associated with each word in the main memory. If these hidden tag bits are set to 0 the remainder of the word is a normal data or instruction word, but if it is set to 1 the rest of the word is part of a capability. Because a capability might take up several words of memory, the address at which it actually begins can be recognized by its byte position, i.e. capabilities must start at fixed byte positions. The tag bits can only be set and unset when the system is in privileged mode.

With this solution the CPU can check, when it is executing instructions, whether these are being applied to capabilities or to normal data and instruction words. To create a new capability the normal user must call the operating system because he cannot set the tag bits. Similarly attempts to modify a capability using normal instructions will be detected by the hardware.

More recent attempts to build secure systems have also used variants of this approach, e.g. CHERI [79], Mondrian memory protection [80, 81], Hardbound [82], Intel's iMPX Memory Protection Extension[39] and the M-Machine [83].

Tagged capability protection is more flexible and far more efficient than the earlier solutions, since there are no tables in the operating system, but it is achieved at the cost of extra bits of memory for each word. What is particularly unfortunate is that these extra bits have to be copied to disc whenever a program segment is discarded from the main memory to make room for another. Disc blocks are usually organized into sizes which are powers of 2, which creates difficulties when words in memory have extra tag bits. This is rather like having to use keys which are too big to fit into your pocket.

[39] According to the Wikipedia article https://en.wikipedia.org/wiki/Intel_MPX "Intel MPX claimed to enhance security to software by checking pointer references whose normal compile-time intentions are maliciously exploited at runtime due to buffer overflows. In practice, there have been too many flaws discovered in the design for it to be useful, and support has been deprecated or removed from most compilers and operating systems. Intel has listed MPX as removed in 2019..."

2.4 Protection by Partitioning Segments

A fourth solution is the use of *partitioned segments* [84]. Here the idea is to allow normal data and capabilities to coexist in a single segment, but to segregate them into two parts of the segment. Normal data words are addressed by positive offsets from a base register, while capabilities exist at negative addresses, below the address at which the base register is pointing. The instruction set of the computer is so organized that negative addresses either cause an exception into the operating system, which can then validate the action required in relation to the capability, or that special capability instructions which use only negative offsets can be supported at the architectural level (see Figure 10.7).

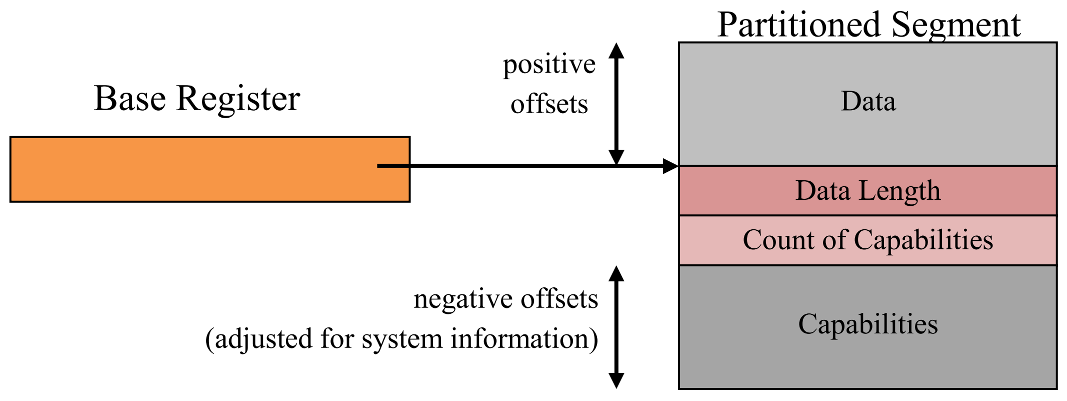


Figure 10:7 Partitioned Segments

The use of partitioned segments makes it easy to organize linked data structures in the application address space using capabilities as the links, and it has none of the disadvantages of the other solutions. There are no operating system tables or hardware tags. In this case the keys fit conveniently into your pocket and you can put them away as a bunch in a convenient filing cabinet or pocket, wherever is convenient.

2.5 Protecting Capabilities via Capabilities

Finally, there is another solution for protecting capabilities: use capabilities to protect other capabilities. This relies on the fact that another solution already exists, so it may seem to contain a circular argument. But if, as we shall later argue, there should be more than one kind of capability, then this solution also makes sense. This is like being able to lock a box containing your keys for safe keeping.

3 Unique Object Identifiers

In order to make the capability technique work, it is essential that object identifiers can be made unique. It might at first appear that there is a simple way to make them unique, by prefixing them with a process/thread number. But then there is a problem with sharing objects between processes: which process prefix does a shared object have? As it is an important aim of capability systems to allow sharing between processes, this solution is clearly unsatisfactory.

What earlier capability systems therefore did was to allocate object identifiers independently of process numbers. A simple way of doing this is to have a counter which is incremented by one each time a new object is created. The counter must be large enough count up to the maximum number of objects.

In order to determine how large such a counter must be, we must first have a clear notion what an object is. So far we have more or less assumed that it is a segment. But this was not the answer in the classical capability systems, at least not the whole answer. An object in these systems is anything which might be potentially shared between processes. If we were to take this to its absolute extreme, we would have to say that *any* variable (e.g. an integer or boolean variable) can be shared between processes. This would mean that every small variable in the system would need a unique identifier. Advocates of capability systems did not go quite this far. But at the next level up – the logical segment – there is a stronger case. A segment might for example contain a routine, which is a good candidate for being shared by many processes/threads. Or it might contain a data record, which can also be usefully shared between processes/threads.

In fact sharing segments also allows for the possibility of sharing smaller entities if necessary, since a segment might simply contain one word of memory. There is at least one kind of very small object which only makes sense when it is shared, viz. a synchronisation variable, which explicitly exists to allow controlled sharing.

On the other hand not all shareable objects are simple segments. A shareable object might also be a composite object (e.g. a program, which consists of many procedures and constant segments, or an abstract data structure which consists of a data segment and the code segments which access it). Since such larger objects can be shared in their entirety they too each need an object identifier in a classical capability system.

If objects as small as logical segments can be shared it is clear that capability systems must be prepared to handle a large number of small objects which need unique identifiers, so the counter which is used to allocate object numbers must be large. It also has to be large for another reason. The problem of uniqueness is not just a question of how many objects exist at a particular time, but how many might exist over the lifetime of the system. So we have to reckon with very large object identifiers, for example implemented using 64 bit numbers. (A 64 bit object space allows for up to 2^{64} objects, which is rather more than 16×10^{30} or 16,000,000,000,000,000,000,000,000,000,000 objects.)

If in terms of addressing segments we have addresses which consist of a pair «unique object identifier, offset in object» (by analogy with «segment number, offset in segment»), these enormous addresses somehow have to be translated into main memory addresses. As mentioned above, the classical capability systems used a central object table for this purpose. But unlike conventional paging and segment tables this cannot be a table indexed by unique object number, because if would be far too large to fit into any memory (whether main or virtual), since an indexed table needs an entry for each of the 2^{64} possible index values. Fortunately such a long table is not needed in practice, because most of its entries would not contain useful information for the address translation process, as most of the possible segment numbers would not map to existing segments. So another technique had to be used.

The usual technique in capability systems, as in most situations where an entry has to be found quickly in a large sparse name space, is hashing. Some of the bits of the object number are used as an index into a much smaller table, and the remaining bits are treated as a tag (not in te sense discussed in the previous section) which is placed in the entry indexed via the index part. If the tag matches then the required entry has been found. Otherwise an overflow technique must be used to find the real entry somewhere else in the table.

But then another problem was encountered in realistic capability systems. The number of entries in the object table at any time (i.e. for existing objects) was usually too large to allow the complete table (even implemented as a hash table with overflow) to be permanently held in the main memory. Consequently some kind of special mechanism was needed to allow parts of the object table to be held on disc. Furthermore, the management of the table was made more difficult by the fact that so many entries had to be dynamically created and deleted at very frequent intervals.

A further source of inefficiency arose from the fact that entries for both segments and for composite objects were held in the same table, with the result that these could hold quite different items of information. For example an entry for a segment would need to look like a segment table entry in a conventional system (with present bit, start address, etc.), while an entry for a composite object would need to hold quite different information about its structure and its component parts.

None of these design problems harmonises well with fast and efficient address translation and main memory accesses. But on top of this there is the general point that the use of very wide virtual addresses considerably increases the cost and affects the efficiency of a Translation Lookaside Buffer, if the system has one at all. (The data and instruction cache(s) need not be affected, if the system caches these on the basis of main memory addresses.)

Then there is a reliability question. The object table is a system-wide table, the correctness of which is essential to the reliability (and protection) of the entire system. In contrast conventional virtual memory schemes have separate tables for each process, so that a corruption in one table does not adversely affect the reliability or security of other processes (unless the operating system's page table goes wrong). It is not even easy to take a checkpoint of a single process, because the table contains entries for all processes.

Some of these implementation problems have been discussed in more detail elsewhere [85]. Apart from the memory management problems, which were similar to those in more conventional systems aimed at supporting the sharing of individual logical segments, the main additional problems in capability systems were (a) the use of a single central table, (b) the decision to manage different kinds of objects which have quite different properties as though they are all comparable objects organized in a single table, and (c) allowing the addressing of items in the main memory to be affected by this table. Together these problems accounted for many of the performance problems in capability based systems.

There is one final problem with the uniqueness of names. In the 1970s most computer systems were independent of each other, but in today's world computers are, or can be, linked together via local area networks and over the worldwide Internet. For this reason the related themes of security, protection and sharing have become ever more important. But if the vehicle to be used to implement these is uniqueness of object identifiers, then even 64 bit numbers are not large enough! Try to imagine not only how many logical segments exist at any one time on all computers linked into the Internet, but how many *might* exist into the distant future! Then there is the question, even if we use numbers which are large enough, how the idea of a central object table could be implemented worldwide, or how the unique identifiers can be allocated. For the present we leave such difficult questions aside, and return to our main theme.

4 Conclusion

This chapter has reviewed the key features of capability systems. These provide a quite different approach and insight into the issues of protection and sharing. But, like the more conventional approaches discussed in chapter 9, capability based systems have a number of drawbacks. However, these are mainly issues which are associated with efficiency, rather than with basic principles.

Part 3
A Memory Structure for SPEEDOS

Chapter 11
An Architectural Basis for SPEEDOS

The question of how to combine segments with paging was not adequately solved by the conventional paged segment model described in Chapter 7. Although variants of it have influenced the thinking of computer manufacturers even until the present time, there has remained a feeling of dissatisfaction because it implies either that logical segments (i.e. segments as seen by compilers, which are typically considerably smaller than a page) have to be abandoned or that they must incur a high penalty in internal fragmentation. For SPEEDOS, as will become evident in later chapters, a solution which allows individually protected segments, regardless whether small or large, to be efficiently paged, is very important. Consequently the issue is pursued further in this chapter, beginning with attempts by others to solve the problem.

1 Combining Segmentation and Paging Efficiently

This section discusses the most significant attempts to solve the problem.

1.1 Multiple Page Sizes

Perhaps the simplest idea is to support more than one page size. This idea was already implemented in the General Electric 645 architecture, which was used as the basis for Multics. In its hardware design two page sizes were available, a larger page size of 1024 words and a smaller size of 64 words. In such a system it is possible to use the small page size for small segments while the larger page size is used to decompose a large segment, with the last part possibly being placed in a small page or pages.

But this kind of scheme has two problems. First it complicates memory management and virtual memory tables compared with the simplicity of a single page size. Second, it is difficult to know how small a small page should be. Professor Brian Randell of the University of Newcastle-upon-Tyne took this idea a stage further by proposing a scheme which flexibly allowed multiple page sizes [86]. But the more page sizes there are the more complex the memory management becomes.

1.2 Segmented Pages

A different possibility is to switch around the relationship between segments and pages, by having a virtual address consisting of the triple «page number, segment number, offset in segment», such that instead of a segment being decomposed into pages, a page is decomposed into segments. But this is not really satisfactory, since it only copes with small segments, and it can still lead to substantial fragmentation if segment sizes happen not to be convenient. Furthermore it adds to the work of the compiler, since the latter has to be concerned with how to fit segments into pages.

For a long time it seemed to be impossible to fully reconcile all the requirements of compiler writers, of programmers and of the operating system. Let us now look at the main requirements again, from these different viewpoints.

2 A Review of the Requirements

Compiler writers work with logical segments. Ideally they would like to have addressing modes which express addresses in terms of logical segments and offsets. This means that they do not have to have a phase at the end of the compilation in which memory management considerations (such as linearizing the addresses in the program or placing segments with similar properties together). In addition they would like hardware bounds checking on the offsets for these logical segments, in order to save the generation of code for achieving this, especially in cases where the lengths of logical segments cannot be determined at compile-time.

Depending on the language being compiled, compiler writers sometimes have another requirement, which we have not yet discussed. That is to have different *views* of a logical segment. To illustrate this, consider the kind of routine in Timor [87] called an *enquiry*, which is a method that returns information about an object while guaranteeing that the state of the object (represented in logical data segments) is not modified. This may seem to be a trivial thing which can be checked at compile time, but unfortunately that is not always possible. What it implies is that a data segment be viewed by some processes as a writable segment but by others as a segment of constants. Even the segmentation schemes which put protection bits in the segment tables cannot handle this requirement in a reasonable way.

What requirements does the programmer have? An important consideration is that his errors are detected as soon as possible, which means that he would like to have protection at the logical segment level in terms of basic access rights and in terms of bounds checking. In so far as this involves runtime checks, he would prefer these to be carried out by hardware rather than by instructions inserted into his program by the compiler, as that results in his program executing more quickly (and the compiler can also compile his program more quickly).

The operating system designer has quite different requirements. He wants to be able to view a program as a set of pages, without the logical structure interfering with memory management. He wants this to be as simple as possible, for example using only a single page size. He would also like the use and change bits to help with his memory management problem. And he would like to have a simple and consistent protection scheme which is uniform and easily organized. And from the protection viewpoint he wants flexible support for segments with differing contents.

Finally, by far the most flexible implementation technique for capability-based systems is the use of partitioned segments, which allow a (typically small) segment to be partitioned such that the application process can have direct access to the data partition, while the pointer partition remains protected.

It may seem a tall order to achieve all of these requirements in a single memory management model, but it is not impossible. In 1980 the author published a paper outlining a memory management model which met all of these requirements [88]. The essentials of this model are now presented.

3 Orthogonal Segments and Pages

When problems become complicated the reason is often that things which are not really related to each other are being mixed up together. This is what has happened with the ideas described earlier for combining segmentation and paging. In particular it has been assumed that the virtual address must in some way be reducible into a part which describes segments and another part which describes pages. This approach inevitably implies that segments are decomposed into pages (an assumption which does not work out well for small segments) or that pages must be decomposed into segments (which is just as bad for large segments). In practice there are certainly more small segments in most programs than there are large segments, but large segments cannot be ignored. What happens if we try to keep the two ideas as separate as possible?

We start with the idea that the address which a compiler wants to produce is a two part address, which we shall initially think of as the pair «segment number, offset». This is an *effective program address*, which says nothing about page boundaries and nothing about how the offset is derived[40].

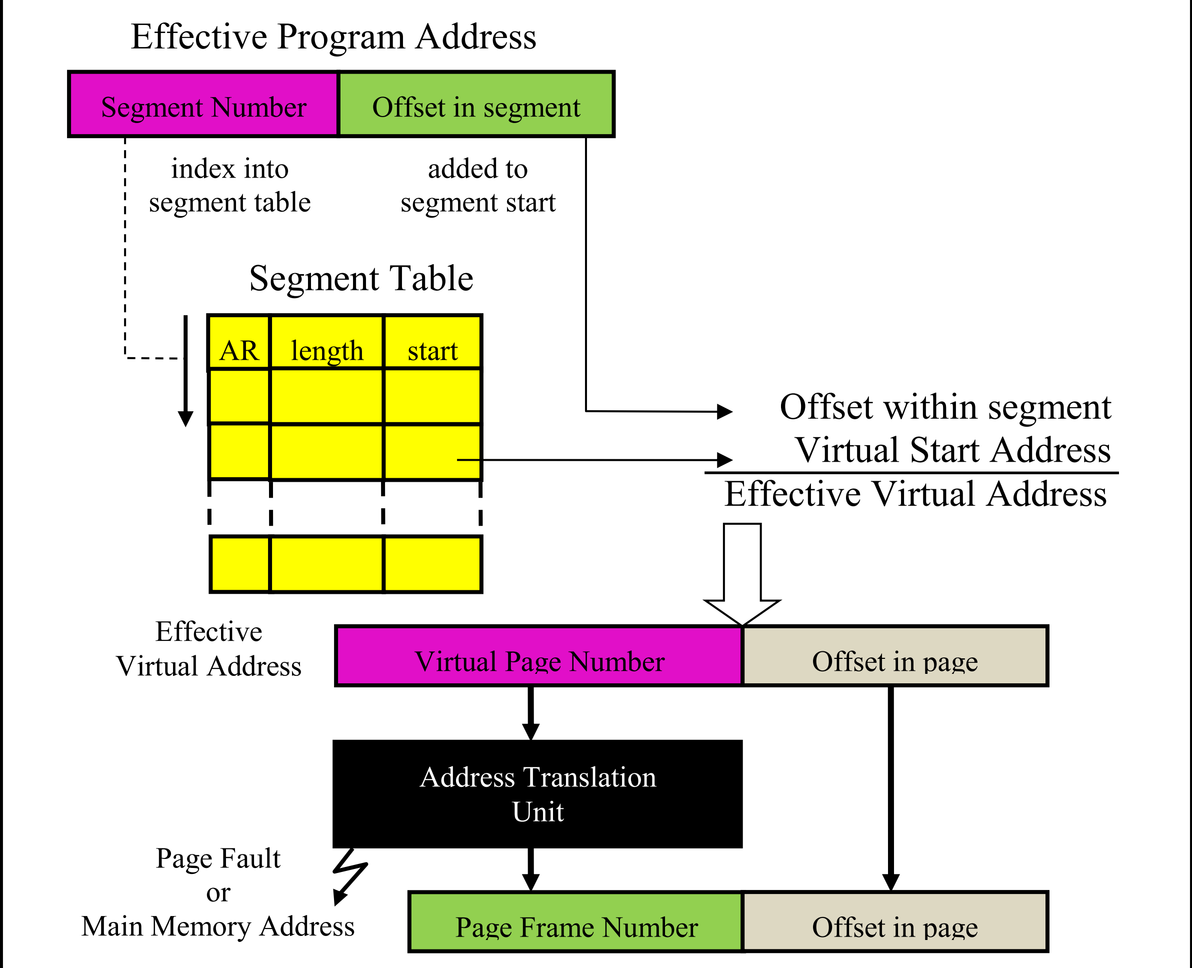


Figure 11.1: Translating an Effective Program Address in the Orthogonal Paging and Segmentation Model

This kind of address can be translated by reference to a segment table (Figure 11.1). Thus far we appear to be following the pure segmentation model discussed at the beginning of the chapter. But here is the twist. Instead of regarding the entry in a segment table as containing either a main memory address or the address of a page table, we simply assume that it contains a *virtual address*. This means that we are now drawing a distinction between effective program addresses and virtual addresses (see Figure 11.1). (Note: In the earlier (conventional) models, the virtual address was at the same time an effective program address.)

From the segment table we acquire a virtual address defining where a segment begins. To this is added the offset from the beginning of the segment, taken from the effective program address. This gives us another address – this time the *effective virtual address* of the word we wish to address. It is this address

[40] The latter is a question for the instruction set's addressing modes (e.g. by using index registers and/or literal offsets).

which has to be translated into a main memory address. As we want a paged virtual memory, this translation can be achieved in one of the usual ways, by using either a conventional page table or an inverted page table. This is illustrated in Figure 11.1.

Notice that the segment table can contain information about the logical properties of the segment (i.e. its length and its access rights), while the ATU can hold information useful for paging, such as a use bit and a change bit.

Let us now consider what this means for the layout of a program. First we consider a simple program which contains all three kinds of segments. Figure 11.2 shows how the segment table and the program both appear. We see from this that there is no difficulty in placing segments of different types adjacent to each other. We also see that a segment can span multiple pages but also that multiple segments can be placed in a single page, in any arbitrary combination.

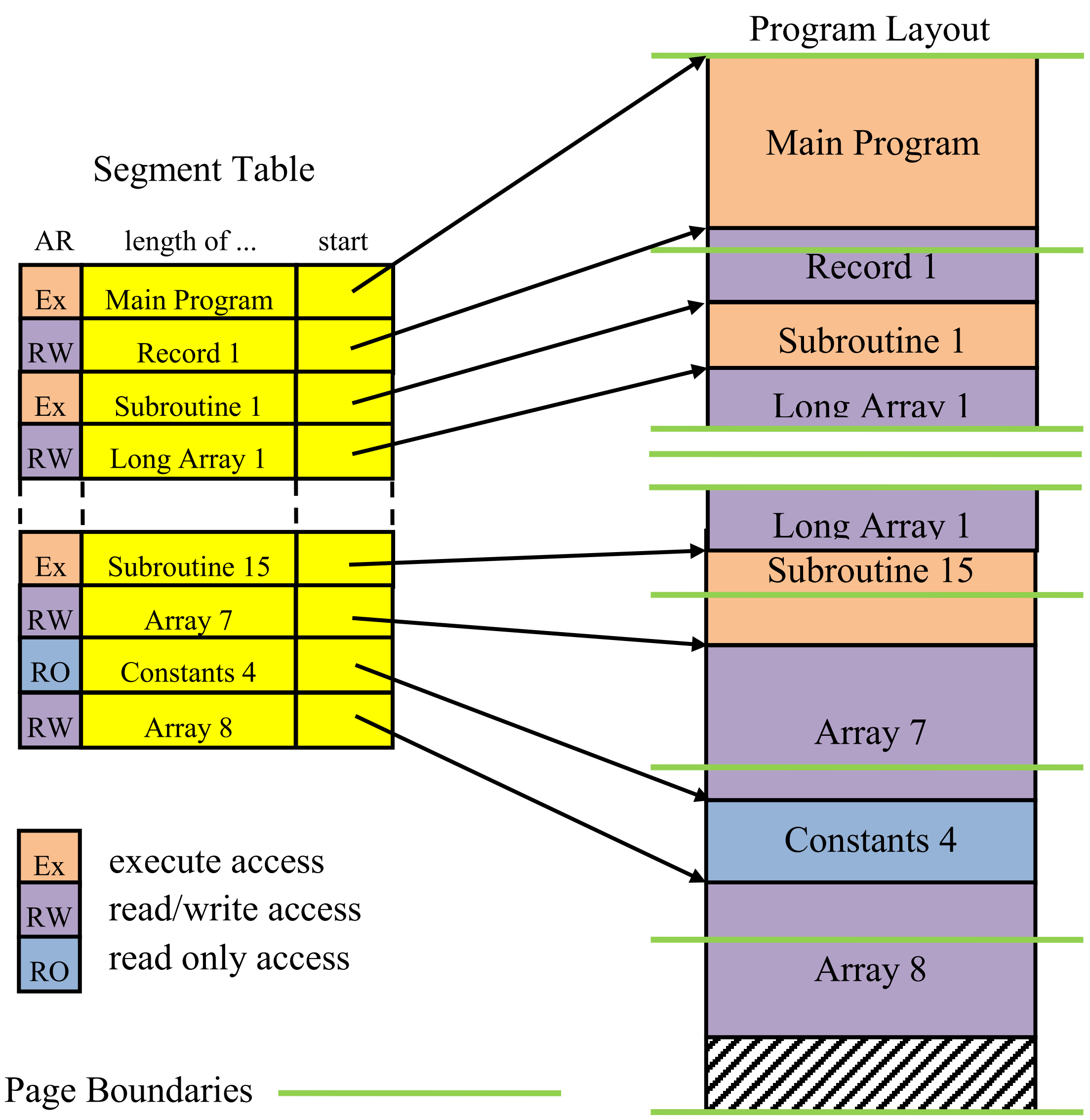


Figure 11.2: A Program Decomposed into Segments and Pages

Furthermore, the problem of internal fragmentation has been restricted to the final page. Recall that this is a serious disadvantage of the conventional segmentation and paging model, assuming that small segments are considered relevant. Like the conventional paging model, the orthogonal model achieves the minimum possible fragmentation of a half page per program on average.

If a conventional page table is used, only one such table per program is required (rather than one per segment). But as the diagram in Figure 11.1 (cf. Figure 7.4) makes clear, conventional page tables need not be used at all. It is equally possible to use inverted page tables with this model.

The key feature of the orthogonal model is that it makes paging and segmentation independent of each other. It allows a number of logical segments to be placed contiguously in the same page. Whereas the conventional segmentation and paging model treats all logical segments in an architectural segment as sharing the same access rights and does not provide information about the length of logical segments, the orthogonal model allows each logical segment to have its own access rights and length information, although it does not define how this information is stored.

At this point it is useful to introduce a new term to define the entity which in the orthogonal model is paged and which holds a group of related segments; we call this a *container*[41]. It can be considered comparable to an architectural segment in the conventional model except that this does not have – and does not need – associated protection information used directly by the hardware.

4 Implementing the "Segment Table"

A significant feature of the orthogonal model, which will play an important role in our emerging protection model, is the assumption that protection information is provided to the hardware only when an instruction is being executed. As described in Figure 11.1 this information is derived from a segment table, the entries of which each contain a segment start address, a segment length and access rights for the segment.

However, a number of advantages are gained if we now redefine the segment table simply as a bank of *segment registers*[42] in the ALU, which can only be loaded by kernel instructions.

The first advantage is that this provides the software with considerable freedom by leaving open how the structures from which information is loaded

41 In the MONADS literature, which successfully implemented the orthogonal model, this was called an *address space*.

42 In the MONADS literature these were called *capability registers*. Another suitable name would be *address registers* or *addressing registers*.

into the segment registers can be defined. One possibility is of course a segment table along the lines of that in the simple segmentation model, but another is that the special instructions use information based on the partitioned segment idea which we described in connection with the implementation of capabilities in Chapter 10. In fact at the virtual memory level there is no need to restrict the implementation to any particular technique, provided that at the kernel level of the system the mechanisms used are secure.

The second advantage is that ALU registers can be accessed far more quickly than information in the main or virtual memory (e.g. segment tables).

The third advantage is that this technique fairly closely corresponds to the fact that addressing in conventional computers is often defined as offsets from general purpose ALU registers, though the latter have the comparative disadvantage that they do not contain length or access rights information.

Each segment register has the format shown in Figure 11.3. A valid bit indicates whether the register's contents are currently valid. The segment registers are implemented as *dedicated* registers. These are registers which, in contrast with general purpose registers, can be used only for the purpose of addressing and can only be used as operands in certain instructions. This is important because the integrity and security of the system depend on the correct information being loaded into segment registers and on the hardware only using them for the purpose to which they are dedicated, i.e. protected addressing.

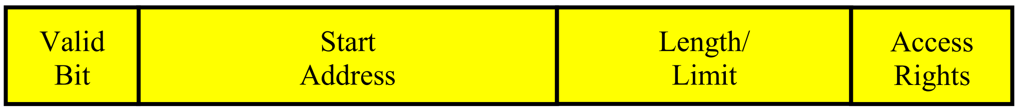


Figure 11.3: A Segment Register

Finally, it is worth mentioning that using dedicated registers for addressing is not a new idea, and need not compromise the efficiency of a system. For many years the world's fastest computers were designed by Seymour Cray, initially at Control Data Corporation (e.g. the CDC6600 supercomputer [71]) and later by his own company (e.g. the CRAY-1 [89, 90][43].). These computers were designed with registers which were dedicated to holding addresses. For example the first of these (the CDC6600 [72]) had 8 address registers. However, they did not include the security features which we now envisage, namely length information and access rights.

In the MONADS Project [91, 92, 93, 94][44], a major project which the author established at Monash University in Australia in 1976, the technique de-

[43] The second reference for the CRAY-1 is a reprint of the first.

[44] see http://www.monads-security.org, where more publications are listed.

scribed here (including the protection aspects) was successfully used, especially in the MONADS II and in the MONADS-PC systems; these systems successfully tested the ideas in a non-RISC environment.

5 Segment Registers and RISC Systems

Chapter 10 described how the RISC movement, which became a serious force in the field of computer architecture in the early 1980s, virtually killed experimentation in the area of capability-based computer architectures, largely because of their association with CISC architectures which were sometimes designed to support high level languages. In doing so the RISC movement in effect killed experimentation in architecturally based security, although this was almost certainly not intended by the RISC proponents.

It is therefore especially interesting to note that in their discussion of protection in the 5th edition of their standard textbook on computer architecture two leading advocates of RISC, John Hennessy and David Patterson, wrote:

> "Security and privacy are two of the most vexing challenges for information technology in 2011. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Hence, both researchers and practitioners are looking for new ways to make computing systems more secure. Although protecting information is not limited to hardware, in our view real security and privacy will likely involve innovation in computer architecture as well as in system software." [8, p. 105]

I believe that this sought after innovation in computer architecture can be provided by the orthogonal paging and segmentation model using the segment register implementation described above. This combination, referred to as S-RISC (as an abbreviation for "secure RISC"), is now presented in the context of RISC architectures.

5.1 Paging

RISC systems are based on paging. This fits well with the orthogonal model, which also uses paging as its basic virtual memory management technique.

5.2 Segmentation

RISC systems provide no architectural support for segmentation. This is where the innovation occurs. However, it is important to introduce segmentation in a form which corresponds to the RISC philosophy.

5.3 RISC Philosophy

RISC systems achieve their very high performance primarily as a consequence of implementing very simple instruction sets. Instructions in RISC systems normally have a single fixed length and the formats of instructions are simple and consistent. Instructions which process data do so in general purpose registers, although there is usually also a separate set of floating point registers. A large number of general purpose registers (usually 32) is available. A typical ALU (computational) instruction uses three registers: a result register and either two source registers or a source register and an immediate value (see Figure 11.4).

Operation Type	Result Register #	Operand 1 Register #	Operand 2 Register #

Operation Type	Result Register #	Operand 1 Register #	immediate value

Figure 11.4: Typical RISC ALU Instruction Formats

Information is never processed in conjunction with main memory accesses (in contrast with many CISC systems). Instead there are special *load and store* instructions for copying information between registers and memory. The addressing modes used in these instructions are simple. The address held in a base register is added to an offset held in an immediate field or in an index register to produce an *effective address*, which is then used as the load or store address (Figure 11.5).

Load/Store Operation	Base Register #	Operand Register #	Index Register #

Load/Store Operation	Base Register #	Operand Register #	offset

Figure 11.5: Typical RISC Load/Store Instruction Formats

5.4 The Proposed Innovation

The proposed innovation simply replaces the base register on Figure 11.5 (which in normal RISC systems is a general purpose register) with a segment register (Figure 11.3). The execution of a load or store instruction involves not only calculating an effective address (using the start address as a base address) but also checking that this is in the range of permitted addresses (i.e. within the segment). The segment length field is compared with the offset field or index register value in Figure 11.5. This comparison can be carried out in parallel with the generation of an effective address. Similarly the access rights field can be checked in parallel with the calculation of the effective address. Hence the speed of execution need not exceed that of a normal load or store instruction, given appropriate hardware. Hence the security advantages which we will illustrate in later chapters can be obtained with RISC efficiency.

Similar considerations apply to the addressing of code. In this case a single register, which we call the Code Segment Register, is needed. This defines the bounds of the current code segment and has an access rights field which will typically be set to allow code execution and the reading of (constant) data. This is complemented by a Program Counter (PC) register, which plays a role equivalent to index registers for the data segment registers.

6 Implementing Address Translation

An inverted page table was successfully implemented fully in hardware in the MONADS systems [95, 91, 96], using a hashing technique with overflow. This allowed us to take advantage of the fact that the structure of the underlying page tables could be kept flexible. However the technique used cannot reasonably be scaled up to support large modern main memories. Nevertheless, a similar effect can be achieved in modern computers by using the ATU technique implemented in the DEC Alpha computers [97] and in other RISC computers, where the entire ATU consists simply of a translation lookaside buffer (TLB). Figure 7.8 (repeated here as Figure 11.6) helps to explain how this works. This shows how in earlier systems a TLB was first accessed. If the required information was not found in the TLB the hardware then searched the appropriate page table and then loaded the required information into the TLB.

With the S-RISC scheme now being described, the ATU is reduced simply to the TLB, which can be loaded by kernel software. Consequently the algorithm in the white box in Figure 11.6 is relegated to the kernel software, as is illustrated in Figure 11.7, where the blue box represents kernel software.

With this implementation, which fits well with the RISC philosophy, we retain the main advantage of the MONADS systems' solution, i.e. that the structure of page tables is of no concern to the hardware, but can be flexibly handled in (kernel) software.

7 Conclusion

In this chapter an alternative model for combining paging and segmentation has been presented, which allows small and large segments to coexist in the same container with full protection both in terms of access rights and bounds checking. This model fulfils all the requirements set out earlier in the chapter. But at least as significant, it can be integrated into the RISC concept and hence does not fall into the category of "inefficient" security. On the contrary, it demonstrates that security and efficiency are not inimical concepts. Henceforth it is assumed that the design ideas presented in following chapters are implemented on an S-RISC system.

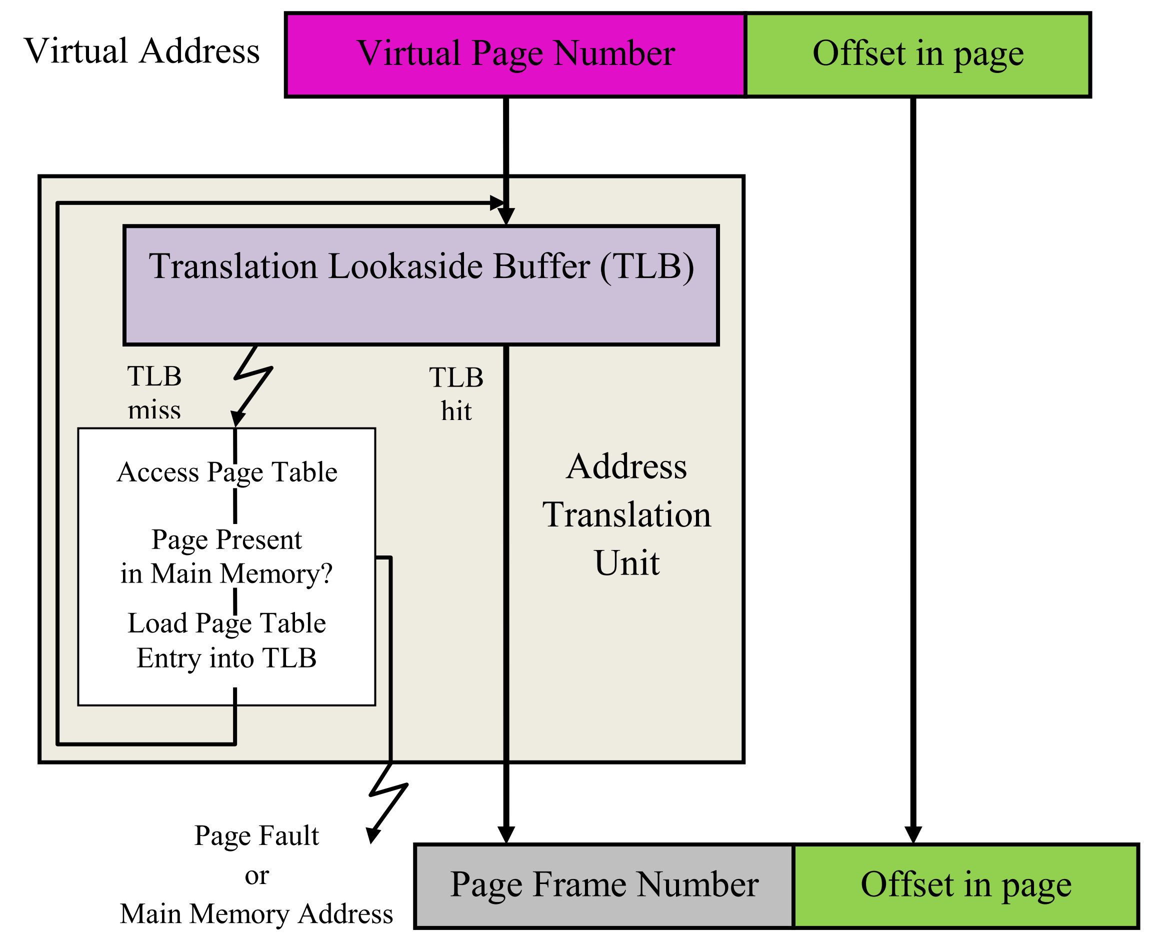


Figure 11.6: The ATU with a TLB using Conventional Page Tables

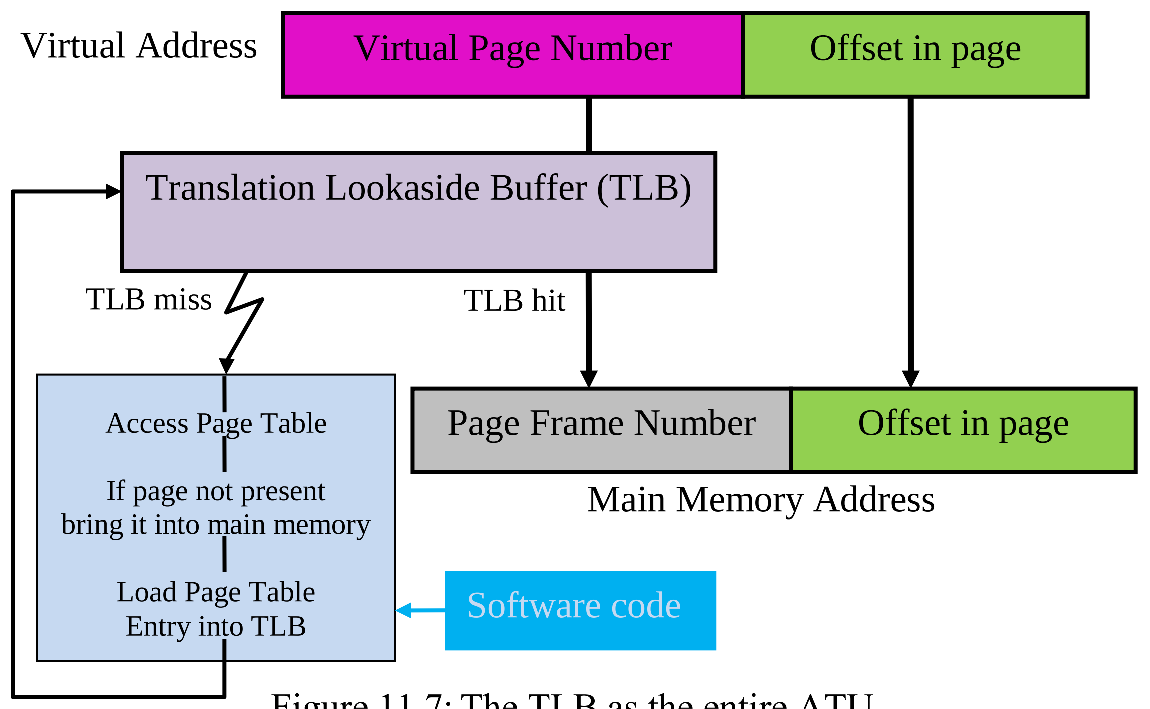


Figure 11.7: The TLB as the entire ATU

That such systems will in future become available is a realistic possibility, because the S-RISC architecture is not only capable of supporting SPEEDOS (and other capability based systems), but can do so in such a way that the many applications which currently exist on current RISC systems could be supported *without change*, except for a re-compilation with modified compilers. Hence manufacturers of current RISC systems can in future build S-RISC systems without fearing the loss of their current customers, but at the same time offer them more security, as I have shown in a recent paper entitled "S-RISC – Adding Security To RISC Systems", which can be obtained from the SPEEDOS website[45].

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

Chapter 12
Direct Addressability and Persistent Virtual Memory

In the previous chapter an alternative way of organising paging and segmentation was presented and we showed how this approach, known as the orthogonal model for paging and segmentation, can be implemented as a relatively small extension to RISC systems. The resulting S-RISC systems provide a very flexible way of supporting segmentation and paging, leaving the kernel of such a system with complete freedom to organise both page tables (mappings from virtual page numbers to disc addresses) and segment tables in any manner which the kernel designer chooses.

This gives us the freedom to begin a more constructive phase in which we consider how a secure software architecture might be designed. This involves taking a fundamentally different view of virtual memory, as *persistent virtual memory*. First the idea of *direct addressability* is introduced. This idea was first implemented in the Multics system and can be considered as a forerunner of persistent virtual memory.

1 Direct Addressability

In the mid-1960s, when mainframe computer systems carried out their work in a batch processing mode and personal computers had not yet been invented, computer architecture researchers at MIT in Cambridge, Massachusetts, developed a significant research system called Multics [98, 42]. Its aim was to demonstrate ideas relevant for time-sharing, i.e. for computer systems where individual users sit at terminals and interact directly with the (shared) computer.

Among the many revolutionary design ideas which appeared in Multics was one which will play a central role in the rest of this book. This was referred to as "direct addressability" by Multics designers. What they aimed to achieve with this idea was to allow *all the information in a system* to be directly addressable in the virtual memory, including information held in the file system. The fundamental advantage which they saw was that it avoids much copying of information between the file system and the computational (virtual) memory. The lack of success of this idea – in my view the most significant of all the Multics ideas – in the last six decades is due not to a fault in the basic idea, but in the way it had to be implemented on the hardware available at that time.

2 Advantages of Direct Addressability

Before looking at various attempts to implement a suitable environment for supporting the Multics idea of direct addressability, let us pause to consider what potential advantages it has and how it might ideally look.

When virtual memory was first conceived, little thought was given to the duplications that the conventional models caused – and still cause in current systems. This solution for the application programmer's overlay problem was only achieved at the price of severely increased complexity in the operating system.

For a start, the way disc space is organised in the extended computational memory is generally quite different from the way it was organised in the file system, so that different parts of the operating system (the virtual memory manager and the file system's disc manager) end up both managing disc space (in quite different ways).

Furthermore an enormous amount of copying of information takes place between the file memory and the computational memory, which in fact often simply leads to the unnecessary movement of data from one location on disc to another. This is the point which the designers of the Multics system most emphasised, describing the avoidance of copying as the fundamental advantage of direct addressability of information [99].

They pointed out for example that with direct addressability the program code files located in the file system could be directly used for executing programs, without first having to load them into the computational virtual memory. This idea had also appeared in the design of the Burroughs computer systems.

Being able to access program files directly in the file system is important not only as a general way of improving system throughput; it can also affect users directly interacting with a computer system, whether they are sitting at terminals connected to a central computer or are using personal computers or workstations. If you have ever become impatient about the time it takes your PC to start executing your program, the main reason is usually that the latter is being copied from the file memory into the computational virtual memory. In most systems the entire program is copied from disc into the main memory and from there it is discarded to the disc space of the extended computational memory.

Put simply, the program is first copied from disc to main memory and then again from main memory to disc! This is illustrated in Figure 12.1.

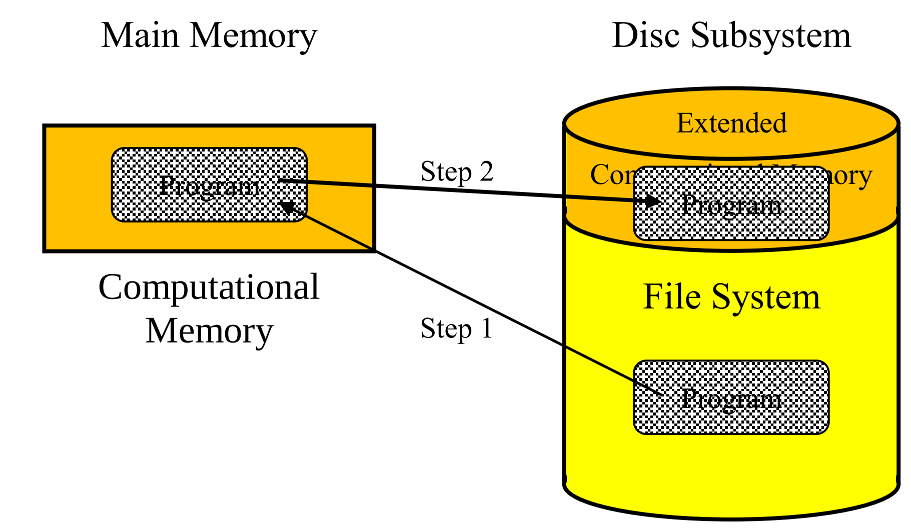


Figure 12.1: Copying Programs in a Conventional Virtual Memory

Almost all this copying can be avoided if the individual program units (segments or pages) can be directly addressed in the file memory, using virtual addresses rather than file system mechanisms. With this arrangement, starting a process means that the first statement in the program can be directly executed in the virtual memory. This causes a virtual memory fault requiring *only the unit containing this statement* to be read into the main memory (assuming it is not already in main memory, for example because another user has run the program recently or is concurrently using it). Thereafter only the units which are actually required need be brought on demand into the main memory. This in turn means that the process can start executing immediately instead of the user having to wait a couple of minutes while all the unnecessary copying takes place.

The conventional virtual memory technique of loading an entire program from the file memory into the computational memory is an overkill solution for another reason. A well-designed robust program contains a very substantial amount of code which is designed to cope with and recover from errors that in practice only rarely occur. On most of the occasions the program runs, these error-handling procedures are (hopefully) never used. So copying these pages at the beginning can be an absolute waste of effort. Similarly programs such as text editors, spread sheets and the like contain a bewildering variety of optional features which many users often never use. It is also a waste of time to copy the code for all these features into the virtual memory.

There is thus little room to doubt that the direct addressability of program object code is both more efficient and at the same time more convenient for users. And there is a final advantage: by eliminating the load operation from file memory into the virtual memory the need for the loader software itself disappears, thus making the operating system simpler.

Similar considerations apply also to the use of files containing data rather than programs. In most computer systems access to data in files is a rather tortuous activity. Data in the file system cannot normally be directly addressed. Instead the application process calls routines of the file system to request that data be read from a file. Such data transfers usually take place via an intermediate buffer area (a group of memory locations set aside for this purpose) in the computational memory. If information is modified by the application it must then be passed back to the file system in a similar way, first being copied into an intermediate buffer and then written back to disc.

In a modern database system the management of the intermediate buffers can itself be a complex problem. Such complexity becomes particularly apparent, for example, if many different transactions have on-line access to a shared database. A former Ph.D. student and I have discussed in detail some of the problems associated with this, as well as some of the corresponding advantages of being able to address information in files directly in database systems [100].

A technique which is sometimes used in an attempt to improve efficiency in accessing files is to implement *memory-mapped files*. This basically involves copying an entire file into the main or virtual memory, so that subsequent accesses are direct and the file system interface is avoided. But the initial act of copying the file into the computational memory and the subsequent recopying of it back to the file memory after the application has finished using it result in very similar copying overheads to those we have already described for program code.

A further advantage of direct addressability which was mentioned by the Multics designers was the promise of a very attractive reduction in program complexity for the programmer. This results from the elimination of the quite separate and distinct techniques used in conventional programming languages and software systems for managing information in the computational memory and for managing information in the file memory. This is an issue which has been tackled by the persistent programming community.

3 Persistent Programming

Conventional programming languages usually provide features for manipulating *temporary* data structures which are generally straightforward and convenient for programmers to use. These include structures such as arrays, records and linked lists. However, these convenient programming constructs *cannot* be directly used for accessing *persistent* information held in the file system. (This reflects the fact that information in the file system cannot be directly addressed by user programs.) Instead the programmer accesses the latter via a special file interface provided by the programming language, which is then transformed into the operating system's interface routines. There are several disadvantages in having one style of interface for file data and another for temporary data.

First, the temporary data structures in a program are not stored in compatible formats with the persistent data structures. Second, storing temporary data structures into files is often not a straightforward task, not only because of the different formats, but also because pointers consisting of addresses in the virtual memory cannot simply be copied into the file system and later reused, because the underlying main memory or virtual memory addresses may be different at a later time. This problem is further complicated by the fact that files are commonly used concurrently by several application processes.

This theme was taken up in the early 1980s by M.P. Atkinson and his colleagues at the University of Glasgow together with R. Morrison and his group at the University of St. Andrews. In order to avoid having two different approaches for programming temporary and persistent data, they developed a programming technique known as *persistent programming*, based on the use of *orthogonal persistence* [101]. They argued inter alia that the same data structuring mechanisms should be used to program temporary data structures in the computational memory and to program persistent data structures. To demonstrate this idea they developed the programming language PS-Algol [102][46] and later a new persistent programming language called Napier [103].

The persistent programming groups set about demonstrating the feasibility of persistent programming by implementing "persistent object stores" for PS-Algol and Napier above conventional hardware, using the basic facilities of conventional file systems. Such a software-oriented approach, which inevitable has a high performance overhead because it had to be implemented in a conventional virtual memory environment, was forced upon them by a lack of appropriate hardware.

4 More Advantages of Direct Addressing

The management of large bulk data files has become a specialized activity, known as database management, and this has resulted in the development of special database languages which have tended to use quite different data models from those underlying the design of programming languages. Hence these too have quite different interfaces from the programming language data structures.

[46] see also https://en.wikipedia.org/wiki/PS-algol.

While such database systems tend to provide much more powerful facilities than basic file systems they add yet another layer of complex software which adds to the inefficiency of data accesses. Thus although in the final analysis the application manipulates its persistent data – like its temporary data structures – in the virtual memory, it may have to do this indirectly via database routines which themselves may call file system routines.

The sharp division in most systems between a computational virtual memory and a file and/or database system gives rise to at least two further areas of duplication and unnecessary complication: synchronisation and protection. With regard to synchronisation, the CPU normally provides simple and efficient mechanisms, but above this the file system provides further synchronisation mechanisms, and then on top of that there are often additional database mechanisms to achieve synchronisation. This is necessary because the CPU instructions cannot act directly on synchronisation variables in the file or database system, since the latter cannot be directly addressed in the virtual memory.

And perhaps most significantly from our current perspective, the conventional virtual memory organisation leads to a multiplicity of protection mechanisms. This is inevitable if data in the file and database systems cannot be directly addressed. This additional complexity is more likely to assist security breaches than to hinder them.

What all of these points clearly indicate is that enormous benefits could be gained if it were possible to address both non-persistent (computational) and persistent (file and database system) data structures in the virtual memory in a uniform manner. How then can such a directly addressable file system be implemented?

In the following sections we shall first present an ideal model for implementing direct addressability. Then we shall consider various hardware attempts which have been made to support it, beginning with Multics. With a knowledge of the pitfalls and strong points we can then look again at whether and how direct addressability can be effectively implemented.

5 An Ideal Persistent Virtual Memory

The fundamental feature of direct addressability is exactly what its name implies: the ability directly to address not only computational data but all persistent information (conventionally held in file systems). With hindsight it is very easy to see where the conventional model for virtual memory went wrong: it left intact the distinction, which had been traditionally present in pre-virtual memory systems, between a computational memory and a file system.

This distinction originally arose because of the different characteristics of main memory and discs. Main memory is fast, expensive and usually non-persistent. Disc memory (along with similar magnetic media) is much slower, much cheaper per byte or word stored, and is persistent. Furthermore, discs are attached to computers in a similar manner to I/O devices. All of these things made it natural in the early systems to keep the two kinds of memory distinct.

The problems really began with the half-hearted virtual memory solution, which simply "stole" some (but not all) of the disc space. All the problems discussed in the previous section point to the need for a much more radical solution: the entire memory should be viewed as a single uniform persistent virtual memory, which has a single virtual addressing mechanism. We call this a *persistent* virtual memory because, unlike conventional virtual memory, it holds the persistent data and programs (i.e. the files) of the system as well as the computational objects. Put simply, the persistent virtual memory includes *the entire disc space* and thereby effectively renders the file system (in its conventional form) unnecessary. The idea behind persistent virtual memory is illustrated in Figure 12.2, which should be compared with the equivalent diagrams for conventional memory (Figures 7.1 and 7.2).

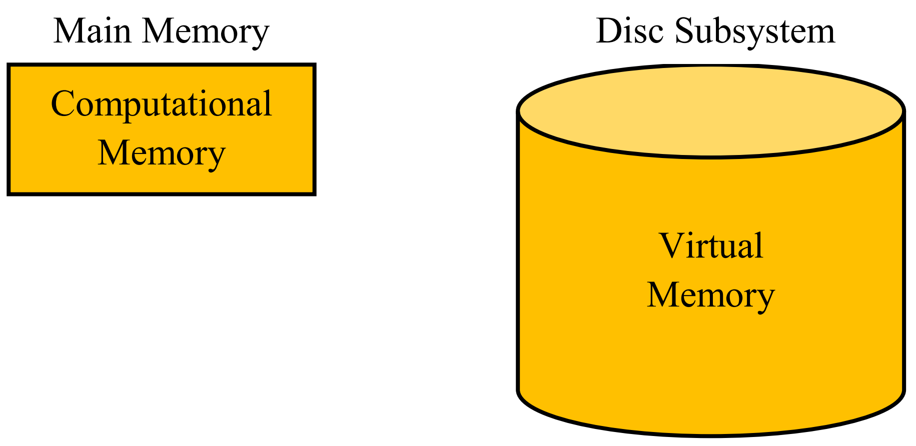


Figure 12.2: Persistent Virtual Memory

We have seen in the last few pages some of the benefits which can be derived from this memory model. It promises a vastly reduced amount of copying of information and a much more attractive persistent programming environment. As our story unfolds further we shall encounter several other benefits. These include much simplified software, and some surprising new benefits, including security benefits, which arise from the fact that the *computational* memory has become persistent and there is no separate file system.

We shall also have to consider some new problems which the model raises. For example, how can a non-persistent main memory (which in this model in effect functions as a cache for the information on disc) become a component of a persistent virtual memory? But for the present we leave all these interesting issues aside, until we have a solution for a more fundamental question. How can a persistent virtual memory be addressed?

6 Direct Addressing in Multics

As its hardware base Multics used the General Electric 645, which had a virtual memory system based on paged segments. The fundamental difference between Multics and other systems of the 1960s was the decision to make files from the file system directly addressable as segments in the virtual memory. However, to achieve that with the GE 645 architecture was by no means a straightforward matter. The fundamental problem was the insufficiency of the available virtual addressing range.

The strategy was to map files from the file system into paged segments of virtual memory. The segment size limited the maximum size of an individual directly addressed file to about 1 MB. This is far too small to implement the business files of a modern database system, although it was probably not a severe problem in the Multics environment, where timesharing users typically have relatively small files (e.g. source programs, object programs, text documents and similar).

A more difficult problem arises with the number of files which may need to be directly addressed as segments. In a timesharing system such as Multics there may be many users, each with say 1,000 files. With 1,000 users one would have to think in terms of at least 1,000,000 files concurrently existing in the file system. But using the virtual address in the obvious way would have led to a limit of 16,384 files. So there was a problem with addressing all the files uniquely. It was this problem which caused many, in fact most, of the complications we are about to encounter.

What Multics actually did was to take advantage of the fact that no process ever wants to access all the files in the system at the same time. In fact no normal process ever wants to access even 16,384 files at the same time. Consequently it was possible to link files to processes dynamically, allocating and deallocating segment numbers for them as needed. On the surface this seems a reasonable solution but in fact it is fraught with problems, because it relies on the use of potentially ambiguous identifiers (the same segment numbers used by different processes).

The first problem, which the Multics designers recognized from the beginning, was that if the same file is used by several processes concurrently then it is to be anticipated that they will use different segment numbers to address it. In other words while Process A is accessing File F using segment number 3187, Process B might be accessing the same file using segment number 5940.

In fact the problem is worse than this. How can a code segment know at all which segment numbers to use to address its files? Clearly addresses containing segment numbers cannot be embedded in program code segments.

The Multics designers also foresaw another problem. Code segments are often recompiled (for example to correct errors). This can lead to the individual code segments beginning at different offsets in an architectural segment in different versions of the same program code. This means that not only do the architectural segments have different numbers, but the offsets needed for inter-segment references (i.e. references to the logical segments within the architectural segments) may also change.

In order to overcome these problems Multics used some extremely complex linking mechanisms, which we have chosen not to describe here. But although the solutions were quite ingenious they were also quite cumbersome. It is presumably because of this complexity that most designers of later operating systems did not follow the Multics philosophy of making files directly addressable in programs. This is unfortunate because the basic idea of direct addressability has many advantages, both in terms of execution efficiency and programmer convenience. In the present author's view the basic concept was the best idea which came from the Multics designers, but as a result of the implementation difficulties it is the one which later received the least attention. However, some years later another attempt was made to harness these advantages, and it came from a rather surprising quarter.

7 Direct Addressing in the AS/400 Family

In reflections on their earlier design of the DEC PDP-11 systems, Bell and Strecker commented:

> "There is only one mistake that can be made in a computer design that is difficult to recover from – not providing enough address bits." [104]

This was a mistake which the designers of the IBM System/38 and its successors in the AS/400 family wanted to avoid. When IBM announced the System/38 to the world on 24th October 1978 there was considerable surprise that it contained 64 bit "pointers" and 48 bit virtual addresses. What is perhaps even more surprising is that one of the major technical aims of this system was to provide direct addressability to files. The way IBM intended to solve the addressing problems created by direct addressability was to make virtual addresses large enough to be usable as unique names.

The System/38 was not a very successful system commercially. In the early days it had severe performance problems, and it was too expensive for IBM's

System/3 series customer base. However, in 1988 IBM announced an apparently new system, the AS/400, but under the covers the design was actually based on the System/38.

In 1991 IBM reached an agreement with Apple Computer and Motorola to develop the PowerPC as a common architecture for computer processors with a wide range of aims and performance capabilities. The PowerPC was planned as a RISC processor architecture and the initial IBM input to the project was based on their previous mainstream RISC development, the RS/6000, which itself came out of an IBM RISC tradition going back to the PC RT and ultimately the IBM 801.

The AS/400, which by this time had become a successful product line, had a quite different background and tradition within IBM, as a commercial database computer, where high processor performance was not as important as good database performance. Not surprisingly the AS/400 team was reluctant to join the PowerPC alliance at first. But under pressure from IBM top management they got involved at a fairly late stage, and managed to influence the design in some important ways. This has resulted in the use of PowerPC processors in later AS/400 systems. This story is entertainingly told by Frank Soltis, chief architect of the System/38 and AS/400 in his fascinating book [105].

The original System/38 actually had a virtual address size of 48 bits. Although its software designers had intended to have a 64 bit virtual address, with the intention of never running out of addresses, engineering decisions forced them to accept a 48 bit address. They therefore used some tricks to make this appear as a 64 bit address when used in a "pointer".

The way the hardware organised the use of virtual addresses gave the system just over 4 billion[47] segments, each segment having a maximum size of 64 KB. The software designers were not happy about either of these limitations. They therefore chose to define the pointer part of *system pointers*, as a 64 bit virtual address. They used two tricks to achieve this. The first theoretically increased the apparent number of segments in the system to 2^{48}, which is a little more than 256 thousand billion segments. The second trick theoretically increased the effective segment length from just over 64 KB to more than 16 MB.

The designers originally calculated that the system could run without problems for about 180 years, basing this on the expectation that there would be one restart of the system per day for 365 days per year. Since nobody expected the System/38 to be around as long as that, there seemed to be no problem.

[47] There is considerable confusion about the meaning of the word billion, see https://en.wikipedia.org/wiki/Billion. In this book I use the short scale billion, i.e. one thousand million.

But the way that the software designers actually implemented the system led to much severer restrictions. One of the tricks involved dividing segments into different types. Consequently only a little over 4 million jobs could be started and only 4 million temporary files could be used over the life of the system. To make matters worse some installations often left the system running over night, with the consequence that much more than a day might elapse between system restarts. According to Soltis larger customers actually began to run out of temporary segments. Their only recourse was to close the system down and reinitialize it. The software was changed in later releases, but in the end the problem was only completely solved by the appearance of the PowerPC with a genuine 64 bit address space. When used in an AS/400 system the PowerPC virtual address is split into a 40 bit segment number and a 24 bit offset, which is what the System/38 software designers wanted from the beginning.

The System/38 designers faced similar problems with the allocation of segment numbers for persistent files. At the hardware address level only one quarter of the possible segment numbers could be allocated to permanent files, and the use of segment groups (the second trick) had the effect of reducing this even further. Thus a maximum of about 4 million segment numbers were available for persistent files. This is not a very large number of files over the lifetime of a system.

A new trick was used to allow up to about 4 million new persistent objects to be created *in each run of the system* (i.e. between initializing and closing down the system). But there was a catch. No two persistent objects could exist at the same time if they had the same 48 bit address, because at the hardware level only 48 bit addresses were used. The solution for this problem entailed *never* completely deleting a persistent segment, even if the owning user had actually "deleted" it. Instead – in the System/38 and earlier AS/400 systems – the some management information was stored in the segment header of a segment containing a persistent object, and this segment header was not destroyed.

The end effect of this was that up to a total of 2^{38} (about 256 billion) persistent segments could be created over the life of a system provided that no more than 2^{22} (about 4 million) existed at the same time. Soltis mentions on p.201 of his book how IBM (artificially) limited the amount of disc space that could be attached to a system to ensure that they would not run out of segment identifiers for persistent files.

The above description in fact oversimplifies the problems, but if you want to know more you can read the full details in Soltis's book.

8 Persistent Virtual Memory

Despite the apparent implementation problems the idea of direct addressability, when taken to its logical conclusion, simplifies so many problems that it should clearly be strived for. At this point we have not yet described sufficient of the SPEEDOS software architecture to explain how the problems discussed above can be solved, but with the confidence that none of these problems is insurmountable, we now present the basic model of direct addressability which will form the basis for the rest of the book.

The proposed solution is quite radical in that in contrast with the systems discussed earlier in this chapter it *totally eliminates* the existence of a separate file system. The basic idea is that instead of "stealing" a part of the file system's disc space (and otherwise leaving the file system intact), we redefine the entire disc space as a persistent virtual memory which encompasses the entire magnetic media in a system, including devices such as external discs, as was already illustrated in Figure 12.2.

In fact we take a further bold step by defining this virtual memory as not being confined to a single computer but as encompassing the entire magnetic media (and other storage devices such as flash memory) *in all systems* which participate in the SPEEDOS architecture.

The conventional view of networked systems is illustrated in Figure 12.3.

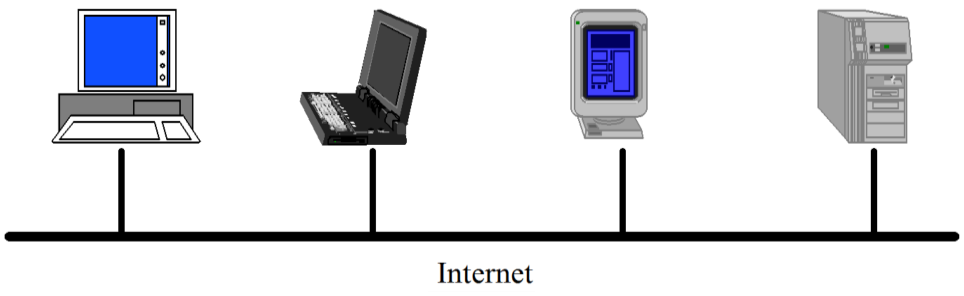


Figure 12.3: Conventional Networks

In a SPEEDOS environment each node in the network, instead of being viewed as a separate computer, with its own virtual memory, is seen by other computers which participate in the same concept as a set of remote discs which can be accessed by a SPEEDOS process, see Figure 12.4.

This view of virtual memory eliminates a myriad of duplications and complications which are found in conventional systems, by supporting all those points mentioned above as advantages of direct addressability, but also by greatly simplifying the problems associated with networking and limiting protection issues to the virtual memory.

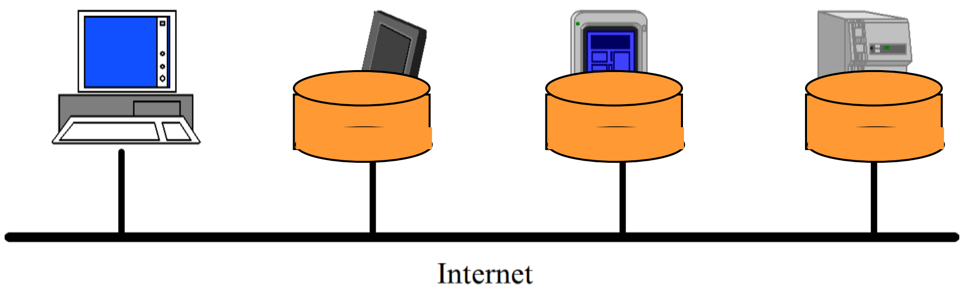


Figure 12.4: The SPEEDOS View of Networks

Of course we will have to provide evidence in future chapters that such a scheme can be implemented, but at this stage we must ask readers to have faith that these proofs will be provided. It can already be said, however, that a limited early version of the idea was successfully tested in the MONADS Project, the major project which the author established at Monash University in Australia in 1976. This included both the general idea of a persistent virtual memory and a limited form of networking in a local area network of four computers.

The main relevant features of the MONADS-PC system were as follows.

i) Virtual addresses were 60 bits wide.

ii) The top two bits indicated which of four networked MONADS-PC systems was being addressed.

iii) The addresses were unique across the four computers.

iv) The memory consisted of three basic kinds of address spaces (equivalent to containers in SPEEDOS): file address spaces (holding persistent file data), code address spaces (holding the code of compiled programs) and stack address spaces (for holding process stacks).

Although this system successfully tested the basic concept of a uniform persistent distributed virtual memory, some of the techniques which were used are not scalable to modern day needs, mainly as a result of the need for unique addressing and distribution throughout the Internet. However, solutions have been found and will be explained in volume 2, with some hints being provided in this volume in chapter 16.

9 Conclusion

This chapter has described an important basis for implementing secure computer systems. Computer memory is where hackers find the information which they wish to steal, modify or even destroy. We have taken a schematic look at conventional memory, i.e. how current computer systems organise their memory, and have found that its interactions with conventional file systems create many duplications and problems. This recalls the situation described in Chapter 1 with the Berlin Wall. The approach to be pursued in the rest of the book can be considered, from the viewpoint of its conceptual simplicity, as more comparable with the Alcatraz approach to prison security.

However, it still remains for us to demonstrate that a simple implementation of the persistent virtual memory concept is possible. Before we can do this, it is first necessary for us to describe some further unusual and unconventional concepts. To this end the next chapter describes how the persistent distributed virtual memory can be populated with software without resorting to a conventional file system.

Part 4
The SPEEDOS Software Model

Chapter 13
Software Structures

Recent chapters have concentrated primarily on the hardware features which might serve as a base on which to build secure computer systems. However, although an appropriate hardware design plays a significant role in the development of secure systems, this alone is not enough. The security of a system depends equally on the strategies adopted in the design of the software system (and of course on the correct implementation of these strategies). In this chapter we examine some ideas for structuring software. In the next chapter we shall then develop a general software model which can serve as a flexible and modular base on which a variety of different security models can be implemented.

An important prerequisite for designing secure and reliable software systems is the existence of a simple and efficient structural framework for the software itself. Such a framework must above all take into account the fact that large and complex software systems cannot be produced by a single person. A complex software system, such as an operating system or an airline reservation system or a banking system, contains millions of lines of program code. The development of such a system involves hundreds or even thousands of programmers working together over several years to produce a single software product. For this reason any large system must be decomposed into separate units which can be programmed by different programmers and programming teams.

In the 1950s and the 1960s, when the first large software systems were developed, software designers had little knowledge or experience of how to go about the task of breaking large systems into smaller units. The pattern which they followed was based largely on their experience with designing individual application programs. The results of this approach are still very much with us today. Large application systems are typically decomposed primarily into two kinds of software units: programs and files. The programs contain the code to be executed, the files contain the data on which the programs operate. This may seem to be a very reasonable way to decompose systems, but in fact it leads to lots of problems, as we shall now see.

1 The Software Crisis

In the early days of computing, during the 1940s and early 1950s, software development was regarded as a relatively simple and straightforward task. In that period the real focus of interest was centred on the computing machinery itself. The programs which were developed to execute on early computers were by modern standards relatively straightforward and unambitious. Programming mistakes were made, of course, but there was a general feeling that this was due simply to lack of practice. In retrospect we realize that such an attitude was unduly optimistic, if not naive.

During the 1950s and 1960s the potential uses of the computer were increasingly recognized and ambitious projects were undertaken to realize this enormous potential. Some of these were application projects, concerned with producing useful end products, such as the development of banking systems and systems to control the reservation and booking of airline seats. Others were system software projects, concerned with improving the use of the computer itself. These included the construction of compilers for high level languages, and of operating systems for improving the throughput of the computer by the use of multiprogramming techniques. As the ambitions of users and programmers grew, so did the complexity of the systems which attempted to realize these ambitions. And as the complexity of the systems increased, their poor quality became increasingly evident.

By the mid-1960s the software industry was in a chaotic state. Systems were delivered late, often several years late. They were unreliable – the MTBF (mean time between failure) for many systems actually delivered to customers could often be measured in minutes! Attempts to rectify errors frequently succeeded often only in creating new errors. Attempts to extend the use of a system or adapt it to solve a different but related problem were often doomed to failure. The idea of trying to transport a large program, such as a compiler or an operating system, for use on a different type of computer was completely out of the question. As a result of all these problems the costs of software systems soared well above the cost of the hardware on which they were executed.

By the late 1960s the software crisis had grown to such proportions that the N.A.T.O. Science Committee organized two international conferences. The first, held in Garmisch, West Germany in 1968, was a working conference on "Software Engineering", the title being provocatively chosen to focus attention on the need for software development based both on theoretical foundations and on practical disciplines, as in the established branches of engineering. Practitioners with first-hand practical experience of the problems were prepared to air these problems in public, and the conference focused largely on the nature of the problems, rather than on possible solutions. This was followed by a second conference in Rome, Italy in 1969, at which the focus of attention was to be on the technical problems of large software projects. However, according to the Conference Report editors, the most important outcome was the recognition of the significance and extent of the communication gap between the academics and the "real-world" practitioners [106, p. 145]. It is interesting to consider the reasons for this communication gap, because they explain to some extent (although not entirely) subsequent developments in software technology.

On the one hand, academics tend to concentrate on solutions to small, relatively manageable, problems which can be tackled in a university environment. As a result the 1970s and 1980s witnessed a good deal of progress in the area of program development. Techniques such as structured programming [107], stepwise refinement [108], abstract data types [109], object oriented programming [110], together with improved programming language designs (e.g. Simula 67 [111], Pascal [112], Smalltalk-80 [113] and Modula-2 [114]) and improved verification techniques, have all played a significant role in raising the quality of modern software.

But despite these very important developments many of the problems experienced in the 1960s are still with us today, during the third decade of the twenty-first century. The main reason for this is that most of the problems stem not from the individual *programs* which constitute a software system, but from the *structure of the system* itself.

This explains why academics and practitioners often found that they were talking at cross-purposes. Academics tend to emphasize the program level, because this is the level with which they can come to grips, given the very limited resources available in universities. But practitioners are more concerned with the problems of system design for systems which might involve thousands of man years of development effort [115] or occupy many megabytes of memory. Academics sometimes express the opinion that the problems are really the same, but the fact is that many of the *programming* techniques cannot simply be scaled up to solve system problems on a large scale. A bigger structured program, or more verification effort, or a better programming language, is simply not the solution to the most pressing system design problems. These techniques can contribute significantly to the quality of the individual programs which form the building blocks for larger systems. But improving the quality of bricks, or even doors and windows, does not solve the architect's problem of how to design a well-structured house!

The analogy is of course exaggerated. Programs may have a more significant role in software system structures than bricks in house design, and on the other hand there is no doubt that some software architects are trying to build cumbersome and bizarre systems. But the facts remain that software systems are not simply scaled-up programs and that large systems do have an important role to play in modern society.

2 Software Systems

A conventional software system is not merely a scaled up computer program. It is a complex entity consisting of many programs which interact with each other, primarily using data structures (usually in the form of files) as their common interfaces. The programs in a software system are not all executed together. Some programs are run frequently, others are run say once a week or once a month, while yet others are executed perhaps only once a year.

For example in a typical banking system those programs which carry out the normal everyday banking transactions (such as recording deposits and withdrawals) are usually executed each night in a batch processing system or they may run throughout the day in an on-line banking system. But there are other programs which are run at less frequent intervals. For example some programs run once a month (e.g. to calculate interest payments or charges, or to provide management reports), while others may be executed quarterly (e.g. to calculate account fees and provide yet more reports), and yet others are needed only once a year (e.g. in connection with taxation requirements and end of year accounting). It is important to realize that all these programs, independently of the question how often or when they run, largely make use of the same set of files.

This situation is not something special about banking systems. It is typical of commercial data processing systems (e.g. airline reservation systems, insurance systems, building society systems) and of computer systems used by government departments, etc. In fact this pattern of usage applies to virtually all computer systems where security is a major issue.

The suite of programs making up such a computer system is not designed and programmed once and for all and then never changed after it has been delivered to its users. A computer system exists to serve an organization, and its programs must be frequently modified to reflect the changing needs and circumstances of that organization. For example if a bank introduces a new kind of bank account, its daily programs must either be changed or new programs added (or both). Similarly a change in the tax law might require the banks to change their annual programs. Or management might decide that it needs a different kind of report, this time affecting one of the quarterly programs.

What often starts out initially as a relatively small and simple application suite is gradually transformed into a monster software system, existing in several versions, in a state of constant change, lacking a master plan, and costing more *annually* in "maintenance" programming than was originally envisaged as the *total* system cost. This is a story which almost every experienced software development manager will tell you. Getting the initial software developed is only the tip of the iceberg. The majority of the cost of computer systems goes into software maintenance.

3 Software Maintenance

Maintenance is the name loosely used throughout the software industry to describe the activity of making changes of any kind to programming systems after the initial development is complete. It includes making changes which are needed to correct errors found in the original programs, making improvements and extensions to reflect the changing needs of an organization, carrying out modifications which become necessary because the programs have to run on a new hardware system, and so on.

Software maintenance costs make up a very significant proportion of data processing department budgets. But that is not all. They often account for a significant slice of the entire budgets of companies and even of the national budgets of the advanced and developing nations. This makes software maintenance a significant problem, not only for the software industry, but also for company managers and politicians alike. Since the 1990s this problem was increasingly recognized and research funding was earmarked for this neglected area of software technology, for example as part of the European Esprit research program.

Software maintenance has been a cinderella for many years both in the software industry, where the mistake is often made of leaving the relatively uninteresting work of software maintenance to trainee programmers, and in academia, where it has long been seen as a relatively unfruitful area for research. So perhaps we should welcome the idea of research funding being earmarked for bringing improvements in an area such as software maintenance, an area which has proved to be such a drain on national economies. However, such funding schemes can only be given a qualified welcome. The reason is that the fundamental problems which arise in software maintenance have been created in the software *design* process[48], not the maintenance process! In this respect it is worth bearing in mind that maintaining software is not an activity comparable with maintaining physical objects. Software does not need to be oiled regularly, or reconditioned, or cleaned, etc.

[48] A problem which we do not consider further here is that there are very many systems still in use which have been built over the last six decades or so. An improved design method does not solve the maintenance problem for these systems.

Instead software maintenance really boils down to two activities. First there is the problem of removing errors which were introduced in the design and implementation process. In this case we are not really talking about problems with a maintenance activity as such, but about problems with an inadequate design and implementation process. If the design and implementation had been done better in the first place, software would not need this kind of maintenance. In fact this is even truer than is the case in conventional engineering projects involving the manufacture of physical objects, where errors and faults can also creep in during the manufacturing process, because there is no equivalent manufacturing process in software. Once the software has been designed and implemented it is simply copied from one disc to another, using a process which is so reliable that the introduction of copying errors is negligible.

The second software maintenance activity involves extending and modifying existing systems to suit new requirements. This also is basically a *design* issue. The really important point in this situation is that the original system should have been designed in an extensible and modular fashion. Although the word *modularity* is one of the catch cries of the software industry, genuinely modular design is an art which is scarcely practiced in any significant sense.

In other industries modularity usually implies *inter alia* that a system can be constructed from components which have been separately designed and implemented according to standard specifications. In most cases such components are general purpose, designed to be incorporated into many different products.

For example, in the automobile manufacturing industry the designer of a new car model does not normally produce his own new designs for the tyres, or for the electric light bulbs to be used in the headlights, or for the spark plugs, etc. He can take advantage of the fact that such components already exist and that they are manufactured by secondary industry to a specification which allows them to be used in many models of car and sometimes in other products.

In contrast the software system designer and his implementation team usually create an entire software system from scratch. It is hardly surprising that such a method produces many errors. If components are designed and implemented anew with each system, usually ignoring the work done on other software systems, then there is much more scope for error than if the same well tried and tested components are reused in many systems.

Similarly if a design is based on the use of modular components it is usually much easier to extend or adapt this to new circumstances. The automobile industry does not produce new models out of thin air; it takes an existing basic design, then adapts and improves it as required.

Modularity is one of the key features contributing to the success of other engineering systems. We now consider what this should and could mean for the software industry.

4 Software Modularity

Although system designers and programmers talk readily of decomposing their software systems into modules and even regard modularity as an essential or a highly desirable aim, in practice there is no widespread agreement about what constitutes a module or what criteria should be used in the design of modules. As this is not a book about software engineering, we shall not embark on a long discussion about different views of software modularity. Instead we shall use a simple working definition which will help to develop our theme.

Most software designers and programmers will probably at least agree that modules are the building blocks out of which software systems should be constructed. Most will probably also be prepared to regard a module as an independent component which marks the transition between the work of the system designer and that of the programmer. In other words a module can be regarded as an object which is specified in a system design stage, and which is then handed over to a programmer or programming team whose responsibility is to implement it.

But that is probably about all that will be widely agreed upon about the meaning of software modularity. For example, there is sometimes discussion about how "big" a module should be. Some will say that it should be a program small enough to fit onto one side of a sheet of A4 paper (thus giving the programmer an overview of the design of the entire algorithm). Others have argued that a module is the same thing as a procedure. Yet others would view a collection of procedures (e.g. a subroutine library) as a module.

But such discussions are quite futile. Why should there be a single "size" for a module? In other engineering systems larger components are constructed from smaller components, which may in turn be constructed from yet smaller components, and so on. Similarly there seems to be no reason to exclude the possibility that software modules can be constructed from other software modules of smaller granularity, and so on.

It is more important to consider the implications of having modules with different granularities than to argue about what the "right" granularity should be. In particular, it is relevant for our discussion to consider where the knowledge of modules of different granularities resides in a computer system.

As we have already seen, there are small software components in a system, such as procedures, records and arrays. These can map onto logical segments at the architectural level of a computer system, assuming that the architecture supports logical segments. If it does not then these are managed entirely within the compiler.

The more interesting issue is, what software structures correspond to larger components? Traditionally operating systems have supported a variety of larger software structures. The most obvious of these are programs and files. But there are several other larger structures that are often explicitly supported in various ways by operating systems. For example most systems distinguish between programs and subroutine libraries. Programs traditionally have a single entry point (where the program execution begins), whereas a subroutine library is a collection of related subroutines which usually provide similar or related functions (e.g. statistical, trigonometric or financial subroutine libraries). In contrast with programs, a subroutine library usually has a separate entry point for each function. In older systems overlays, which we briefly encountered in Chapter 7, were regarded as modules. Another kind of module is an operating system module, which is usually managed and organized separately from application modules.

Whatever the definition of a module, the emphasis has in one way or another been on units which primarily contain code. This was undoubtedly influenced by the fact that programmers think of themselves mainly as designers of algorithms. Producing programs is their job, and the dynamic flow of control of the code is generally uppermost in their minds when they think about computing issues. Data on the other hand is what an application program or an operating system program produces. This can simply be stored in files or in operating system tables, etc.

This separation of software structures into programs and other code modules on the one hand and into files and other data structures on the other hand has determined the entire structure of software systems. As we have seen, the operating system itself is usually regarded as consisting of two main parts: the part which manages the computational memory and the part which handles files (i.e. the file system). Similarly database systems traditionally consist of a code part which implements the data base and a part holding the data of the data base. Application systems likewise consist on the one hand of programs and code modules and on the other hand of the files which they manipulate. Let us now look at some of the problems which arise when this approach is adopted.

5 Flow of Control Modules

Module decomposition influenced exclusively by algorithm design and other dynamic flow of control considerations inevitably leads to the design of systems which allow several modules to access the same data structure. As an example of this we consider an operating system design approach which was common in the 1960s for mainframe computers. A typical operating system from that period includes various data structures representing the state of particular aspects of the system.

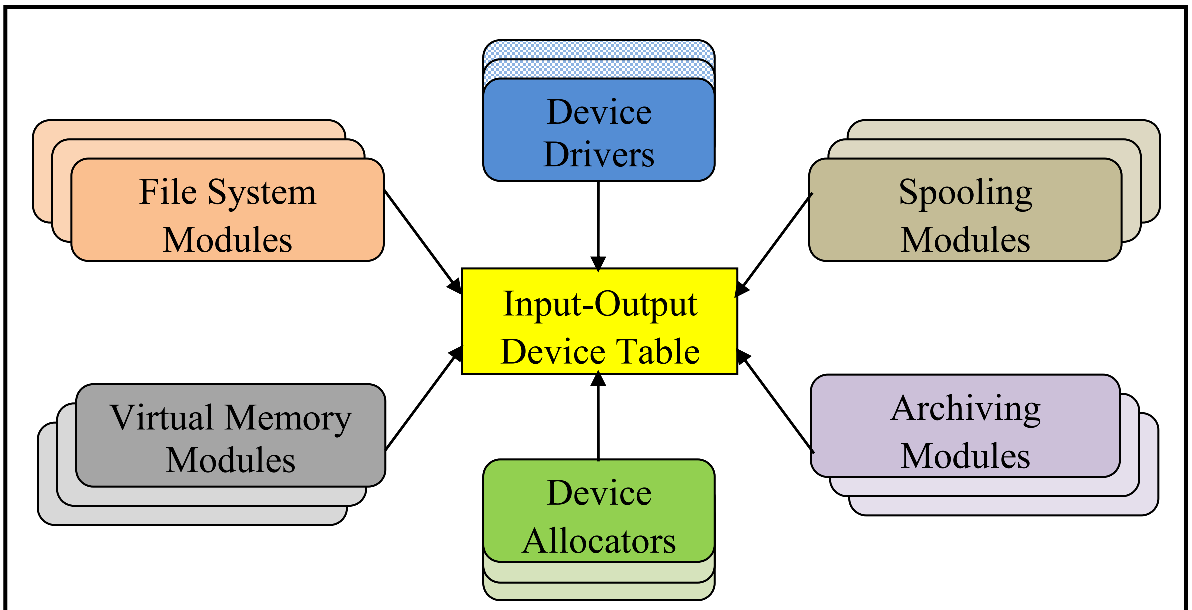


Figure 13.1: Flow of Control Modules in an Operating System

A common example is a table defining the properties of the system's input-output devices. This data structure must be accessed by several modules. As Figure 13.1 illustrates, these might for example include:

(a) device drivers which perform the actual input-output operations;

(b) file system modules which need to know the properties of the disc drives and the identities of mounted discs;

(c) virtual memory modules concerned with the discs holding the extended computational memory;

(d) modules which allocate input-output devices to user programs;

(e) spooling modules which read and write data for slow devices;

(f) archiving modules which transfer files between magnetic tape and disc.

A number of serious problems arise in such a situation:

(i) *The specification of the system design is difficult.* Each major data structure in the system must be specified down to the last byte and bit at a very early stage. This in turn means that the system designers must anticipate many details of the design of the algorithms which access such structures. Consequently there is a strong risk that changes will have to be made to the specifications when the algorithms are eventually developed.

(ii) *Communication between the implementers of separate modules is high.* It is impossible with present specification methods to achieve an absolutely unambiguous specification for a raw data structure, particularly in terms of the interpretation of the values which it contains. Consequently the implementers of the modules which access it often need to spend an excessive amount of time discussing the details (without any certainty that they finally reach a common understanding). Misunderstandings of this kind inevitably result in programming inconsistencies.

(iii) *Inconsistent modules create difficult debugging problems.* Because each module accessing a shared data structure relies on the correctness of other modules, an error in one module often manifests itself as strange behaviour by another module (which may itself be correct). Detecting the source of such an error can be a very difficult task, because it may involve a careful examination of *all* the modules which access the structure.

(iv) *Verification is difficult.* Verification of the correctness of the system, either by formal proof or by testing, is extremely difficult if several modules access a common data structure, because the validity of assumptions which a module makes about the data structure depends on the actions of other modules.

(v) *Synchronisation problems easily arise.* If the separate modules can execute concurrently, access to the data structure must be properly synchronised to ensure that its contents remain consistent. This means that each accessing module must contain the correct synchronisation protocols. If one of the programmers forgets this, or gets it wrong, then the system will once again be in error.

(vi) *Maintenance of the system is difficult.* Apart from the debugging problem mentioned above, maintenance becomes a difficult problem, especially if it is not carried out by the original system programmers, because the maintenance programmer is faced with the formidable task of understanding many complex indirect interactions between the various modules.

(vii) *Extension/adaptation of the system is difficult.* Changes to the system design are extremely difficult to make without errors both because of the complexity of the interactions and because a change to a shared data structure incurs the risk of requiring changes to all the modules which access it.

(viii) *Optimisation of the system is difficult.* Optimisation, e.g. to improve system performance, often involves changes to the underlying data structures. In this case the problems which arise in system extension or adaptation arise here also.

These are among the more serious of the problems which gave rise to the software crisis that has been with us since the 1960s and which the N.A.T.O Conferences [106] did not succeed in solving. Among the systems most affected at that time were those major operating systems which were designed according to the above technique, relying on a system decomposition based on flow of control considerations and the extensive use of shared tables as major interfaces between modules.

It is not difficult to see where the basic weakness of such a design method lies. The fundamental problem is that there are many complex interactions which take place indirectly (via the shared data structures) between apparently independent modules.

Since the 1960s some progress has been made in modular design techniques. This is particularly true of the object oriented design approach, which has been applied with some success to the design of a few newer operating systems (but not those widely in use). Unfortunately it has not been widely recognized that the same problems occur in other systems, particularly in commercial data processing systems, which are similarly decomposed primarily into programs and code modules. Interactions between the programs take place indirectly, as in the operating system example just discussed, via independent accesses to major data structures – in this case in the form of files. This approach to system design continues to result in expensive, poor quality software which is error-prone and difficult to maintain, to extend and to adapt to other environments. Even the appearance of object oriented databases has not provided the necessary breakthrough, because the objects in that approach correspond by and large not to the larger granularity units under consideration here, but to the smaller objects such as the database records.

In the rest of this chapter we shall consider the main ideas behind object oriented programming, since this offers the most promising basis for finding a solution to software system design problems. But in the course of this review the reader should bear in mind that in general the object oriented programming approach has in the past been used mainly as a technique for structuring individual programs, not entire systems. In other words, the "objects" in object oriented programs are modules of small or medium granularity that are contained within programs, not the programs and files themselves.

6 The Information Hiding Principle

Before exploring the idea of object oriented design in more detail, it is worth considering an important forerunner of this approach, the information hiding principle, to see how the problems associated with flow of control modularisations can be solved. The information hiding principle was proposed as a software decomposition technique by D.L. Parnas in the early 1970s [116].

Observing the excessive amount of communication which had to take place among implementers of separate modules [117], the tendency of programmers to take advantage of detailed implementation information about other modules, and the problems of specifying module interfaces [118], Parnas proposed that all detailed items of information in a system or program should be hidden within a module, and that this module should present a relatively simple interface to other modules. He went on to illustrate how this can be achieved by placing together into a single module both a major data structure and the routines which access it. Other modules which need access to the information in the data structure obtain this indirectly by calling the interface routines of the module.

If the information hiding rule about data structures is strictly enforced throughout a system the consequence is that all inter-module interfaces can be expressed in terms of routine calls. For example, suppose that some program (e.g. a compiler) needs to store items (say integers) in a queue, then the queue (called a First In First Out – or for short FIFO – Queue) can be built as a separate information-hiding module, with the following interface routines:

(a) an operation "enqueue", which puts an element at the end of the queue,

(b) an operation "dequeue", which removes the element at the beginning of the queue,

(c) an enquiry "first", which returns the value of the first element without changing the state of the queue, and

(d) an enquiry "length", which returns the number of elements in the queue without changing its state.

This is illustrated in Figure 13.2. Hereafter, diagrams of this kind are used to represent information hiding modules.

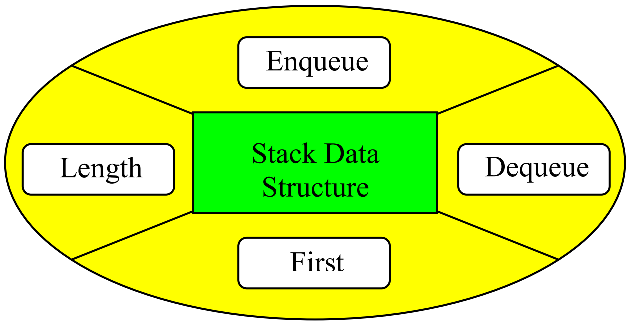


Figure 13.2: A Simple Information Hiding Module – A FIFO Queue

A crucial advantage of this technique is that all the major implementation decisions remain hidden from users of the module. There is no indication whatsoever in the interface definition of the queue module (which consists of a list of routines with their parameters) how the data structure is implemented. It could be an array, but it could equally be a linked list. And if a linked list, it might be maintained by single links, by double links, or in a circular structure, etc.

The great benefit of this approach is that if the implementation details are hidden from client modules, the programmers of these client modules cannot take advantage of them. For example the user of such a module cannot directly read the value of a variable defining the current length of the queue. Such a variable might possibly exist in the implementation of the queue, but it is equally possible for example that in an array implementation the "length" enquiry calculates this value by subtracting a base of queue pointer from a top of queue pointer[49]. Thus the information hiding principle gives the implementer of such a module considerable freedom both to choose a suitable implementation and to change that implementation later – without affecting the client programs or modules using the queue module via its interface routines.

Although this technique is remarkably simple to state and is usually easy to use, it was ignored for many years by most system designers, who in general preferred the flow of control decomposition technique. Most importantly, it goes a long way towards eliminating the many problems associated with the flow of control technique discussed in the previous section.

A simple-minded application of the information hiding principle to the input-output device table problem discussed in the previous section would result in the introduction of a new module with interface routines which include enquiries for providing the remaining modules with indirect access to the information they require and with operations which on request modify the information. In reality there would be much to criticize in such a simple-minded design, as will become evident later, but these criticisms are unrelated to the information hiding principle.

Despite its shortcomings a decomposition of the system which simply hides the details of the input-output device tables would already solve most of the problems which arise in its flow of control counterpart, as we shall now show, using the same enumeration as was used to describe the problems.

(i) The first problem which we encountered was that the specification of the system design decomposed according to the flow of control principle is difficult, because every detail of the very complex table would need to be specified. While we have not discussed specification techniques in any detail, readers will realize that a specification which is expressed in terms of routine calls is much simpler to achieve than one which involves the extensive use of shared tables. For example it is much easier to convey the meaning of two routines for allocating and deallocating devices expressed along the lines:

```
routine allocate_device (device#:int)
routine deallocate_device (device#: int)
```

[49] In fact the calculation is a little more complicated than this if the queue is allowed to wrap around in the array, but that does not affect our point.

than it is to convey the significance of the setting and unsetting of bits (or the noting of job numbers to which devices are allocated, etc.) in tables. Nevertheless, the information hiding technique does not fully solve the specification problem, which is a difficult issue that we shall shortly discuss in more detail.

(ii) In the flow of control approach the level of communication between the implementers of separate modules is likely to be high, because of the need to clarify ambiguities in the meaning of values in the shared data structure. But with the information hiding principle the input-output device table is no longer visible to the designers of other modules, and the routine interface is likely to be much easier to understand. Consequently it is to be expected that the need for communication between implementers of separate modules will be greatly reduced. Long discussions about the precise significance of particular values of words and bits in tables are avoided.

(iii) The difficult debugging problems which arise when an error in a data structure caused by one module manifests itself as an apparent error in another module sharing the same data structure do not occur in an information hiding system, because there is only one module which accesses each major data structure and which can therefore be responsible for errors in the data structure. Consequently the search for the error is confined to a single module and to one programmer who understands the module. (Here we are assuming here that the computer's basic protection mechanisms ensure that no other module can directly access the internal data structures of another module. It will be shown in a later chapter how this is achieved.)

(iv) In an information hiding environment, the verification of module correctness is much easier than in systems which use the flow of control technique. On the one hand testing environments can be constructed which are based simply around a generalised routine call mechanism rather than around specific data structures. On the other hand formal program proofs become easier (though not easy) because all the relevant information is collected together in a single module, and the program prover need not be concerned with side-effects from other modules.

(v) Synchronisation problems, which can become very complex when several distinct modules attempt to access a shared data structure, are likewise easier to handle. Because all access to the data is confined within a single module, a single programmer has an overview of all the interactions which need to be synchronized. This is the basis on which synchronisation techniques such as "monitors" [119] and "path expressions" [120] are based.

(vi) The difficult task of the maintenance programmer becomes easier when the information hiding technique is used, because related definitions (of both data and code) are contained in a single module, which can therefore be understood without reference to the texts of a large number of other modules.

(vii) Adapting and extending systems designed according to the information hiding principle promise to be easier, because both modules and the interactions between them are easier to understand, and because in many cases a change to a data structure will have only local effects on the information hiding module and on a new client module which needs the additional information. It also becomes straightforward to add additional routines to a module interface without affecting existing software.

(viii) Finally, in contrast with systems designed around the flow of control technique, optimisation of a data structure and its related access routines can be undertaken with the confidence that only a single module need be changed. Provided that the new version faithfully implements the same interface as the former version, other modules remain unaffected.

Compared with the flow of control technique for module decomposition, the information hiding technique offers many benefits, at least for systems which manipulate substantial data structures. The reason for this is clear. The information hiding technique is an expression of the central idea in general systems design that the complexity of interactions in a system is kept within manageable bounds by clustering together into a single subsystem the components which have the greatest need to interact with each other.

In this case the components which clearly have the most intensive interactions with each other are the major data structures and their access routines. The main mistake in the flow of control decomposition method is to separate these strongly interacting components into different modules.

7 Abstract Data Types

The key concepts of the information hiding principle reappear in an idea known as *abstract data types*. This takes the further step of allowing a module that has been defined according to the information hiding principle to be treated as a type definition.

Just as in normal conversation the word "type" is used to indicate the common features of similar objects, so in programming language jargon a *type definition* defines the features of variables with similar characteristics. Given a type definition, a programmer can declare individual instances of that type and thus determine their behaviour.

"Typed" programming languages are languages which support such a concept. However, not all typed languages support abstract data types. Most of the conventional programming languages, for example Pascal, C, Fortran and Cobol, support only certain standard in-built types. These are types the behaviour of which is fixed by the definition of the programming language, such as booleans, integers, reals, and characters (which usually map directly onto the types supported at the hardware level).

A type definition principally determines the range of values which an object or variable can validly have and the operations which may be validly carried out on them. For example variables of the type integer have a fixed range of valid values (usually determined in practice by the size of a computer word) and there is a fixed set of valid operations defined for integers, such as addition, subtraction, multiplication, and so on. The compiler for a typed language can usually determine at compile time whether the operations on variables which appear in the program that it is compiling are valid. For example the statement

a = b + c

is valid if a, b and c are integers. This statement means that the integer values stored in b and c should be added together and the result stored in an integer variable called a. It is valid because there is an addition operation defined for the type integer. (In fact the compiler can determine at compile time that the operation is a valid operation but it is only possible at run time – when the addition is carried out – to determine if the result of the addition is within the range of values determined by the type integer.)

On the other hand the compiler can at compile time recognize that the statement

a = b ÷ c

is invalid if a, b and c are integers, because the normal division operation taught in schools is not guaranteed to return a valid integer value. This is because the result of dividing one integer into another integer can result in a fraction. For example 2 ÷ 3 gives a result which is a fraction, not an integer (recall that an integer is a whole number). Thus the "normal" division operation is usually not defined as a valid operation for the type integer (but it is for the type real) and the compiler can recognise mistakes of this kind at compile time.

Languages which are defined in such a way that type rules are always rigorously enforced by the compiler are called strongly typed languages. There are weakly typed languages (such as C) which are not quite so rigorous, and there are some completely typeless languages, where anything goes, such as Smalltalk.

A language supporting *abstract* data types allows the programmer to go a step further by introducing new types to supplement the in-built types. For example if the definition for a queue module which we used in Figure 13.2 is regarded as an abstract data type, this can then be used to define not only a single queue, but a type of object called a *queue type*. Then the programmer can declare many queues in his program (just as he can declare many integers), all of which can be operated on only by calling its routines, i.e. the operations *enqueue*, *dequeue*, *length* and *first*.

To illustrate this idea further let us now define a bank account type as an abstract data type. We base this on the example which was already used when we were discussing protection models, and which was illustrated diagrammatically in Figure 2.7. A simplified version of this is now shown in Figure 13.3.

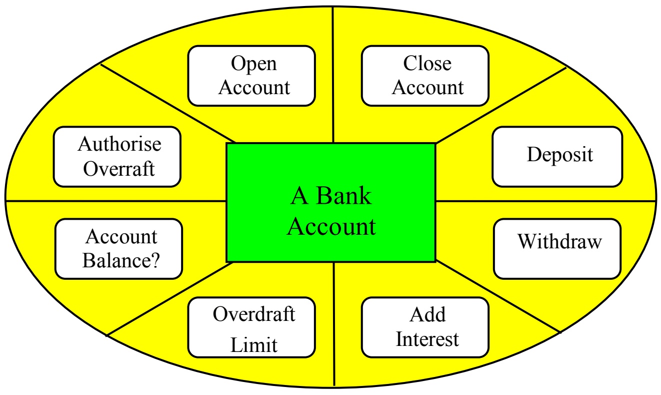


Figure 13.3: A Bank Account as an Abstract Data Type

There are several important advantages to be gained by the use of abstract data types in programs. These include the following.

Routines and their associated data structures are defined together as a recognisable structural element of the program. This is important for those – such as maintenance programmers – who need to understand a program. It is usually not possible in programming languages which do not support abstract data types, such as Pascal or C.

Data structures are hidden from other parts of a program, being accessible only indirectly via the interface routines of the type. This is the information hiding principle showing through, and brings the benefits which we have already described.

The same definition can be used for the declaration of many variables. For example, many bank accounts can be created and accessed using the definition in Figure 13.3.

The definition can be framed in terms of "semantic" operations. The operations defined for the type bank account, for example, correspond to the "real life" operations on bank accounts. (At least they are intended to. If they don't it is only because I am not an expert in banking systems!) This again makes it easier to understand programs.

Inappropriate operations are excluded. The compiler can check that only the operations defined as interface features of the abstract data type can be invoked in programs.

8 Specifications and Implementations

There is one further advantage which I would like to have been able to add to the list of benefits of abstract data types. That is, that abstract data types offer an attractive way of defining components which can be widely re-used in many programs. This would be comparable for example with the way an electrical engineer uses electronics components. If he needs an AND gate or a multiplexer or a ROM or a seven segment number display, he looks in the appropriate data books for a suitable component, finds what he wants and then incorporates it into his system.

In the same way, we could imagine that programmers who need a queue module or a bank account module or a calendar module, etc. could look up definitions for appropriate abstract data types in software data books, order the code and use it in their programs. But this unfortunately does not happen in practice. There are many reasons for this, most of which need not be discussed here. But one of them is important: most languages which support abstract data types (and most object oriented languages, which we shall discuss shortly) do not provide a clear separation between the specification of a type and the code implementation of it. This is unfortunate, for at least four reasons.

First, without a separate interface specification it is difficult to produce useful software data books. To include the entire implementation code is undesirable, because it does not provide a clear overview of the module. Furthermore, it would then be possible for potential customers simply to copy the code, so no-one would actually have to buy it, which is unlikely to encourage the development of a software components industry!

Second, the absence of a separate specification is not conducive to the idea of using different implementations of a module in different situations (e.g. one version optimized for speed and another for memory usage, or one version suitable for a module containing only a few elements and another capable of handling large numbers of elements). In order to make such different implementations interchangeable they must have a common specification.

Third, without a clear specification technique it is difficult for a system designer to define the types which he needs and then hand over a specification for them to the programmers whose job it is to provide an implementation.

Fourth, without an interface specification the programmer of a client module must examine the code of the modules he uses in order to know how to use the interface.

One of the reasons why most languages do not clearly distinguish between specifications and implementations is, as we mentioned earlier, that complete formal specifications are feasible only for relatively small and simple modules.

However, there is an alternative – to use those parts of a type definition which define how to use the interface (e.g. the routine headings) as a partial specification, and to supplement this with further useful information that can be provided formally and/or informally. Supplementary formal information can for example be provided to define pre- and post-conditions for routines (i.e. conditions that must hold before and after their execution) and also as invariant conditions for the entire type (which must always be true). Supplementary informal information can be supplied in the form of comments, describing in natural language aspects of the specification which cannot be expressed formally.

9 Object Oriented Programming

Abstract data types have also been incorporated into object oriented (OO) programming languages such as C++ and Java. The terminology changes a little but most of the ideas of abstract data types are carried over. Instead of talking about types it is usual to refer to *classes*. Similarly the instances of a class are called *objects* rather than variables. This emphasises the idea that the instances of an abstract data type can represent real world objects. The routines associated with a class are called its *methods*.

Unlike abstract data types, classes in most OO languages have unfortunately dropped the strict information hiding requirement of permitting only routines in the interface definitions, by allowing data types to be declared alongside methods. This saves programmers a little work in some cases, but it has the unfortunate effect that a class, unlike an abstract data type, cannot have different implementations. (This is also excluded by the fact that a class serves both as a type and an implementation.)

The most interesting new step which was introduced through OO programming is "inheritance". The basic idea of inheritance is that the objects of a class can "inherit" features from another object class. (We shall call the class from which features are inherited the *parent* class, and the class which inherits them a *child* class.) The relationship between these parent and child classes is often described as an "is-a" relationship. For example we might say that a student (child class) *is a* person (parent class) or that a savings account (child class) *is a* bank account (parent class). Typically OO (object oriented) languages allow parent classes to be *extended* and hence specialised in child classes. (A student is a specialised kind of person, a savings account is a specialised kind of bank account.) In this way a class person, for example, can be specialised in many different ways, e.g. as a student, a teacher, an administrator, etc. This allows the parent class defining person be re-used in all the various child definitions. And child classes can be treated in programs as if they were the parent class. Thus for example if a university wishes to send a letter to all persons connected with the university, it could obtain their addresses (which would be part of the parent class person) using the same code, regardless whether they are in fact students, academics or administrative staff. This is an example of the more general idea called polymorphism.

10 Qualifying Types

Qualifying types, an extension of object oriented programming, is the name of a software structure which is currently not widely known, but which will play a significant role in our later discussions of security and protection. An initial version of the idea was published in 1997 [121] under the name *attribute types*. The idea has since been extended considerably and incorporated into the Timor programming language, a new OO language developed to provide SPEEDOS with programming language support for the novel features which cannot be programmed in normal programming languages. An overview of Timor appears in [87]. In accordance with the above discussion (and in contrast with OO languages such as Java and C++) it strictly enforces the information-hiding principle.

Timor differs from other programming languages in that it supports not only the normal OO concepts but also qualifying types [122, 123] (and many other protection-related features needed for SPEEDOS [124, 125, 126]).

A *qualifier* is an instance of a qualifying type which has all the normal features of objects, including its own data and methods. But it also has some special methods, known as *bracket methods*, which are designed to bracket the code of other objects. There are two kinds of bracket methods, *call-in* and *call-out* brackets, which are activated differently from normal methods.

10.1 Call-In Bracket Methods

When one object calls a method of another, this can be represented as shown in Figure 13.4:

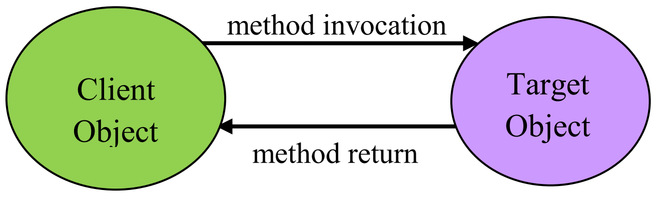


Figure 13.4: A Normal Method Invocation

A qualifier can be associated with a target object such that its call-in bracket methods can "catch" a normal method invocation before it reaches the target object (i.e. its qualified object), i.e. instead of the code of the method of the target being invoked, the code of the appropriate call-in bracket method is invoked

[122] (see Figure 13.5).

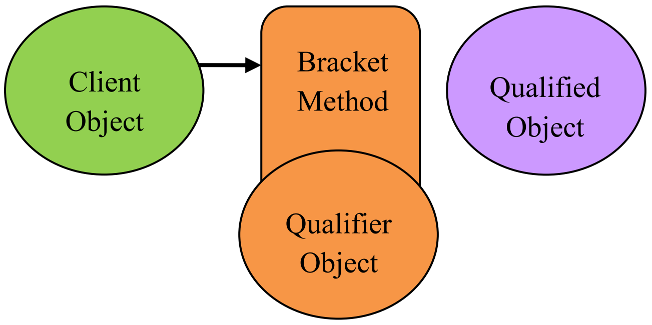


Figure 13.5: A Qualifying Type with a Call-In Bracket Method

Depending on how it has been defined, the bracket method may have access to the parameters which the client object intended to pass to the qualified object. But it has no access to the data of either the client object or of the qualified object.

10.2 The Body Statement

A call-in bracket method contains normal code, but it has one extra feature, called a *body* statement. The effect of this is to call the method of the qualified object which the client originally intended to call. This organisation of bracket methods gives its programmer a number of interesting options.

10.3 Augmenting Bracket Methods

Additional code can be added before calling the qualified object (in the part of the bracket method called a *prelude*). This code might for example access synchronising variables in the data of the qualifier, thus causing an unsynchronised qualified object to be synchronised [127]. Or from the security viewpoint it might for example maintain a log of calls to the qualified object which can later be printed out or analysed by another computer program to detect attempts to hack the qualified object.

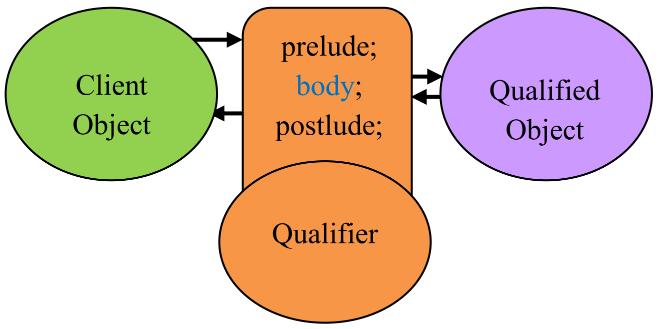


Figure 13.6: An Augmenting Bracket Method

When the method of the qualified object has completed its task, it returns to the *postlude* section of the call-in method (i.e. the statements following the *body* call). In the postlude section it can, for example, reset the synchronisation variables. This option, which augments the qualified object, is shown in Figure 13.6.

10.4 Testing Bracket Methods

Code in the prelude can check some condition (e.g. a security condition) and depending on the result might decide not to invoke the interface method of the qualified object. The result might be that the target object is not called at all. This is illustrated in Figure 13.7.

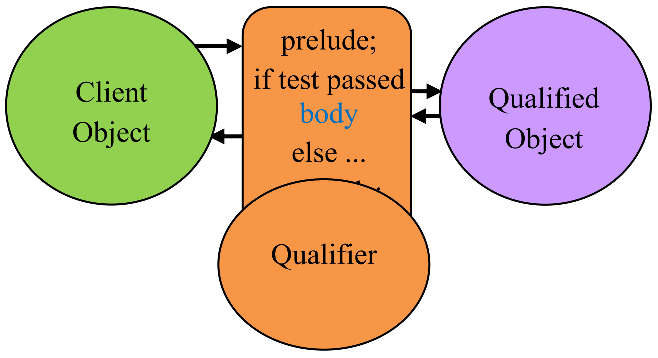


Figure 13.7: A Testing Bracket Method

10.5 Replacing Bracket Methods

Finally, the bracket method need not contain a *body* call at all (not even in a conditional statement). In this case the target object is in effect replaced by the qualifying object. One possible use of this is to set up a qualifier as a decoy which can be used as a disinformation technique. Figure 13.8 illustrates this possibility.

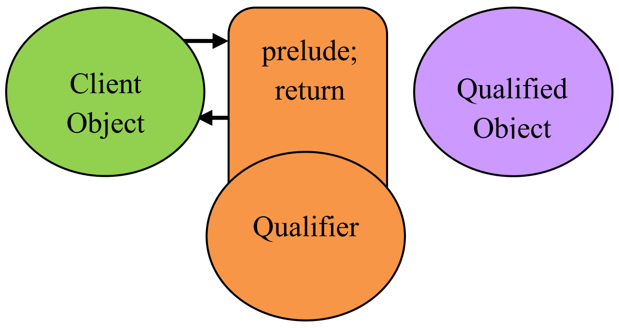


Figure 13.8: A Replacing Bracket Method

10.6 Multiple Qualifiers

More than one qualifier can be associated with a qualified object. In this case there is a defined order such that the first is invoked as a result of a routine call from a client object, the next is then invoked if this makes a body call, etc.; a body call from the final qualifying object (if it ever happens) results in the target object being called. The postludes are executed in reverse order.

10.7 Call-Out Bracket Methods

The principle of call-out bracket methods [123] is similar to that of call-in methods, except that

a) they are triggered by a call *from* a qualified object to some other object (the call-out object);

b) a *call* statement (cf. the *body* statement for call-in methods) is used if the call-out bracket decides to pass the call on to the call-out object.

The basic concept is illustrated in Figure 13.9, where a qualifying object has both call-in and call-out bracket methods. However, a qualifier can be programmed to have only call-in or only call-out routines if that is appropriate.

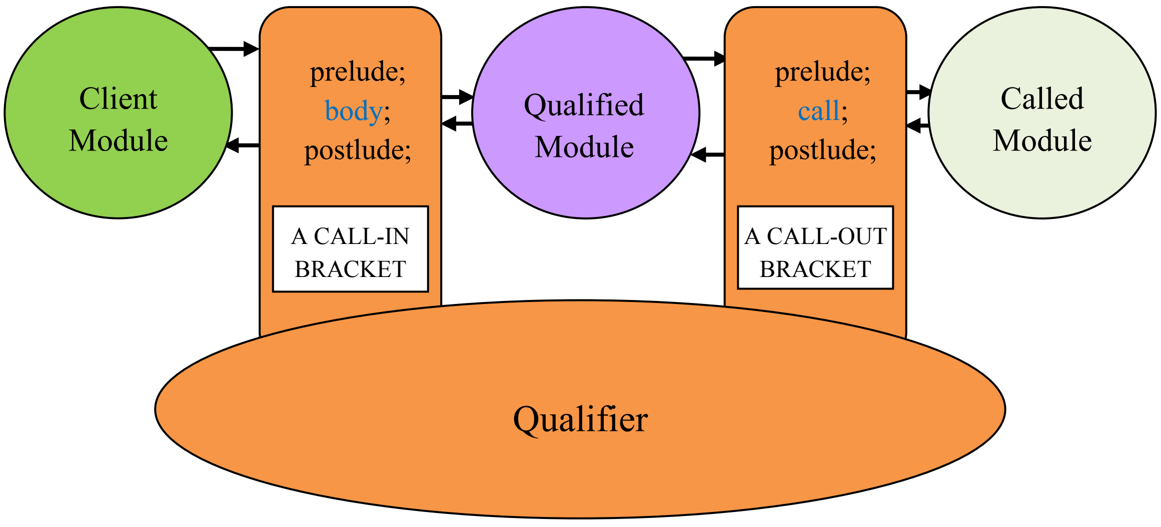


Figure 13.9: A Qualifier with Call-In and Call-Out Bracket Methods

Call-out brackets can be freely programmed to include or omit a *call* statement, and can optionally place it in a conditional statement.

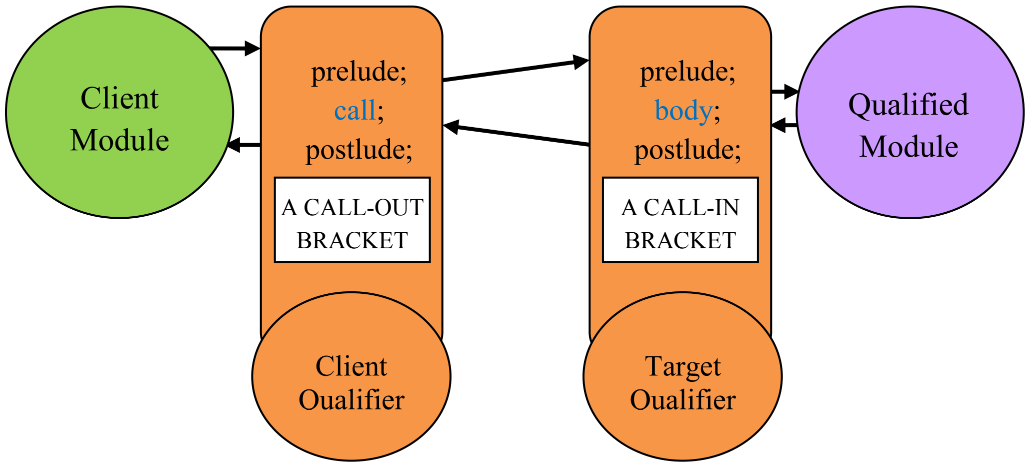


Figure 13.10: A Client with Call-Out and a Target with Call-In Brackets

At first sight it might be thought that call-out routines are superfluous, with the argument that they could be implemented as call-in brackets of the call-out object. However, this is not the case, because a call-in bracket is activated whenever the qualified object is called, while a call-out bracket is activated each time the qualified object makes a call to another object, not each time the called object is invoked. However both a client object and its qualified object can be qualified (usually, but not necessarily, by different qualifier objects), as is shown in Figure 13.10.

11 Conclusion

In this chapter we have discussed the software crisis which became apparent in the 1960s and is still largely with us today, despite many advances in software technology. A major reason for limited success of current software technology is that researchers have focused almost entirely on achieving improvements at the level of *individual programs*, whereas a major cause of the problems lies in the conventional methods used to structure software at the *system level*.

We have spent a good deal of space in describing both currently used object oriented programming techniques and an extension to the OO technique, called qualifiers, a feature of the Timor programming language which is not found in conventional OO programming languages. Although object oriented techniques, like most other software techniques which have been developed since the N.A.T.O. Conferences in the late 1960s, have been produced primarily to solve problems involving small and medium level software components, we shall see in the next chapter that these, unlike many other techniques, can be scaled up to provide solutions for the larger problems of software structuring at the level of operating system design.[50] What is even more important is that they also turn out to provide a framework which makes it relatively easy to improve vastly the security of future software systems, as we shall also see in the next chapter.

[50] If an application program is programmed in Timor it is possible to use qualifiers both within the individual program (implemented by the Timor compiler) and at the intermodule call level.

Chapter 14
Modules and Protection

This chapter builds on the ideas of information hiding, abstract data types and object orientation, discussed in the previous chapter, to develop a framework which will allow entire systems – not merely the content of individual programs – to be flexibly decomposed into modules with properties which not only eradicate the fundamental structuring problems in current systems but which can make a significant contribution to improving the security of software systems.

But before embarking on this task we observe that there are two quite separate and orthogonal aspects to software structuring which are often confused in practice. First there is the *static* structure of a software system, which is concerned with its decomposition into modules. These static modules can be viewed in isolation from each other as components which are separately programmable. In principle, and often in practice, it should be possible to re-use the same modules as components of different systems and to replace these modules with newer versions, for example to improve the efficiency of the system.

These modules can be compared with the physical components of other systems, such as the motor or the electrical system of a car. And like such major components of a car, major software modules are themselves decomposable into modules of smaller granularity. The use of object orientation as a method for decomposing large granularity modules into smaller granularity objects was discussed in the previous chapter. In this chapter we tackle the issue how complex software systems themselves can be decomposed statically into major units.

Independently of their static structure, software systems also have a *dynamic* structure, which is visible during the execution of code in a system. This is concerned not with modules but with *processes*. This dynamic aspect of systems is *not* discussed in the present chapter, but will be the subject of the following chapter.

1 Programs and Files

The ideas of information hiding, data abstraction and object orientation represent interesting and fruitful developments in the quest for better software structures. But they have been developed and used primarily as techniques concerned with the internal structuring of individual programs. In other words they have been used to shape the components into which traditional programs are decomposed, but they have scarcely influenced the structural design of large software systems. The conventional decomposition of software systems (as programs and data files) is still that in common use, and is the only approach supported by conventional programming languages, operating systems and computer architecture.

However, the decomposition of application systems into programs and data files as separate units is particularly harmful, because it creates separate major system components which have to interact with each other in an extraordinarily detailed manner. With programs serving as the major modules at the system level, interactions in the system take the form of file reads and writes. Consequently interfaces have to be precisely defined in terms of detailed file data structures which are used by programs that may otherwise execute independently of each other. The result is that instead of complex interactions being hidden within a module, they are visible on the interfaces between different modules.

This raises some interesting questions. Can the information hiding, data abstraction and object orientation techniques which are effective at the program component level scale up for use in substantial software systems? Can they serve as criteria for determining the major units when decomposing entire systems?

As we consider these questions it should be kept in mind that we are searching for software structuring techniques which will function well in a very large persistent virtual memory (see Chapter 12). It will therefore be helpful to free our thinking from the traditional dichotomy created by conventional computer architectures and by conventional operating systems, which divides the memory of computer systems into a computational virtual memory and a file system. A uniform persistent virtual memory gives us much more freedom to think about supporting unconventional software structures.

What happens if we use these software structuring techniques at this higher level of system decomposition? The most obvious change is that conventional data files can be replaced by information hiding, object oriented abstract data types. This is an idea which I first published with a research student as long ago as 1982 [128]. One of its major advantages is that it creates the possibility of basing file protection and security on the semantic operations associated with the file. This results in a much more powerful form of protection than conventional file system protection based on the right to read and/or to write files. That is a theme which will be developed in later sections. First we first mention some of the advantages which it brings in terms both of software engineering and of simplifications to the structure of operating systems.

2 Object Oriented Files

An object oriented file is in principle very similar to the small grained object oriented program components which we already considered earlier, except that the files which are most vulnerable to security problems often consists of a collection of smaller items, which we here call records. Thus instead of having a type definition which describes a single object, e.g. a single bank account (cf. Figure 13.3), a type definition for a file may (but need not) describe an object which consists of a whole collection of smaller objects, e.g. a file of bank accounts (cf. Figure 14.1). Several of the routines on the file interface are similar to those for an individual bank account, except that an additional parameter is needed to identify the account on which a particular operation is to be carried out. A further routine (coloured grey) has been added which operates on the file as a collection of accounts.

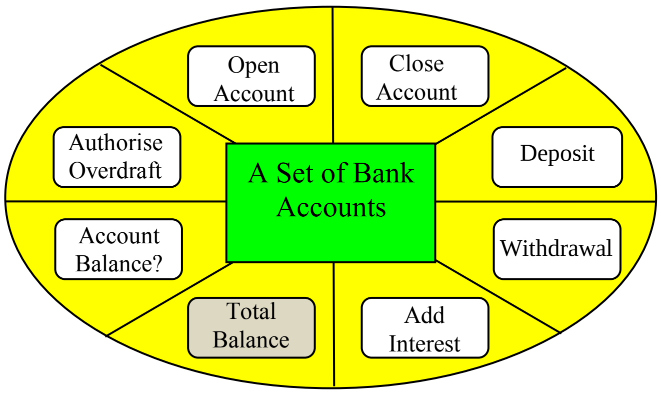


Figure 14.1: A File of Bank Accounts

No attempt at completeness has been made in this illustration. The important point is to see that it is possible, and relatively straightforward, to define a typical data processing file as an information hiding object with semantically appropriate interfaces.

With this approach the operating system no longer views files as raw data structures. Instead they appear as modules consisting of several code routines, which have separately defined entry points, together with an internal data structure that is not directly accessible to other modules. Furthermore, the code routines represent an implementation of an abstract type which can be used to define and implement many different file instances.

What effects does this approach imply for the design of conventional programs? First, they become smaller. Much of the code which is held as internal routines of conventional programs (see Figure 14.2) is transferred into file modules (see Figure 14.3).

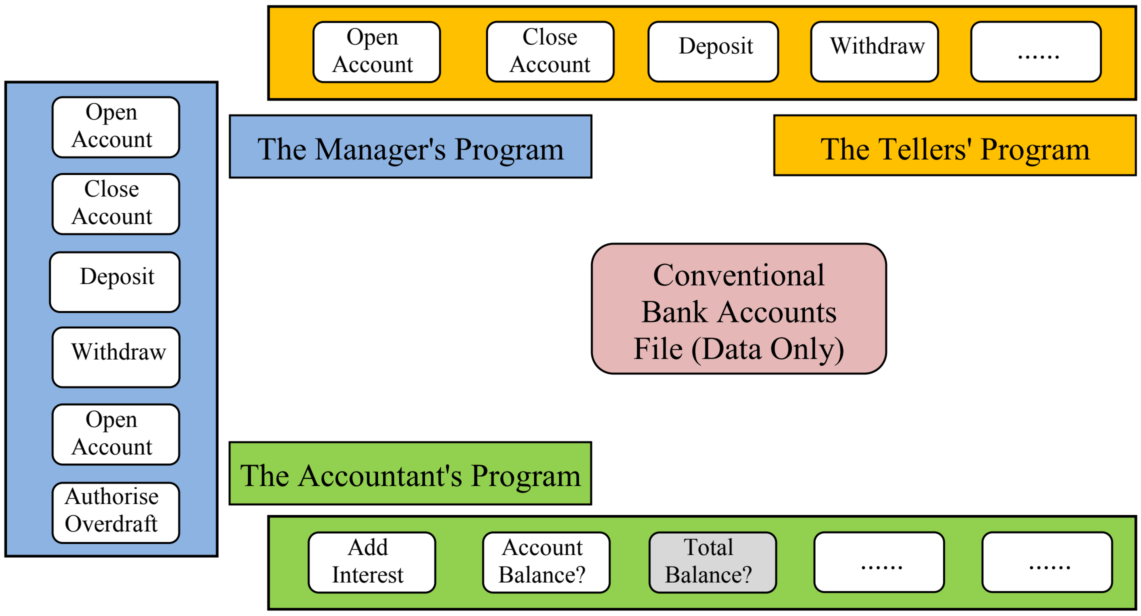


Figure 14.2: A Conventional View of a Bank Accounts File

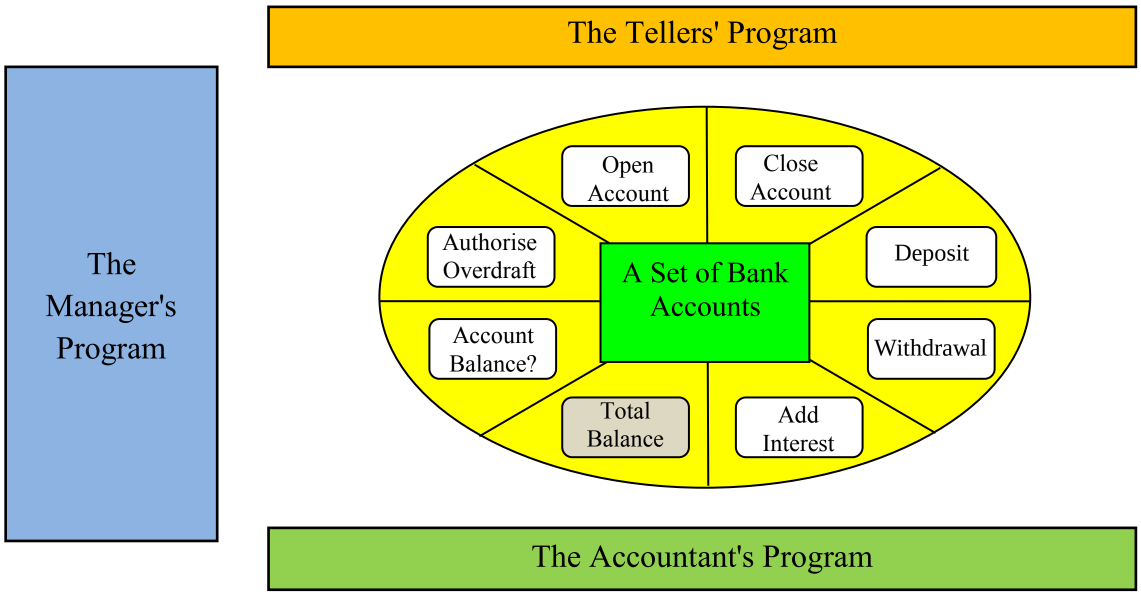


Figure 14.3: The Information Hiding Solution

But that is not the end of the story. In conventional systems the semantic routines appear as internal subroutines; hence they need never appear in a specification. This explains to some extent the mismatch which is commonly found between the application user's expectations of a system and the actual implementation. If the most important operations do not appear in software specifications, how can they be expected to be correctly implemented? What is worse, the same operation often appears (probably with different implementations) in several programs, as Figure 14.2 shows. The reason is that in conventional systems the semantic operations associated with a particular human activity or work role are collected together into a single program, because only entire programs – not semantic operations – can be protected in a conventional system. Since the same operations are often needed by several types of user with different protection requirements (in our example bank tellers, branch managers and head office accountants) they must appear in each of these programs. On the other hand the semantic operations on files appear only once in the scheme proposed above, i.e. within the file itself, as Figure 14.3 shows.

One effect of this structure is that programs can become much simpler, consisting largely of *control routines* which invoke the semantic operations of files. It is the absence of such a division of labour between programs and files which makes the maintenance of software systems far more difficult than it need be. This can be illustrated by considering the changes which banking systems, for example, have undergone over the last few decades as a result of a series of technological changes.

Early banking systems, like other early commercial data processing systems, were batch processing systems. In such a system 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 and then were read into the system. There the transactions were checked for consistency, reasonableness and so, and after that they were copied onto a second magnetic tape and sorted into the same order as the master file. In the next step the master file update program would 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.

The control code in the master file update program basically consisted of a large loop in which the next transaction was read and the appropriate routine 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 seen in computer rooms in science fiction 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.

The next stage in the development of banking systems was the introduction of on-line terminals for the bank staff. For those banks adventurous enough to introduce on-line updating of the master files on disc, a transaction processing monitor program was needed, which would read transactions from terminals, process them and update the master file – by this time a disc file with the relevant accounts being 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.

The next development was the introduction of ATMs (automatic teller machines), from which customers can directly initiate transactions. New programs were needed with new control routines to read in the customers' plastic cards, to check PIN 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.

Then on-line 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.

In the final banking development (at the time of writing) customers were given the opportunity to access their accounts from their smartphones. This implied yet another set of software developments in which banking routines had to be incorporated into new programs.

We see from this example how the separation of software into monolithic programs and data files is not adequate as a structuring tool. The alternative which we are proposing, i.e. associating files with their semantic operations and having separate programs which primarily consist of control code, is a natural consequence of rigorously following the information hiding principle. The resulting clear separation between semantic operations and control code makes it straightforward to modify the file modules when banking operations change or the control modules when for example the technology changes. Such a separation of concerns would go a long way to reducing the maintenance costs of SPEEDOS systems.

3 Protection Advantages

The substantial advantages of using semantic routines as a basis for defining semantic access rights was already pointed out in chapter 2 section 3.2. Later in this chapter we will see how SPEEDOS uses semantic access rights as one of the foundations of its protection mechanisms (see also [23]).

4 A Uniform Module Structure

In the previous sections we developed the view of a persistent file as an information hiding object which is characterised, from the viewpoint of the operating system, as a module with multiple code entry points and a hidden persistent data structure. As we shall now see, such a framework can be used not only to implement file objects, but can provide a general software structuring framework to implement any kind of major software resource which might be needed.

4.1 Programs

A *program* fits very nicely into this framework. Although conventional programs do not always need a persistent data structure, their structure can be regarded as a rudimentary form of the module structure under discussion. In fact most programs need some sort of a heap for storing data structures. Furthermore there are often special files associated with programs, such as a "preferences" or an "options" file, defining what options a particular user prefers when he is editing (e.g. default font, character size, style settings) or drawing diagrams (e.g. centimetres or inches, page size, guidelines and rulers) etc. Such information can be accommodated in the proposed module structure as a persistent data structure of a "program". In this case the program effectively plays the role of type manager for the preferences information. (Of course if such a data structure is non-trivial it should itself be implemented as a separate information-hiding file.)

The design of conventional programming languages and operating systems usually restricts programs to having a single entry point. However, the module structure under discussion can easily support multiple entry points. Assuming that an operating system design allows programs to have many entry points, which can for example be invoked by a command language interpreter (or an equivalent graphical interface), this will turn out to be very useful in practice. For example instead of having two nearly identical programs for calculating salaries, one for those employees who are paid weekly and another for those paid monthly, it becomes possible to have a single program with two (or more) entry points. Similarly a provider of software games need not produce separate programs for chess, draughts, fox and hounds, etc. He can instead sell a single program with different entry points for these games. When he develops this multiple entry point program it needs only one common graphics routine for drawing the chess board, for example. The interface for a games compendium is illustrated in Figure 14.4. Allowing a program to have multiple entry points can only make it more flexible!

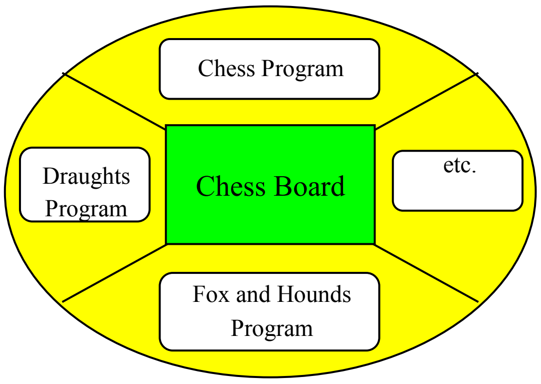


Figure 14.4: A Compendium of Games

Another point should be considered if programs are to be integrated into the same module structure as files. The single program entry point supported by conventional languages and operating systems is restricted by programming languages and operating systems to be either completely parameterless or to have a very special "parameter" mechanism, which for example only allows an input and an output "file" to be nominated. In practice it would be far more useful to allow a wider range of parameters, specified in the same way as parameters for other routines. Then it would become feasible for example to develop a "pocket calculator" program with multiple entry points, such that each entry point corresponds to one of the calculator operations with its input and its result being provided as normal parameters.

4.2 Subroutine Libraries

We now begin to see that the distinction between a program and a *subroutine library*, supported as a separate mechanism by most operating systems, is rather artificial. After all, a pocket calculator is just a simple subroutine library. The principle characteristic of many subroutine libraries is that they provide a related collection of routines which have entry points that can be independently invoked, and which simply carry out calculations. For such subroutine libraries the proposed module structure can be used without adding or changing anything. We refer to these as *external* subroutine libraries.

However there are subroutine libraries which have a somewhat different character, although they can still be adequately defined as information hiding modules. These are libraries which manipulate or help to organise the – often persistent – data of a client module (e.g. a character string library, a collection library). These need the ability to access and modify the data of some other module, and efficiency dictates that this access should be direct. The best way to view such libraries is not as independent libraries but as useful extensions to the code module of the file module for which they provide a useful service. In the sequel these are called *internal* subroutine libraries.

4.3 Operating System Modules

In conventional systems, *operating system modules* are usually handled by special mechanisms. With the proposed module structure, this becomes unnecessary. Let us consider a few examples, starting with a process scheduler module. Figure 14.5 gives an impression of how this can be implemented using a uniform module structure.

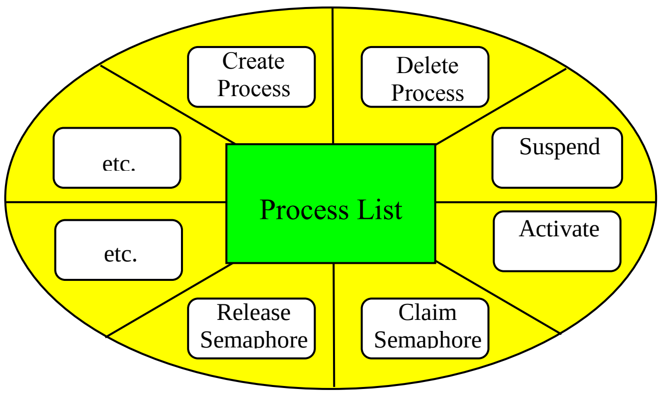


Figure 14.5: A Process Scheduler Module

Another kind of operating system module, a *file directory* (sometimes called a *folder*, but which we now call a *module directory*), is rather like a file. It also fits nicely into this structure, as Figure 14.6 illustrates.

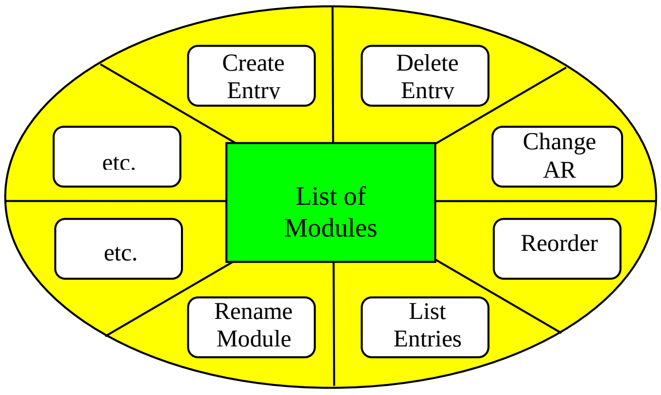


Figure 14.6: A Directory Module

Yet a third kind of operating system module, a *command language interpreter* (CLI) is structurally more like a program in older systems[51]. This too can be easily implemented within the uniform module framework under discussion.

[51] In modern systems the graphical interface also needs a mechanism to invoke programs.

In fact the functionality of a CLI can very usefully be extended slightly in a system of the kind now under discussion. In particular much can be gained if the CLI is capable of invoking a selected entry point not just of a program, but of *any module*, and of passing to it the appropriate parameters. In this way any entry point of any module can be regarded as a potential command which can be directly invoked by the user.

To achieve this technically is not particularly difficult. The details need not concern us here. A similar uniform module structure was realised in the MONADS systems. The resulting environment is one in which programs are not special entities. Any entry point of any module in the system can be invoked as a command, provided that certain template information has been supplied to allow the command name and the parameters to be converted into an appropriate internal format [129].

This kind of CLI has the advantage not only that any entry point of any module can be viewed as a command, but also that the CLI becomes a general module testing tool, because it is able to invoke any of the interface routines of any module under test. Even those entry points of modules which are never normally invoked as commands can be called in a straightforward manner and passed parameters to test their correct functioning. In contrast the testing of any kind of module except a program in a conventional system usually requires the tester to do considerable work to construct a suitable test environment.

4.4 Device Drivers

Hardware devices, such as printers, keyboards and monitor screens, are usually interfaced to the rest of an operating system and/or to application programs by software modules called device drivers. Such drivers are specialised software modules usually provided by the manufacturers of the hardware or by the operating system supplier. The interface between the device driver and the device itself can be quite complicated, but there is no reason why the driver itself cannot be designed as an information hiding module.

5 The Proposed Module Structure

The uniform module structure described above requires that all modules, however simple or complex, in principle require only two containers[52], viz.

– a data container, which may hold persistent data, but also temporary data created during the course of a computation carried out within the module, and

[52] Containers were introduced at the end of chapter 11 as a paged unit of virtual memory into which segments can be placed.

– a code container, which holds the code segments of a module, but which in practice can include a mechanism for switching control to certain subroutine libraries that provide assistance in manipulating and organising information in the data container.

How such containers are organised will be discussed in later chapters.

6 Simpler Operating Systems

Through adapting the ideas of information hiding, data abstraction and object orientation we have arrived at a module structure capable of implementing any kind of software resource needed as a major component of a software system. This framework has the remarkable property that it can equally well be used to implement files, programs, subroutine libraries and even operating system modules.

In conventional systems these different kinds of modules are usually implemented using quite different operating system mechanisms and thus make operating systems far more complicated than is really necessary. Part of this unnecessary complexity arises not only from implementing different module structures as such, but also from the fact that further mechanisms have to exist in order to allow these different kinds of modules to interact with each other. For each pair of module types which can be linked, at least one linking mechanism is necessary. These linking mechanisms are often all different. To link a program to a subroutine library is usually quite different from linking it to an operating system module and this is different again from linking it to a file, etc. In conventional operating systems the number of potential linking mechanisms grows as the square of the number of kinds of modules.

Not all potential linking mechanisms are implemented in practice in conventional operating systems. For example a subroutine library may normally not invoke a program and an operating system module may not invoke a subroutine library routine. So in practice the operating system is complicated not only by the fact that it implements modules in different ways but also that it has to provide a variety of different linking mechanisms. And even then it does not allow all linking possibilities!

By contrast the uniform module structure proposed in this chapter requires only one kind of mechanism to implement the module and only one mechanism to link any module to any other module, namely a mechanism which allows the code of a module to invoke an entry point of another module. Strange as it may sound, a "file module" can now call the operating system, or a subroutine library can invoke a program, etc. without any special mechanisms (but subject to the usual protection rules).

7 Protecting Modules

But what has all this to do with security and protection? The answer is that this single linking mechanism can be extended to create a powerful basis for checking access rights, and so provide a single very straightforward protection mechanism when any kind of software unit is called. A hint of this possibility was already given in Chapter 2 in the initial discussion of semantic access rights.

Semantic access rights determine which semantic routines may be invoked, i.e. which semantic routines of major modules of a system as discussed in earlier sections of this chapter. A uniform protection mechanism based on semantic access rights can be embodied in the module invocation mechanism. That is, when a process executing in the code of a major module invokes a semantic operation of another module its right to make the call can be checked by the kernel. The implementation of such checks can in principle be based either on capabilities or on access control lists.

8 Capabilities or Access Control Lists?

As was described in Chapter 2, capabilities are stored with subjects and they name the objects to be protected, while ACLs reside with the objects and contain lists of subjects. Maintaining lists of subjects is potentially a rather complex matter. Subjects, as we saw in Chapter 2 and will see in more detail in the second volume, are not necessarily simply users. When a module is invoked, the right to invoke it may be vested in the user process, in the file module making the call or even in the code module implementing its semantic routines.

For example in a banking system security might be enhanced by ensuring that the right to call some semantic interfaces of a bank accounts file does not (only) depend on the identity of the calling user, but also (or only) on the identifier of the code module accessing the interfaces, thus ensuring that these are not being called from a hacker's program.

From the viewpoint of the kernel's design, carrying out such extensive checks based on access control lists would lead to an extremely complicated kernel. This complexity can be avoided if the checking of access rights is based on capabilities rather than ACLs (provided capabilities can easily be associated with different kinds of subjects, which we shall demonstrate later).

For this reason the protection of modules in SPEEDOS is based on the possession and presentation of capabilities for modules, not on ACLs. We now consider some of the implementation issues involved in this decision.

9 Module Capabilities and Inter-Module Calls

A module capability (see Figure 14.7) consists of a unique module identifier, an associated set of permissions and some status bits. The list of permissions can be divided into three basic groups. The first group contains the list of permissions to call the entry points of the semantic routines associated with the module[53], the second contains generic access rights associated with the module and the third contains the metarights associated with the capability itself (see Chapter 2). In Chapter 16 these fields are discussed in more detail.

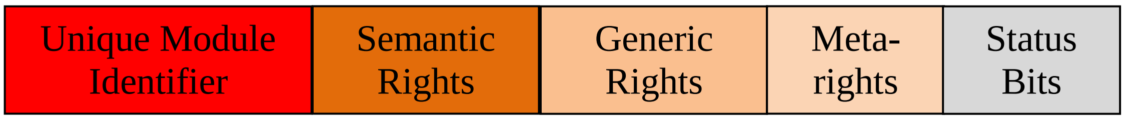


Figure 14.7: The Basic Structure of a Module Capability

In earlier capability systems the capabilities were normally stored in capability lists (C-Lists). However, they can be more flexibly used if they can be stored like other data in user modules. This is possible using the idea of partitioned segments (discussed in Chapter 10 section 2.4), which allow not only simple pointers (as discussed there) but also capabilities to be stored in any segment of any container. They are protected from arbitrary changes by the fact that they can only be created and managed by the SPEEDOS kernel, which only makes the data sections of partitioned segments directly accessible for normal user access.

In order to make an *inter-module call* (i.e. to invoke a semantic routine of some other module) the calling process/module must provide the SPEEDOS kernel with a capability which (a) uniquely identifies the module to be called (i.e. the unique module identifier in Figure 14.7) and (b) contains a list of the entry points which it may legally access (i.e. the semantic access rights in Figure 14.7). Thus the module capability is used as an operand for the inter-module call instruction. As a second operand the caller nominates the particular semantic operation to be invoked. It then becomes a function of the kernel to implement the inter-module call instruction in such a way that the call may proceed only if a permission for the requested semantic operation is contained in the list of semantic access rights.

A simple implementation of this might involve numbering the entry points of each major module with integers starting at zero. Taking the bank account file module in Figure 14.1 as an example the operation "open account" might be

[53] This implies that, in accordance with data abstraction and information hiding principles, the interface of a major module is always framed in terms of semantic routines. Direct access by one module to the internal data of another module is not permitted (in contrast with the laxer conventions used for small grain objects in most OO programming languages.

numbered 0, "close account" 1, "deposit" 2, "withdraw" 3, and so on. Then the semantic access rights in the module capability are implemented as a "bit list" in which each bit represents an entry point. Bit 0 in the list indicates whether the operation "open account" can be called, the next bit whether "close account" may be called, and so on. If the appropriate bit is set to one the presenter of the capability has permission to invoke the corresponding entry point of the module, but if it is set to zero the entry point may not called. This is illustrated in Figure 14.8. The unique identifier in the capability indicates the container for the file data of the module. (From this the code container can then be located.)

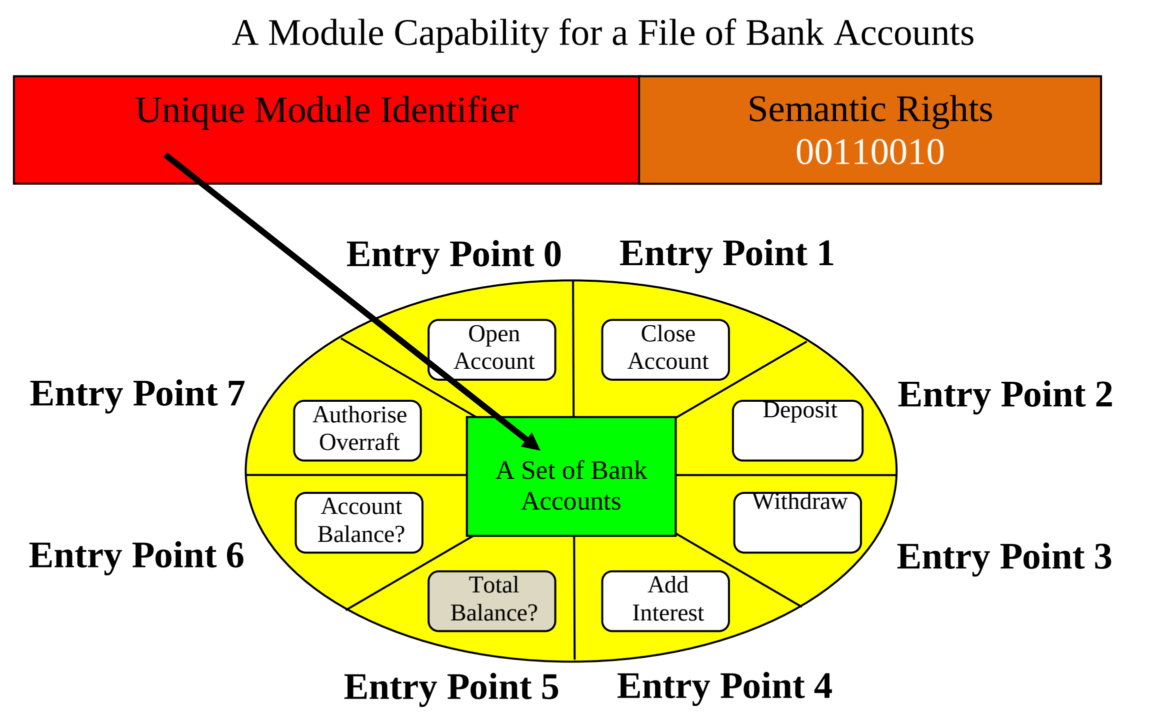


Figure 14.8: A File of Bank Accounts

10 Protecting File Modules

One of the basic ideas behind the object oriented philosophy is that a system may contain many objects which are instances of the same type. This principle applies not only to the small and medium granularity objects found in conventional object oriented programs, but also when object oriented techniques are used as a tool for decomposing systems into major objects. For example, there may be many bank account files in a system (e.g. files for each branch of a bank). Thus the protection of semantic operations must be organized on the basis of individual files rather than on the basis of the code module implementing the operations.

This implies that the module number in a *file capability* (i.e. a module capability for a module with persistent data) refers to the unique identifier of the container holding the persistent data structure part of the module, i.e. a persistent heap container number. From this the inter-module call mechanism must then be able to locate the container of the associated code module (i.e. a code execution module produced by a compiler) in order to activate the required entry point (subject to the access rights held in the capability) but of course only in association with the appropriate file. This is illustrated in Figure 14.9.

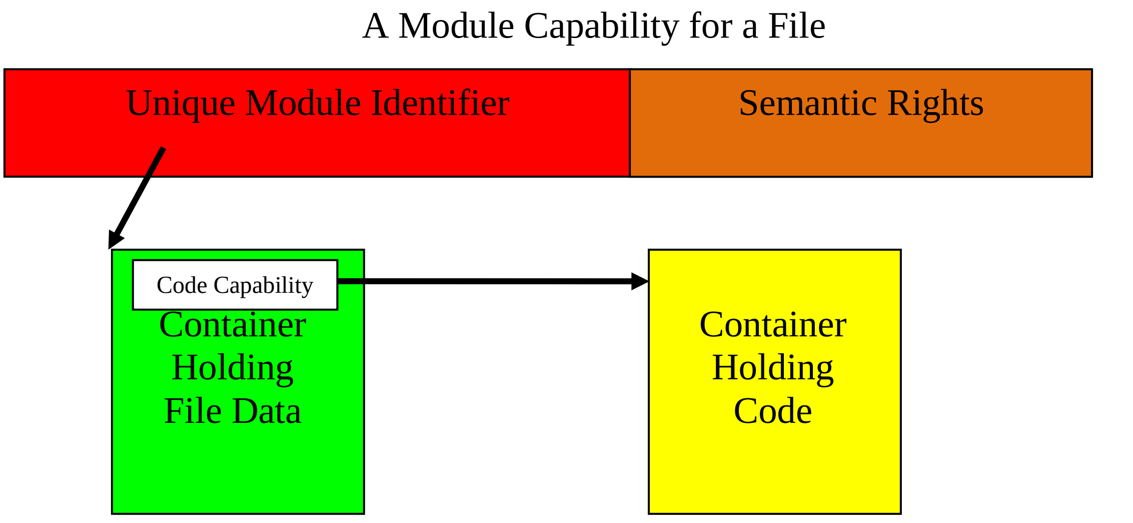


Figure 14.9: Calling a File Module

Notice that with this organisation the right to access a file module implies the right to use the associated code module. This is essential to guarantee protection based on the information hiding principle. If the caller of a file module were permitted to nominate an arbitrary code module for use with a file then the entire basis of semantic protection would be undermined. The kernel design must guarantee that this requirement conforms with the protection of proprietary software at the time a link is set up between a file module and its code module, which implies that the pointer to the code module should be a *code capability*.

When a semantic routine is activated, the calling process is given access to the persistent data in the file container. The run-time code created by the compiler can also set up an internal process stack, etc. in the file container. Information about the code organisation (e.g. entry points for the semantic routines) is held in the code container and the kernel activates the appropriate semantic routine.

11 Protecting Code Modules

The simplest kinds of modules in a system of the kind envisaged in this chapter are *code modules*, i.e. those code modules which do not encapsulate a persistent data structure (other than constant segments). If used directly (rather than via a file capability) these correspond to both programs and subroutine libraries in conventional systems, since they may have more than one protected entry point. For them the protection mechanism is very straightforward. Assuming that all code is re-entrant, there is no need to create a separate "instance" of the module before invoking one of its entry points. However each such code module does have a single "temporary" data file permanently associated with it, in which the temporary information generated by processes (e.g. internal process stack) is stored as the code is executed. In this case the unique module identifier supplied as an operand to the inter-module call mechanism identifies the temporary container.

This mechanism can be used to validate the right to use proprietary software. In order to invoke a code module the caller (e.g. a CLI on behalf of a user) must present a module capability containing the correct access rights to make the requested call. Thus only a user who actually has permission to invoke the appropriate entry point into a program can do so. If he has no right to access proprietary software (or if the access rights limit him to a subset of the routines) then the inter-module call mechanism prevents the misuse of the software by refusing to carry out an invalid call.

12 Protecting Internal Objects

With the decision to provide a basic protection mechanism which works at the file level, it might appear that we have created a different problem. It could be argued that the objects which require protection are not the file objects in a system but the smaller granularity objects within the files. For example, it may seem that what needs to be protected in a banking system are not bank account files but the individual accounts in the files. While this view has some merit, we must keep firmly in mind that the discussion in this chapter is concerned solely with the basic protection mechanism which is centrally implemented to control interactions between the major modules of a system. A central protection mechanism in a kernel must provide certain basic guarantees about the security of a system, but it cannot be treated as a substitute for application implemented security measures. What is important is that the central mechanism does not interfere with or restrict such additional measures.

13 Conclusion

We have now established an alternative framework for statically decomposing software in a secure system. It involves defining all modules as information hiding object oriented modules in the persistent virtual memory. In this way higher level protection can be based on the right to invoke the semantic operations of modules (with the memory architecture guaranteeing that a process executing within a module is confined to accessing only segments related to that module).

But the significance of this model is not merely that access rights have been defined in terms of semantically appropriate operations. Although this alone represents a significant step towards building secure systems, it also has some further advantages which should not be overlooked.

First, it binds a specific code module to a specific file. The significance of this should not be underestimated. In conventional systems there is no rule about which code modules can be used with which files in the file system, and that goes a long way towards explaining how hackers can write their own "hacking" programs and then use them to access files.

Second, in one fell swoop an enormous amount of complexity which exists in conventional systems has been eliminated. The file system has all but disappeared. The complex handling routines for different kinds of software units in conventional systems (and the mechanisms for linking between them) have also been replaced by a single module structure and a single linking mechanism (i.e. the inter-module call mechanism). As was argued in Chapter 1, a good part of the security problem is created simply by the existence of multiple mechanisms and the complexity which goes with it.

Third, the issue of structuring the data efficiently for "files" (e.g. indexed sequential, B-Trees, etc.) has now been relegated to the compilers. To see how this can be done, see the Timor language description [87][54].

Fourth, although we still have to discuss how the persistent virtual memory can be efficiently implemented, we have at least in principle eliminated the inefficiencies of conventional memory by providing direct addressability.

We now turn to the question of organising processes in the virtual memory.

[54] The Timor language description can be downloaded at the Timor website (https://www.timor-programming.org/)

Chapter 15
Processes and Protection

The previous chapter emphasised the static aspects of a system, showing how information hiding modules can form a basis for designing secure systems. But that is only one side of the coin. To be useful the software must be executed on a CPU. Here we consider in more detail than previously how this more dynamic aspect of a system might be organised in a secure way.

1 Process Structures

In Chapter 8 section 9 two ways of decomposing an operating system into processes were described. The first and most widely used technique (*out-of-process*) involves having a separate process for carrying out each operating system activity. Applied to the concepts developed in the previous chapter it would mean that a separate process is needed for each major (code, operating system and file) module in a system.

The alternative is the technique implemented in the B6700 (*in-process*), where operating system services are executed in the application's own process. This means that the major modules of a system do not each have a process of their own but are invoked as routine calls within the application process requiring their services.

Lauer and Needham [47] concluded that these alternative process models are duals of each other. But as was already mentioned in Chapter 8, they have some fundamentally different characteristics both with respect to protection and security and with respect to their dynamic properties. Before examining the security aspects we take a more detailed look at some of the more important dynamic properties of the two models.

1.1 Dynamic Process Properties

With the out-of-process model each module provides a set of services in its own process, i.e. *out of* the application *process* and on its own stack. When a module (the client module) requires the services of another module (the server module) it *creates a message* indicating which service is required, followed by further information relevant to the request.

The equivalent in-process action is for the client module to call a routine *in* the application *process* (i.e. on the stack of the application process) where the required service corresponds to the destination routine of the service module, and provides as *parameters* on the stack further information relevant to the request.

In the out-of-process case a process stack is associated with each module whereas an in-process stack potentially contains information relating to many modules. As was explained in Chapter 8, the in-process organisation also potentially leads to greater parallelism.

But more significantly from the viewpoint of protection, an out-of-process module is normally executed as a continuous loop, taking messages from clients one after another out of its message buffer (see Figure 8.6), analysing these in turn to determine which service is required and then processing them one by one. Consequently the individual services are not visible at the architectural level, but only within the module's request analysis code. However the services corresponding to these encoded messages are in fact equivalent to semantic routines. But since they are not visible at the architectural level they cannot be protected at the architectural level in the manner described in the last chapter.

On the other hand, if a module is defined as an information hiding module, the semantic routines are visible as interface routines of the module, and can be directly invoked individually. This corresponds well with the in-process model, because in that also modules are invoked via their individual routines.

A further significant point is that in its normal implementation the out-of-process model requires a separate process for each service module. With the introduction of a persistent virtual memory, each file in the system in effect becomes a service module for its users. The implication of this is that each file in an out-of-process system would also have its own process. Consequently the process scheduler (the most important module in the system from the viewpoint of efficiency) would either be cluttered up with an extra process for each file in the system, or would have to create and delete processes dynamically as users open and close files!

We conclude therefore that the in-process model and the idea of information hiding modules are natural partners which both lead to greater efficiency (through greater potential parallelism and less process scheduler overheads) and, above all, to better protection (through semantic routines).

1.2 Further Advantages of the In-Process Technique

Some further characteristics of processes are affected by the choice of process structuring model. For example, in a system where users are charged for the CPU time they use or have a limited budget of CPU time at their disposal, the process scheduler must keep a record of how much time is consumed by each user. In an in-process system the time used by an application process corresponds to the sum of the CPU time spent executing his own program and that spent on his behalf in operating system service routines (and, in our model, while accessing files), because the process scheduler is not involved in module invocations and therefore cannot (but also need not) record these. The effect is that the process scheduler records the amount of time each user genuinely costs the system.

But in an out-of-process system the CPU time used by an operating system process cannot easily be charged to the user who requests operating system services, because an operating system service runs continuously regardless of the user for whom the service is being performed, and therefore the amount of CPU time consumed by different users is not known to the process scheduler. To provide a more accurate measure based on individual users would create even further inefficiency in the system. While the amount of time spent in operating system modules by all applications taken together (which would be what the process scheduler can easily calculate in an out-of-process system) may be of interest for statistical purposes, it is useful neither for budgeting nor for charging purposes.

Similarly the *priority* of a process in an in-process system represents the user's priority, regardless whether it is executing in the application program itself or in an operating system routine. But in out-of-process systems it is usual to give operating system processes a higher priority than user processes. To understand why the in-process form is the natural choice, consider the (deliberately unrealistic) example of a system in which a nuclear power station is being controlled and a payroll application is also active in the same system. It is self-evident that the nuclear power station application should always take precedence over the payroll application, even when the payroll application is using operating system services. (In fact the arguments are rather more complex than I have suggested, but the main point is evident.)

These points suggest that the dynamic properties of the two process structuring models are not only different, but also that the more efficient and more natural model is the in-process one, which forms a natural partner with information hiding modules.

2 Managing Inter-Module Calls

The stack structure needed conceptually to support an inter-module call is shown on Figure 15.1.

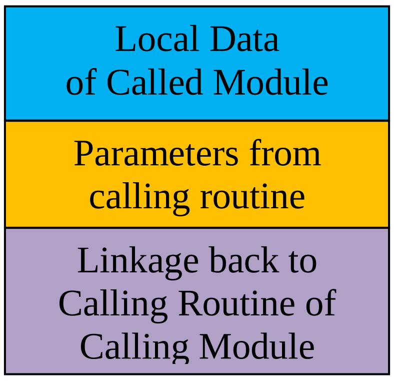

Local Data of Called Module
Parameters from calling routine
Linkage back to Calling Routine of Calling Module

Figure 15.1: Stack Support for an Inter-Module Call

2.1 Linkage

The linkage segment contains the information which is later needed to enable the kernel to complete the corresponding inter-module return instruction. This will include at least the following items:

– the unique identifier of the calling module,

– the number of the calling module's calling routine,

– the offset within the calling routine at which execution should be resumed on return,

– sufficient information to allow the calling routine to reload its former register values.

At this stage it is not necessary to determine the exact details of the linkage. For example there are a number of possibilities for determining how the former register values can be restored. Such details are determined by an actual kernel design and are discussed for SPEEDOS in chapter 20 in volume 2.

2.2 Parameters

The parameter segment contains the parameters which will be made available to the called routine of the new module. These can be module capabilities or values. However, within-container pointers may *not* be passed as parameters, since that would give the called routine access to the internal data of the calling module. Not only would that be an infringement of the information hiding principle, but it would also lead to a situation in which individual containers could not be independently garbage collected. That is an especially significant point in a system which supports a world-wide persistent virtual memory. It will be shown in later chapters that this apparent restriction is a non-problem.

As in the case of linkage, the details of how parameters are passed are not relevant at this point. For example, the RISC philosophy requires that as far as possible these should be passed in registers (which in our case means the normal general purpose registers). However it is already clear that just as pointers may not be passed as parameters, so also segment registers must be invalidated on an inter-module call. It is also clear that module capabilities passed as parameters must be passed in a segment on the stack, since there is no provision for special registers for these.

2.3 Local Data of Called Module

The kernel must set up an appropriate environment in which the semantic routine of the called module can begin to execute, including where appropriate providing access to its persistent data. The organisation and stack structure of the local data will be left to the various compilers and will be held as part of the module's data, i.e. *not* on the process stack (see section 4 below).

3 Persistent Processes

The rest of this chapter uses a minimal notation, which we call a module stack frame, as shown in Figure 15.2, which is all that is necessary to discuss the significance of making processes persistent.

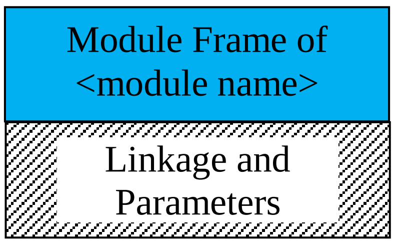


Figure 15.2: A Module Frame

A process is defined as a module in a container of the virtual memory which holds its stack(s)[55]. Since containers are implemented in the persistent virtual memory, an interesting new property emerges. Process modules, like other modules, persist over time, i.e. they are *persistent processes*. (In conventional systems processes cannot be regarded as persistent because they exist only in the computational virtual memory, which is not persistent.) Furthermore a user (given a capability with the appropriate permission) can have multiple persistent processes in separate process modules. This arrangement introduces some new possibilities, including a much better protection against breaking into pro-

[55] In SPEEDOS a process is defined to consist of one or more threads, each of which has a thread stack.

cesses.

Let us suppose that an initial process is created for a user when he is first introduced into a system. This can then continue to be used by him as needed throughout the entire time he is authorised to use the system. A variant of this idea of persistent processes was already implemented successfully in the MONADS systems [23, 24]. We now consider the interesting effect which it can have on logging in and logging out of processes.

3.1 Logging in and Logging out

When a conventional operating system detects new activity at a terminal it establishes the authenticity of a user (usually by requesting him to provide a username and password) and then creates a process for the following terminal session. Using implicit command files (such as `.login` and `.cshrc` in Unix for example) and explicit commands (e.g. `cd` in Unix) the user then establishes the working environment which he needs for his process and invokes a further command or commands to begin the activity which he wants to carry out.

The creation of a process and then tailoring it to the required environment can involve considerable processing time and file activity, which usually manifests itself to the user as a delay before he can begin his real work.

Suppose, however, that the user has a persistent process in the persistent virtual memory, waiting to be activated. The considerable activity involved in process creation can be saved (because the process already exists) and – depending on the state of the inactive process – the time spent in tailoring it to the required environment can also be saved in part or in full. To understand this we first consider the state of an inactive process before logging in takes place, and how it got into that state. In other words we first need to consider what happens to a process when a `logout` command is issued in the previous session.

3.2 Executing Commands

Before looking at the `logout` command we need to understand how a normal command is executed. When the user issues a basic command (or its equivalent via a graphical interface) he will normally be communicating with a command language interpreter (CLI), or in modern systems a graphical form thereof, so that the module frame at the top of the stack will be that of the CLI. In order to read a command, the CLI will call a device driver to read the command from the user's terminal or equivalent device. During the reading of the command the device driver module will have a stack frame above that of the CLI, but this will be deleted when the command has been passed back to the CLI. The stages through which the stack progresses during this command-reading process are illustrated in Figure 15.3.

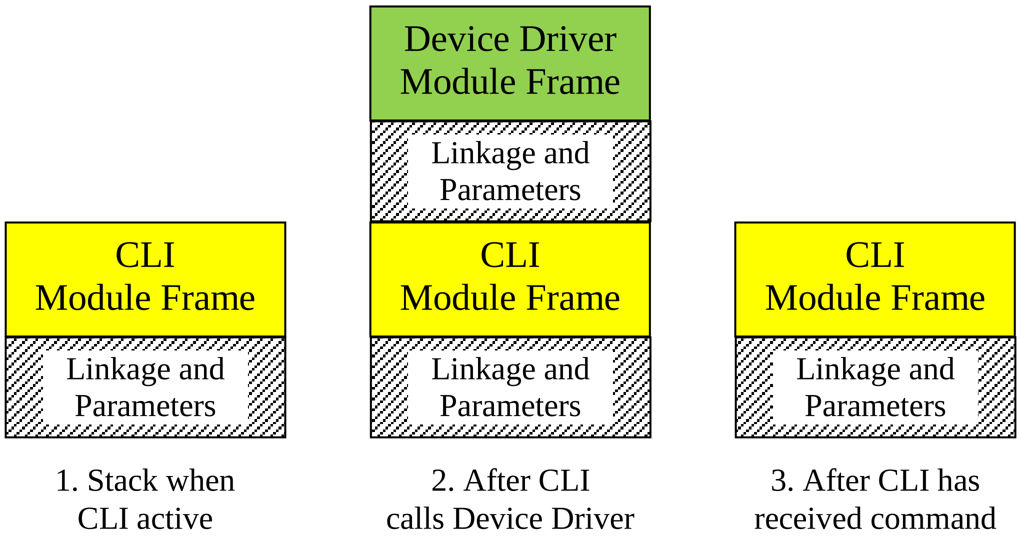


Figure 15.3: CLI Invoking a Device Driver on a Persistent Stack

If the command just read were a command to `edit` a file the CLI would then invoke the edit module on the stack, which would in turn invoke the file module for the file to be edited, as shown in Figure 15.4.

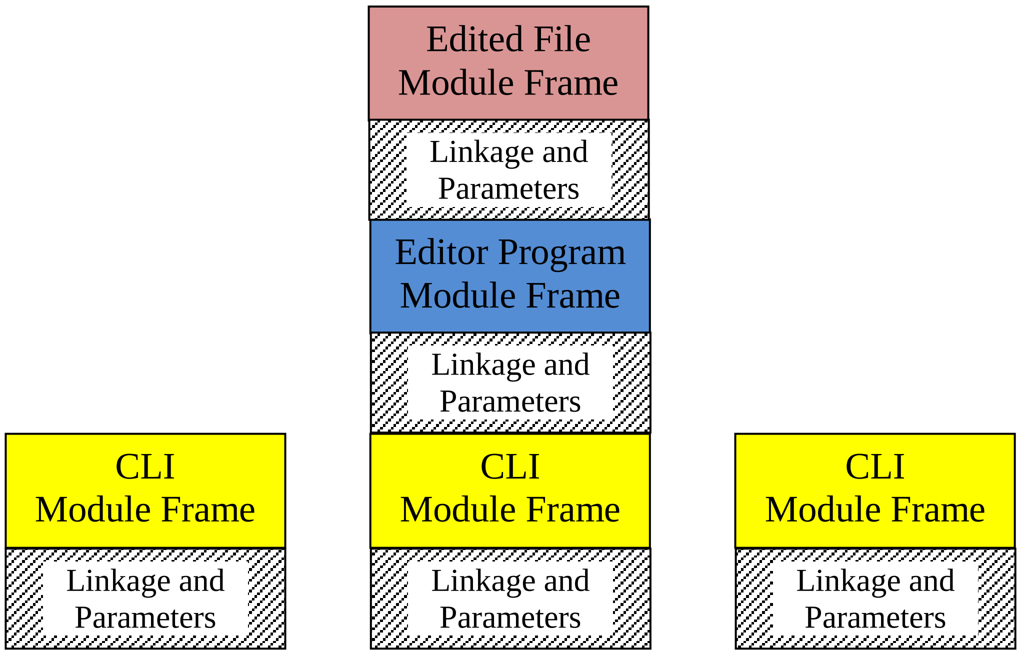


Figure 15.4: CLI Invoking an Editor on a Persistent Stack

3.3 The Logout Command

Having seen in principle how the stack is used by the CLI to read commands and to execute them, we can consider how it implements a `logout` command. From the viewpoint of the CLI this can be implemented in a persistent system (in contrast with a conventional system) like any other command, so the module frame for the logout module is invoked above the CLI module frame on the stack, just as the `edit` command was.

What now happens depends on the code of the logout command. After it has done some housekeeping activities (such as releasing the terminal) this does *not* return to the CLI like other commands, but instead it calls a special operation provided by the kernel, which we call its "long suspend" function, as is shown in Figure 15.5.

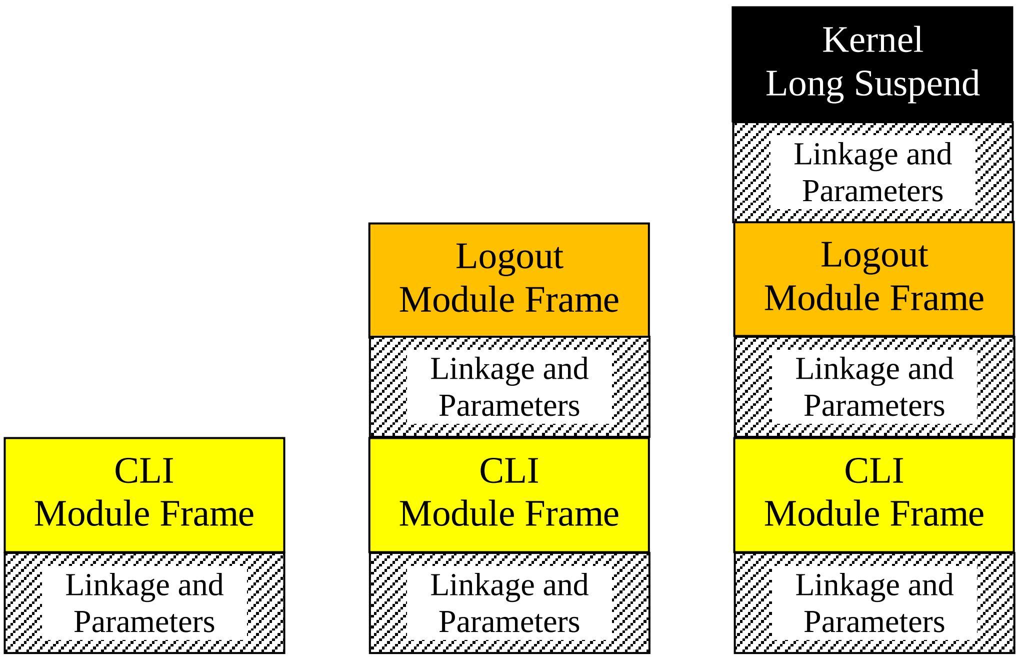


Figure 15.5: Logging Out a Persistent Stack

The long suspend function of the kernel advises the process scheduler that the process is to be deactivated, and in the normal course of virtual memory management the stack's page frames in the main memory will be copied back to disc and released for other use. This is the state of a logged out process.

When at some later time the kernel detects that the user wishes to log in again, it advises the process scheduler that the process can now be scheduled in the usual way. In a timesharing or transaction processing system it may be necessary to change the command input device to that at which the user now logs in. The kernel then exits from the long suspend routine, leaving the user process free to execute the next instruction in the logout module. When this exits back to the CLI (which is unaware that the process has been logged out) the latter requests the next command in the usual way.

This scheme is efficient in that it saves the CPU processing time and disc accesses involved in creating and deleting a new process for each terminal session and in setting up the process to suit the user's particular requirements.

It is possible to take this idea a step further. The logout module is itself a normal module so that it can be called from any other module which has the appropriate permissions, not just from the CLI. This means, for example, that if an editor provides its users with a facility to log out in the middle of an edit session, then the invocation of the logout module can take place without returning to the CLI. This is illustrated in Figure 15.6. It means that even less processing time is required to re-establish the user's work, and it provides a very convenient working environment for the user himself.

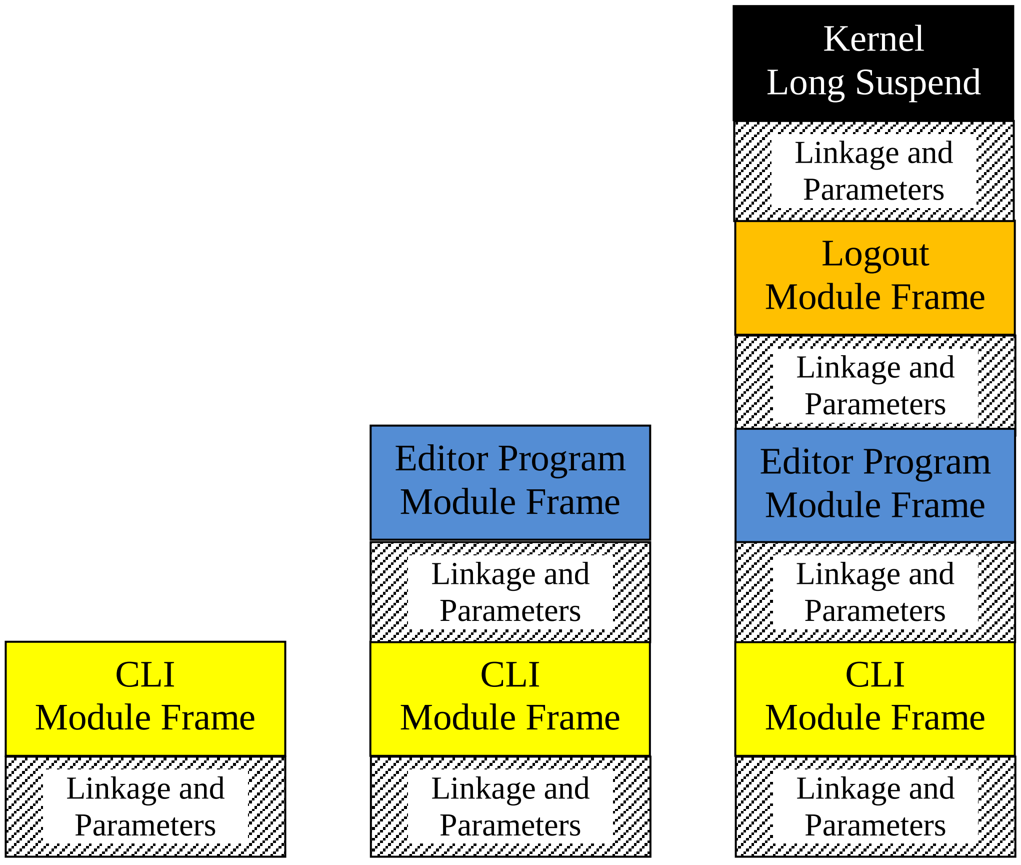


Figure 15.6: Logging Out a Persistent Stack from an Editor

3.4 Identification and Authentication

So far we have completely ignored the important questions of the identification and authentication of users when they log into a system. It would of course be possible to build into the kernel's long suspend routine some conventional tests such as the checking of a password. However, the new framework provided by persistent processes offers a much more powerful and at the same more flexible possibility for guaranteeing the security of access to processes.

The basic idea is to separate identification from authentication, leaving the kernel to carry out the relatively simple task of identification and giving the user the opportunity to define his own authentication protocol. This is very simple to achieve in the SPEEDOS environment. When a user wishes to log in, the kernel must in any case determine which persistent process is to be activated. This corresponds to the initial identification of a user by a username in a conventional system. Thus the kernel needs to have a mapping between user names and persistent process names or numbers[56]. When a user identifies himself to the system the kernel activates the corresponding persistent process, as described in the previous section. Thus the long suspend routine exits back to the user logout module (in the logout routine) *without any authentication checks having been carried out*. This may appear to be a foolhardy approach. But as we shall now see, it can be used to good advantage to improve security, as follows.

When the logout routine is reactivated this is the point at which to authenticate the user who is attempting to log in. How this is achieved is not dictated by the system, as the logout routine is a user module like any other[57]. This means in principle that each user can himself determine what authentication tests are to be carried out on persons attempting to log into his process.

In Chapter 4 it was argued that leaving the authentication of users to the operating system is a root cause of weak security, because it gives the hacker several advantages. First, he knows what he has to do to penetrate the system. Second the central repository of authentication information (e.g. the password file) implied by a centralized authentication system provides him with an ideal target. Both these advantages for the hacker are removed if each user can carry out his own authentication in whatever way he sees fit. With arbitrary user authentication procedures in operation the hacker doesn't know whether he has to crack a simple password, a dynamic password, a cognitive password and/or whether he has to conform to some required actions. The possibilities are endless.

In keeping with the principles of modularity, it is unwise to pack the actual authentication procedures into the logout module, which has important system housekeeping activities to carry out, such as the de-allocation and re-establishment of the user terminal as the command source. Instead the logout routine simply needs the possibility to invoke a separately programmed user authentication module, which carries out the tests and advises of the result. In this way each user can link a different authentication module to the logout module.

This would lead to the authentication module being invoked above the logout module on the process stack (see Figure 15.7). It could be organised that regardless how the authentication module carries out its work, it returns a simple binary message to the logout module, where the value *true* indicates that the au-

[56] If a user has multiple persistent processes (e.g. for different projects or activities on which he is working), then he must identify the process to be activated.

[57] This does not exclude the possibility that standard logout modules are made available by the operating system or can be bought from component suppliers.

thentication is successful, while *false* indicates that it failed.

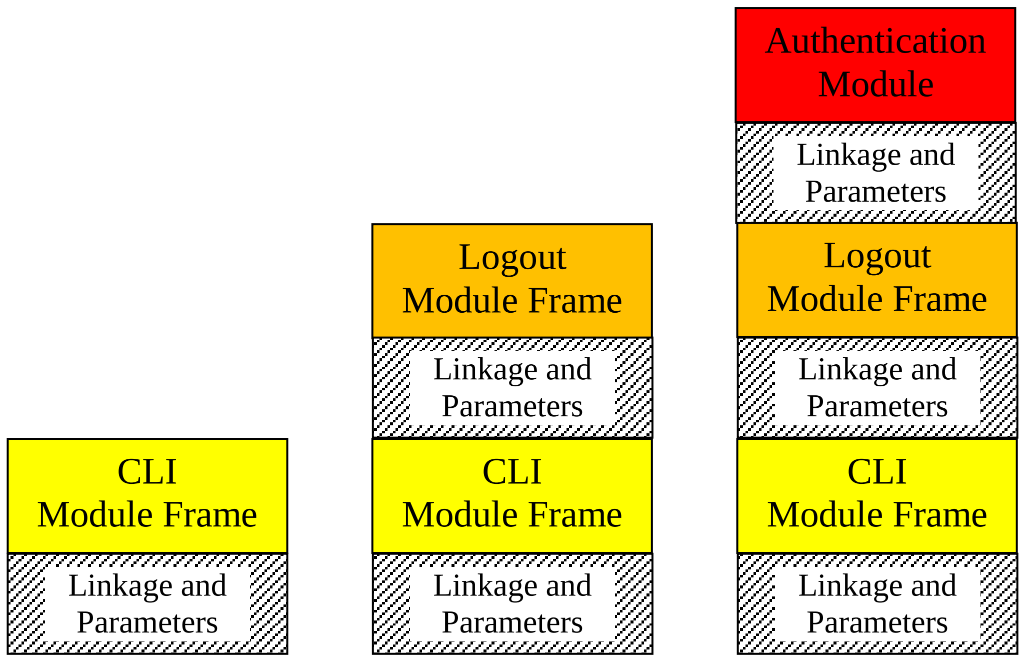


Figure 15.7: Authenticating a User at Login

The very simplest module would be one which always returns the value *true*. In other words no authentication whatsoever takes place. This might be useful for cases where a public service is provided by the process (e.g. a process for interrogating a library catalogue). More complex modules might use any of the authentication techniques discussed in Chapter 4 (or any combination thereof) or any other way of authenticating users which might in future be devised.

You might now ask what happens when a user forgets how to authenticate himself. In conventional systems this is usually where a superuser has to be called in, but that is not necessary if users themselves prepare in advance for this situation. A user might for example entrust a friend with a module capability which provides access to an entry point of the authentication module that resets the protection (e.g. by changing a password or by linking it to another checking routine). And if he doesn't completely trust a friend he might involve two friends, each of whom has a module capability allowing half a password to be changed (just as in bank vaults two keys are often necessary to unlock the vault). Alternatively he might have his own further persistent process(es) with a different authentication module which can make the necessary changes or allows him to be reminded of a password (for example). The possibilities are endless, and a privileged superuser is certainly not needed for this purpose.

4 Implementing a Process as Threads

When a process is created it is assigned a new persistent container. The process can thereafter be identified by the unique number of this container. The process itself does not have a separate stack. It can be viewed as a persistent "file" in which one or more threads can be created. Each thread has its own stack, which is identified by its unique thread number, consisting of the unique container number and its *thread index*, a small integer starting with 0 for the first thread created, 1 for the second and so on.

The primary purpose of a thread stack is to provide the linkage and parameters needed for inter-module routine calls. In contrast with the stack structure discussed in Chapter 8 it does not hold the local variables of the routines which are invoked from it. The reasons for this are that

a) different high level languages have different scope roles which can affect the stack structure and the organisation and addressing of their on-stack data, and

b) as the advocates of the RISC philosophy argued, it is more efficient for compilers to have the freedom to organise their stacks internally, rather than imposing on them an architecturally enforced structure.

Hence the main purpose of a thread stack is to provide a framework allowing

– the interface routines of a module to be invoked, and parameters passed to it; and conversely

– the transition back to the calling routine from an interface routine of a returning module, together with the return parameters.

Since all the thread stacks of a process are organised in this way, each thread can invoke modules (whether operating system modules, file modules or other applications) independently of each other. However, they will often cooperate with each other within the same module.

While a thread is active within a module it will typically have a "continuation" of its stack in the data container of the module, which has a structure determined by the compiler generated run-time code. Each thread active in the module has its own root segment, which is separate from the root segment for the persistent data of the module, but nevertheless in the same container. When the kernel executes an inter-module call, part of its work is to create a root segment for the thread. This persists so long as the thread is active in the module (even after it has called further modules), and it is deleted by the kernel when the thread makes an inter-module return from the module.

As part of the inter-module call the kernel sets up a segment register which allows the module's code to address the persistent data while the thread is active and a further register to address thread-related temporary data. Both kinds of data are held in a single heap. While executing in the module the thread can create new segments in the heap and can link these into its own temporary data and/or the persistent data. Those segments which are only temporary are deleted when the thread returns back to its calling module, but any segments which it has linked into the persistent data structure continue to exist independently of the thread.

In principle any thread can log out without causing other threads of the same process to be suspended. However, sometimes threads will synchronise their activities such that they are all deactivated together by a single thread when it logs out. This need not be the thread with index 0.

5 Multiple Processes

As hinted above, a user is not limited to having a single process module. Hence he can carry out independent activities in parallel (e.g. writing a letter or document, processing email, using a spreadsheet and carrying out banking transactions). By extending the model to allow a user to have any number of persistent processes (each potentially with several threads) he can dedicate each to a separate use, leaving each in a different unfinished state when he logs them out.

6 Conclusion

A framework for structuring software in a secure system has now been developed. It involves defining all modules as information hiding modules and implementing all processes as persistent processes in the persistent virtual memory. In this way higher level protection can be based on the right to invoke the semantic operations of modules (with the memory architecture guaranteeing that a process executing within a module is confined to accessing only the segments of that module).

It has also been shown how persistent stacks can be used to detach authentication checking mechanisms from the operating system, providing the user with more flexibility, the hacker with less knowledge and the system with more overall security against hackers.

Chapter 16
Architectural Implications of the Software Model

Chapters 14 and 15 outline the basic software model used to structure the SPEEDOS system. This model does not go into detail, but in the second volume a more detailed picture of the kernel and of basic operating system modules will emerge and a few extensions will be made, for example to introduce n-ary routines[58], to integrate qualifiers (see chapter 13) into the model, to describe synchronisation and to support internal subroutine libraries. To have described such features here would have led to a good deal of detail which is best left to volume 2. However, we have now provided sufficient background information to allow us to complete the architectural picture which was started in earlier chapters.

1 Containers in SPEEDOS

Containers were introduced at the end of chapter 11 as a paged unit of persistent virtual memory into which segments can be placed[59]. Here we assume that modules and processes are held in segmented and paged containers. Since a container must potentially be capable of holding very large data structures for file modules, we provisionally define its maximum length as 2^{64} bytes. In reality most containers will be very much smaller than this, so that the paging mechanism must be capable of handling both small and large segments. How containers are structured internally will be described in Chapter 23.

2 Worldwide Unique Addresses

We recall from Chapter 14 that module capabilities hold a unique module identifier (see Figure 16.1, which repeats Figure 14.7).

58 N-ary routines operate on two or more file modules at the same time, e.g. to convert the data from one format to another or to compare two files.

59 In the literature on the MONADS systems, which first implemented the orthogonal model, containers were called *address spaces*.

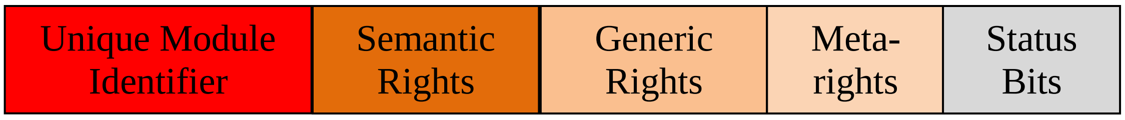


Figure 16.1: The Basic Structure of a Module Capability

Mindful both of Bell and Strecker's comment:

"There is only one mistake that can be made in a computer design that is difficult to recover from – not providing enough address bits." [104].

and also of the fact that the identifier should be large enough to identify uniquely every future SPEEDOS module in the world, we now consider how large the unique module identifier should be, and how it can be structured.

Since it would be totally infeasible to have a single central registry for all modules, a module must be locatable from its identifier. The module identifier therefore includes the actual number of the container in which the module resides (e.g. in the case of a file module, the container holding the file data.)

As they appear in capabilities, container identifiers are very large numbers which must be unique across all SPEEDOS nodes in the Internet. Because such numbers need to be allocated at different individual SPEEDOS nodes, they can be structured rather like telephone numbers (see Chapter 2), thus enabling each node locally to allocate and manage its own range of numbers. Thus as a first approximation a container number can be defined as a pair «unique node #, container # in node».

Similarly node numbers can be kept unique if each company which manufacture SPEEDOS systems has a unique "manufacturer number" which is prefixed to a unique "node number within manufacturer" for each new node.

Containers are not simply associated with nodes, but reside on particular discs at a node. Hence in order to help locate a container it is helpful for the "container # in node" part of a container identifier to be decomposed further into a disc number part and a container number within disc. This leads to the structure for a container identifier illustrated in Figure 16.2.

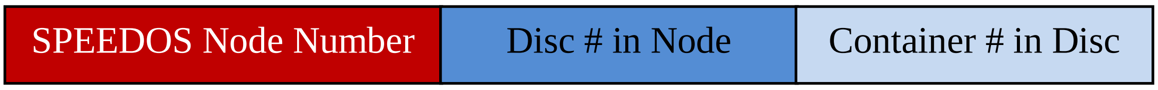


Figure 16.2: A SPEEDOS Container Identifier

Allocating numbers for discs locally at a node can be organised simply by adding one to the last disc number allocated. Here the oversimplifying assumption is made that a disc has a fixed association with a particular node and that it is never used on different nodes. While this assumption often corresponds to practice, it is not always true; we consider this issue in volume 2, chapter 27[60].

Allocating numbers for the containers on a disc can be undertaken by incrementing the last container number used on the relevant disc, in this case by the *Disc Directory Manager* for the disc in question.

It is important to avoid the ambiguities which can arise in a telephone system, where numbers are reallocated as users no longer need their telephone, etc. SPEEDOS avoids this problem by making all three parts of the identifier sufficiently large that it will not be necessary to re-use them. We here assume that each part is 64 bits long. In volume 2, chapter 23 the actual details are discussed.

3 Translating Virtual Addresses

Logically a SPEEDOS container identifier can be considered to be the first part of a worldwide unique SPEEDOS virtual address, which consists of the pair «container identifier, offset in container». From the viewpoint of page management in the main memory the "offset in container" part of a virtual address itself decomposes into the pair «page # in container, offset in page», see Figure 16.3.

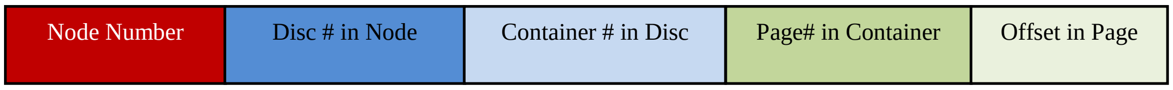


Figure 16.3: A SPEEDOS Full Virtual Address

Assuming that a virtual address is as described above then a virtual page number has the structure shown in Figure 16.4. In principle the task of an address translation unit (ATU) for SPEEDOS is to map very large virtual page numbers onto main memory page frames.

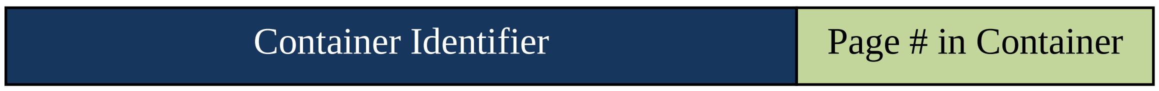


Figure 16.4: A SPEEDOS Virtual Page Number

In the MONADS systems the ATU actually achieved the equivalent of this. Each virtual address in the MONADS local area network was unique, and David Abramson designed an ATU, 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) [95].

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 (in-

60 In chapter 6 of his thesis [130] Frans Henskens addresses this issue in detail from the perspective of a future MONADS system.

cluding two bits to indicate on which of the four nodes in the network the container resides), whereas in SPEEDOS we are discussing very much larger unique virtual addresses.

These parameters create two sets of problems for a SPEEDOS implementation based on the MONADS ATU technique. First, the increased size of main memories means that the number of entries in an ATU 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 an ATU would also be significantly greater.

The first problem alone makes a MONADS style implementation infeasible, but the second problem creates substantially greater problems. Hence a different approach is adopted 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.

3.1 Managing the Number of Entries in the Main Memory Page Table

At the time the RISC idea was becoming popular (in the early 1980s) 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 16.5, indicates the task of the software in this RISC scenario.

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[61], in SPEEDOS terminology the Main Memory Page Table, MMPT) must also be maintained in software.

[61] 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 the same functionality as the MONADS ATU mentioned above.

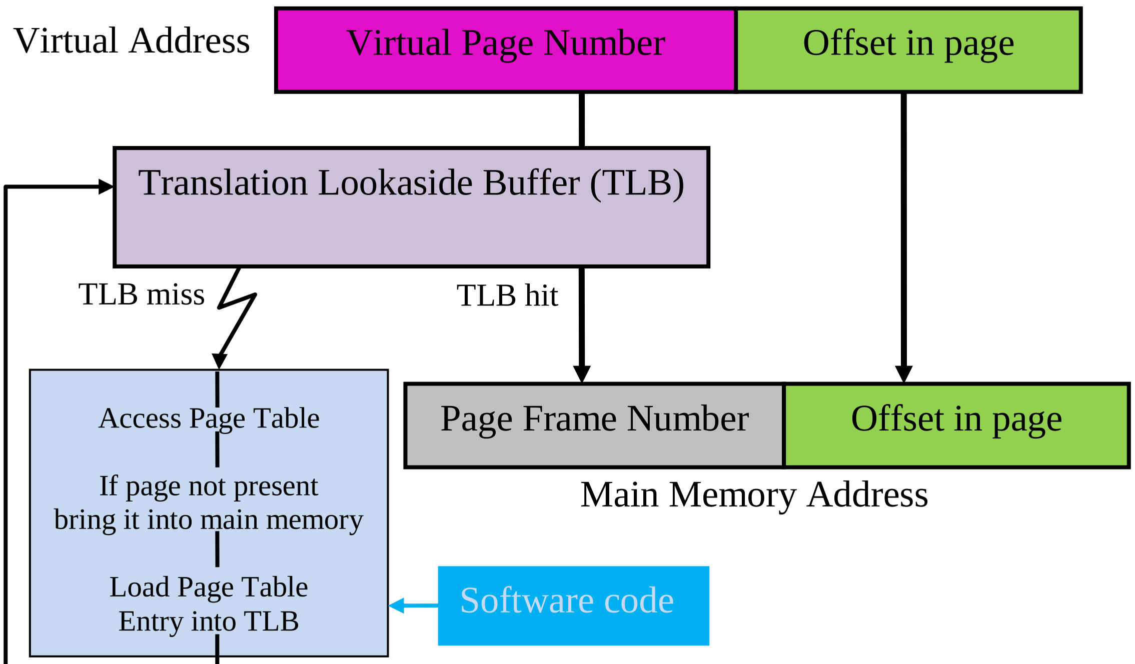


Figure 16.5: The TLB as the entire ATU

When a TLB miss occurs the hardware interrupts into the core kernel code. This first examines the inverted page table 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 inverted page table and loads the appropriate information into the TLB, allowing the process/thread to continue execution without being suspended.

If on the other hand the TLB miss arises because a genuine page fault has occurred, the kernel must undertake steps to 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. The details will be clarified in more detail in volume 2 chapter 23, in the more detailed discussion of virtual memory organisation.

3.2 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 internet-wide 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 must be found.

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 process 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/process switch.

This is not the place to provide a definitive solution for this problem, since an actual solution must depend on what actual TLB hardware is available. For illustration purposes we now describe the more difficult case: how SPEEDOS can effectively use a TLB which supports only a single address space.

3.3 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, then this raises a special problem for SPEEDOS, because a SPEEDOS container is never active alone. However, thanks to the rigid enforcement of the information-hiding principle, normally there are three active containers at any point in time: a process/thread 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.
– 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, a kernel designer could use the three top bits of a virtual address to act as a *short container identifier* (SCID). Figure 16.6 shows how eight containers can be addressed simultaneously in what the TLB views as a single address space.

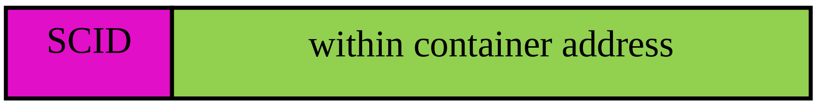


Figure 16.6 Prefixing an Address with a Short Container Identifier

The actual mapping of the 3 bits might by kernel convention be defined as is shown in Figure 16.7. Of course the state of a thread must include not only this mapping but also a set of pseudo-registers (which we call Container Registers) that contain the full meanings of the SCIDs. These must be saved and restored by the kernel on context switches.

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.

Figure 16.7 A Possible Allocation of Short Container Identifiers

This approach might appear to be rather similar to the Multics mapping of architectural segments onto an address space. But there are some very significant differences.

(a) In contrast with Multics, which used 18 bits of a virtual address as a segment number, only three bits are needed in SPEEDOS for the equivalent mapping, thanks to the orderly (information hiding) use of containers to implement SPEEDOS modules. This leaves far more bits for use as within container addresses, which in any case are likely to be significantly larger as a result of 64 bit addressing.

(b) Whereas the attempt to map files onto architectural segments led to severe complications in the management of addressing in Multics, 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, thread switches and in association with the loading of segment registers.

(c) On an inter-module call and also on a thread switch between two threads of the same process, the stack address space does not need to be flushed.

(d) It would be a straightforward matter to improve efficiency by implementing separate TLBs for stack, data and code addressing.

In contrast with MONADS, it is necessary on each process switch and on each inter-module call to flush the TLB and caches, just as occurs in conventional systems when process switches and operating system calls occur[62].

4 Segment Management

Since S-RISC addressing is based on the use of protected segment registers (see Chapter 11), the SPEEDOS kernel is free to organise segments into three partitions: for data, pointers and module capabilities (cf. Figure 16.8). The pointer and module capability partitions are protected by the kernel in that the latter never makes these partitions addressable to normal users, i.e. it never loads a segment register to allow such access, but only loads segment registers to pro-

[62] On an inter-module call/return, TLB and cache entries for the stack address space need not be flushed. It would also be possible for hardware designers to optimise the address translation process (e.g. by using different TLBs and caches not only for code and data but also for stacks). However such optimisations are not further considered here, as it is not our intention at this point to provide a detailed hardware design for SPEEDOS.

vide processes with access to data partitions. It also checks that the pointer partition is empty for parameter segments on inter-module calls and returns[63].

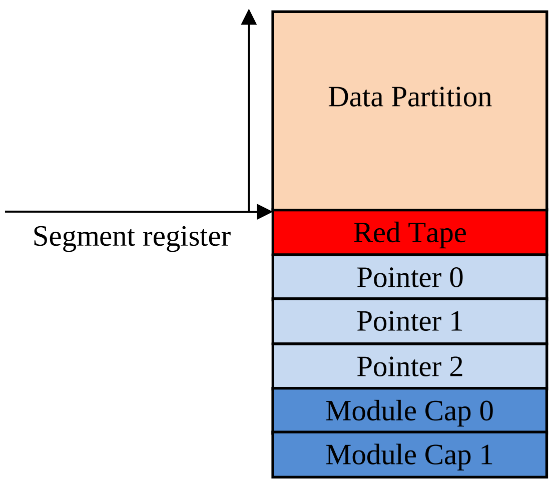


Figure 16.8: SPEEDOS Partitioned Segments with Module Capabilities

There are separate kernel calls for accessing pointers and for accessing module capabilities. In each case the user must provide

– the number of a segment register which already addresses the segment in question; and a

– a non-negative integer which selects the pointer or module capability.

The kernel calls provide a range of protected functions such as following pointers, making inter-module calls and returns, reducing the access rights in module capabilities, etc.

5 Conclusion

In this chapter we have built on the descriptions of the basic hardware features described in Chapters 11 and 12, presenting an overview of the hardware and complementary software needed to support a SPEEDOS system and have shown that efficient implementations of the hardware are possible.

[63] This helps to enforce the information-hiding principle and at the same time avoids a worldwide garbage collection problem!

References

[1] J. L. Keedy, “S-RISC: Adding Security to RISC Computers,” SPEEDOS Website (https://www.speedos-security.org/), 2023.

[2] C. Stoll, “Stalking the Wiley Hacker,” *Communications of the ACM,* vol. 31, no. 5, pp. 484-497, 1988.

[3] U.S. Department of Defense, “Trusted Computer System Evaluation Criteria,” 1985.

[4] S. J. Gould, The Panda's Thumb, Norton, 1992.

[5] S. J. Gould, Bully for Brontosaurus, Norton, 1992.

[6] J. Cohen and I. Stewart, The Collapse of Chaos, Penguin, 1994.

[7] J. Dewey, Experience and Education, Collier Books, 1938.

[8] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5 ed., Elsevier, 2012.

[9] B. W. Lampson, “Protection,” *ACM Operating Systems Review,* vol. 8, no. 1, pp. 18-24, January 1974.

[10] D. E. Bell and L. J. LaPadula, “Secure Computer Systems: Mathematical Foundations,” Mitre Corp, Bedford, Massachusetts, 1973.

[11] 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.

[12] M. Evered and J. L. Keedy, “A Model for Protection in Persistent Object-Oriented Systems,” in *International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, Germany, 1990.

[13] B. Thuraisingham, “Mandatory Security in Object-Oriented Database Systems in 'Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications',” *ACM Sigplan Notices,* 1989.

[14] D. Denning, “A Lattice Model for Secure Information Flow,” *Communications of the ACM,* vol. 19, no. 5, pp. 236-243, 1976.

[15] K. Biba, “Integrity Considerations for Secure Computer Systems,” USAF Electronic System Division, 1977.

[16] D. Clark and D. Wilson, “A Comparison of Commercial and Military Computer Security Policies,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[17] J. Lanier, Ten Arguments for Deleting your Social Media Accounts right now, N.Y.: Henry Holt and Company, 2018.

[18] National Bureau of Standards, “Data Encryption Standard,” Washington, D.C., 1977.

[19] R. Rivest, A. Shamir and A. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of the ACM,* vol. 21, no. 2, pp. 120-126, 1978.

[20] T. Mark, A. Lomas, L. Gong, J. H. Saltzer and R. M. Needham, “Reducing Risks from Poorly Chosen Keys,” *ACM Operating Systems Review,* pp. 14-18, 1989.

[21] R. Morris and K. Thompson, “Password Security - A Case History,” *Communications of the ACM,* vol. 22, no. 11, pp. 594-597, 1979.

[22] B. L. Riddle, M. S. Miron and J. A. Semo, “Passwords in Use in a University Timesharing Environment,” *Computers and Security,* vol. 8, no. 7, pp. 569-579, 1989.

[23] J. L. Keedy, “A Model for Security and Protection in Persistent Systems,” *Microprocessors and Microsystems,* vol. 17, no. 3, pp. 139-146, 1993.

[24] J. L. Keedy and K. Vosseberg, “Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System,” in *Proceedings of the 25th Hawaii International Conference on System Sciences*, 1992.

[25] S. R. Ames, M. Gasser and R. R. Shell, “Security Kernel Design and Implementation: An Introduction,” *IEEE Computer,* vol. 16, no. 7, pp. 14-25, 1983.

[26] L. J. Fraim, “Scomp: A Solution to the Multilevel Security Problem,” *IEEE Computer,* vol. 16, no. 7, pp. 26-46, 1983.

[27] J. K. Millen, “Security Kernel Validation in Practice,” *Communications of the ACM,* vol. 19, no. 5, pp. 243-250, 1976.

[28] G. J. Popek, M. Kampe, C. Cline, A. Stroughton, M. Urban and E. J. Walton, “UCLA Secure Unix,” in *Proceedings of the AFIPS National Computer Conference,* 1979.

[29] L. W. Schiller, “The Design and Specification of a Security Kernel for the PDP 11/45,” Report Number ESD-TR-75-69, Bedford, Mass., 1975.

[30] M. D. Schroeder, D. D. Clark and J. H. Saltzer, “The MULTICS Kernel Design Project,” in *Proceedings of the Sixth Symposium on Operating System Principles*, 1977.

[31] K. G. Walter, S. I. Schaen, W. F. Ogden, W. C. Rounds, D. G. Shumway, D. D. Schaeffer and F. T. Bradshaw, “Structured Specification of a Security Kernel,” in *Proceedings of the International Conference on Reliable Software*, 1975.

[32] 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.

[33] B. W. Lampson, “Computer security in the real world,” *IEEE Computer,* vol. 37, no. 6, pp. 37-46, 2004.

[34] A. W. Burks, H. H. Goldstine and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Report to the U.S. Army Ordnance Department, 1946.

[35] M. V. Wilkes, "Slave Memories and Dynamic Storage Allocation," *IEEE Transactions on Electronic Computers*, vol. 14, no. 2, pp. 270-271, 1965.

[36] T. Kilburn, D. Edwards, M. Lanigan and F. Sumner, "One Level Storage System," *I.R.E. Transactions on Electronic Computers,* vol. 11, no. 2, pp. 223-235, 1962.

[37] W. Lonergan and P. King, "Design of the B5000 System," *Datamation,* vol. 7, no. 5, pp. 28-32, 1961.

[38] P. J. Denning, "The Working Set Model for Program Behaviour," *Communications of the ACM,,* vol. 11, no. 5, pp. 323-333, 1968.

[39] P. J. Denning, "Virtual Memory," *ACM Computing Surveys,* vol. 2, no. 3, pp. 153-189, 1970.

[40] D. B. G. Edwards, A. E. Knowles and J. V. Woods, "MU6-G: A New Design to Achieve Mainframe Performance from a Mini Sized Computer," in *Proceedings of the 7th Annual Symposium on Computer Architecture*, 1980.

[41] E. I. Organick, Computer Systems Organization, the B5700/6700 Series, N.Y.: Academic Press, 1973.

[42] E. I. Organick, The Multics System: An Examination of its Structure, Cambridge, Mass.: MIT Press, 1972.

[43] A. P. Batson, S. Ju and D. Wood, "Measurements of Segment Sizes," *Communications of the ACM,* vol. 13, no. 3, pp. 155-159, 1970.

[44] A. P. Batson and R. E. Brundage, "Segment Sizes and Lifetimes in Algol 60 Programs," *Communications of the ACM,* vol. 20, no. 1, pp. 36-44, 1977.

[45] J. L. Keedy, "An Outline of the ICL2900 Series System Architecture," *Australian Computer Journal,* vol. 9, no. 2, pp. 53-62, 1977.

[46] J. L. Keedy, "An Outline of the ICL2900 Series System Architecture," in *Computer Structures: Principles and Examples (ed. Siewiorek D.P., Bell, C.G. and Newell, A.),* N.Y., McGraw-Hill, 1982, pp. 251-259.

[47] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *ACM Operating Systems Review,* vol. 13, no. 2, pp. 3-19, 1979.

[48] K. Ramamohanarao, "A New Model for Job Management Systems," *PhD. Thesis, Monash University, Australia,* 1980.

[49] E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages, ed. E. Genuys,* Academic Press, 1968, pp. 43-112.

[50] P. J. Courtois, F. Heymans and D. L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM,* vol. 14, no. 10, pp. 667-668, 1971.

[51] 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.

[52] G. A. Blaauw and F. P. Brooks, "The Structure of the System/360: Part I - Outline of the Logical Structure," *IBM System Journal,* vol. 3, no. 2, pp. 119-135, 1964.

[53] G. A. Blaauw and F. P. Brooks, "The Structure of the System/360: Part I - Outline of the Logical Structure," in *Computer Structures: Principles and Examples (ed. Siewiorek, D.P., Bell, C.G. and Newall, A.),* New York, McGraw-Hill, 1982, pp. 695-710.

[54] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," *Communications of the ACM,* vol. 11, no. 5, pp. 341-346, 1968.

[55] D. L. Parnas, "On a Buzzword: Hierarchical Structure," in *Information Processing 74, IFIP Congress 74*, 1974.

[56] R. S. Fabry, "Capability Based Addressing," *Communications of the ACM,* vol. 17, no. 7, pp. 403-412, 1974.

[57] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM,* vol. 9, no. 3, pp. 143-155, 1966.

[58] H. M. Levy, Capability-Based Computer Systems, Bedford, Mass.: Digital Press, 1984, p. 220.

[59] 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.

[60] W. A. Wulf, R. Levin and S. P. Harbison, HYDRA/C.mmp: An Experimental Computer System, New York: McGraw-Hill, 1981.

[61] B. W. Lampson and H. Sturgis, "Reflections on an Operating System Design," *Communications of the ACM,* vol. 19, no. 5, pp. 251-265, 1976.

[62] D. M. England, „Architectural Features of System 250," in *Infotech State of the Art Report/Operating Systems Vol. 14*, 1972, pp. 395-426.

[63] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its Protection System," in *Proceedings of the 6th ACM Symposium on Operating System Principles*, 1977.

[64] M. V. Wilkes und R. M. Needham, The Cambridge CAP Computer and its Operating System, Oxford: North Holland, 1979.

[65] Intel Corporation, Introduction to the iAPX432 Architecture, 1981.

[66] A. Valenzano, Advanced Microprocessor Architectures, Addison-Wesley, 1987.

[67] U.S. Department of Defense (ed.), Programming Language ADA: Reference Manual, vol. Lecture Notes in Computer Science vol. 106, Springer-Verlag, 1981.

[68] J. L. Hennessey, "VLSI Processor Architecture," *IEEE Transactions on*

Computers, Vols. C-33, no. 12, pp. 1221-1246, 1984.

[69] D. A. Patterson, “Reduced Instruction Set Computers",” *Communications of the ACM,* vol. 28, no. 1, pp. 8-21, 1985.

[70] G. Radin, “The 801 Minicomputer,” in *Proceedings of the First InternationalSymposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, 1982.

[71] J. E. Thornton, “Parallel Operation in the Control Data 6600,” in *Proceedings of the Fall Joint Computer Conference,* 1964.

[72] J. E. Thornton, Design of a Computer: The Control Data 6600, Glenview, Illinois: Scott Foresman & Co., 1970.

[73] T. A. Laliotis, “Architecture of the SYMBOL Computer,” in *High-Level Language Computer Archecture (ed. Y. Chu),* New York, Academic Press, 1975, pp. 110-185.

[74] W. R. Smith, R. Rice, G. Chesley, T. A. Laliotis, S. F. Lundstrom, M. A. Calhoun, L. D. Gerould and T. G. Cook, “SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software,” in *Proceedings of the Spring Joint Computer Conference,* 1971.

[75] D. R. Ditzel and D. A. Patterson, “Retrospective on High-Level Language Computer Architecture,” in *Proceedings of the 7th Annual Symposium on Computer Architecture, ACM Computer Architecture News,* La Baule, France, 1980.

[76] M. Anderson, R. D. Pose and C. S. Wallace, “A Password Capability System,” *The Computer Journal,* vol. 9, no. 1, pp. 1-8, 1986.

[77] V. Berstis, “Security and Protection of Data in the IBM System/38,” *ACM Computer Architecture News,* vol. 8, no. 3, pp. 245-252, 1980.

[78] M. E. Houdek, F. G. Soltis and R. L. Hoffman, “IBM System Support for Capability Based Addressing,” in *Proceedings of the 8th SIGARCH Symposium on Computer Architecture,* 1981.

[79] R. N. M. Watson, S. W. Moore, P. Sewell and P. G. Neumann, “An Introduction to CHERI,” University of Cambridge, September 2019.

[80] E. Wichel, J. Cates and K. Asanovic, “Mondrian memory protection,” *Communications of the ACM,* vol. 37, no. 10, 2002.

[81] E. Witchel, J. Rhee and K. Asanovic, “Mondrix: Memory isolation for Linux using Mondrian memory protection,” in *Proceedings of the 20th ACM Symposium on Operating System Principles,* October 2005.

[82] J. Devietti, C. Blundell, M. M. K. Martin and S. Zdancewic, “Hardbound: architectural support for spatial safety of the C programming language,” *SIGARCH Comput. Archit. News,* vol. 36, no. 1, pp. 103-114, March 2008.

[83] N. P. Carter, S. W. Keckler and W. J. Dally, “Hardware Support for Fast Capability-based Addressing,” *SIGPLAN Notices, no. 11,* vol. 29, no. 11, pp. 319-327.

[84] A. K. Jones, “Capability Architecture Revisited,” *ACM Operating Systems Review*, vol. 14, no. 3, pp. 33-35, 1980.

[85] J. L. Keedy, “An Implementation of Capabilities without a Central Mapping Table,” in *Proceedings of the 17th Hawaii International Conference on System Sciences*, 1984.

[86] B. Randell, “A Note on Storage Fragmentation and Program Segmentation,” *Communications of the ACM*, vol. 12, no. 7, pp. 365-369, 1969.

[87] J. L. Keedy, Timor: An Object- and Component Oriented Language, 2020.

[88] J. L. Keedy, “Paging and Small Segments: A Memory Management Model,” in *Proceedings of the 8th World Computer Congress*, Melbourne, Australia, 1980.

[89] R. M. Russell, “The CRAY-1 Computer System,” *Communications of the ACM*, vol. 1, no. 21, pp. 63-72, 1978.

[90] R. M. Russell, “The CRAY-1 Computer System,” in *Computer Structures: Principles and Examples (ed. Siewiorek D.P., Bell, C.G. and Newell, A.)*, N.Y., McGraw-Hill, 1982, pp. 743-752.

[91] J. Rosenberg and D. A. Abramson, “MONADS-PC: A Capability Based Workstation to Support Software Engineering,” in *Proceedings of the 18th Hawaii International Conference on Systems Sciences*, 1985.

[92] D. A. Abramson and J. Rosenberg, “Supporting a Capability Based Architecture in Silicon,” in *Proceeding of the 4th Microelectronics Conference*, 1985.

[93] J. Rosenberg and J. L. Keedy, “Object Management and Addressing in the MONADS Architecture,” in *Proceedings of the International Workshop on Persistent Object Systems, 1987*, Appin, Scotland, 1987.

[94] J. L. Keedy and J. Rosenberg, “Support for Objects in the MONADS Architecture,” in *Proceedings of the International Workshop on Persistent Object Systems*, Newcastle, Australia, 1990.

[95] 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.

[96] 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.

[97] R. L. Sites, Alpha Architecture Reference Manual, Digital Press, 1992.

[98] F. J. Corbato and V. A. Vyssotsky, “Introduction and Overview of the MULTICS System,” in *Proceedings of the 1965 Fall Joint Computer Conference*, 1965.

[99] A. Bensoussan, C. T. Clingen and E. C. Daley, “The MULTICS Virtual Memory: Concepts and Design,” *Communications of the ACM*, vol. 15, no. 5, pp. 308-318, 1972.

[100] J. L. Keedy and P. Brössler, “Implementing Databases in the Monads Virtual Memory,” in *Proceedings of the Fifth International Workshop on Persistent Object Systems, Design Implementation and Use*, San Miniato (Pisa), Italy, 1992.

[101] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison, “An Approach to Persistent Programming,” *The Computer Journal*, vol. 26, no. 4, pp. 360-365, 1983.

[102] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison, “PS-Algol: A Language for Persistent Programming,” in *Proceedings of the 10th Australian National Computer Conference*, Melbourne, Australia, 1983.

[103] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle and M. P. Atkinson, “The Napier Type System,” in *Proceedings of the 3rd International Workshop on Persistent Object Systems*, 1989.

[104] G. Bell and W. D. Strecker, “Computer Structures: What Have We Learned From The PDP-11?,” in *Proceedings of the 3rd Annual Symposium on Computer Architecture*, 1976.

[105] F. G. Soltis, Inside the AS/400, Loveland, Colorado: Duke Communications International, 1996.

[106] “Software Engineering: Concepts and Techniques,” in *Proceedings of the NATO Conferences*, New York, 1976.

[107] O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Structured Programming, New York, 1972.: Academic Press, 1972.

[108] N. Wirth, “Program Development by Stepwise Refinement,” *Communications of the ACM*, vol. 14, no. 4, pp. 221-227, 1971.

[109] J. Guttag, “Abstract Data Types and the Development of Data Structures,” *Communications of the ACM*, vol. 20, no. 6, pp. 396-404, 1977.

[110] P. Wegner, “Concepts and Paradigms of Object-Oriented Programming,” *ACM SIGPLAN OOPS Messenger*, vol. 1, no. 1, pp. 7-87, 1990.

[111] O. J. Dahl, B. Myhrhaug and K. Nygaard, The Simula 67 Common Base Language, Oslo: Norwegian Computer Centre, 1968.

[112] K. Jensen and N. Wirth, Pascal User Manual and Report, 2nd ed., New York: Springer, 1978.

[113] A. Goldberg and D. Robson, Smalltalk-80: The Language and its Implementation, Reading, Mass.: Addison-Wesley, 1983.

[114] N. Wirth, Programming in MODULA-2, Springer, 1982.

[115] F. J. Brooks, The Mythical Man Month: Essays on Software Engineering, Addison Wesley Publishing Co., 1975.

[116] D. L. Parnas, “On the Criteria to be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[117] D. L. Parnas, “Information Distribution Aspects of Design Methodology,”

in *Proceedings of the 5th World Computer Congress*, 1971.

[118] D. L. Parnas, "A Technique for Module Specification with Examples," *Communications of the ACM,* vol. 15, no. 5, pp. 330-336, 1972.

[119] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM,* vol. 17, no. 10, pp. 549-557, 1974.

[120] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronisation by Path Expressions," in *Lecture Notes in Computer Science, Vol. 16*, Springer, 1974.

[121] 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.

[122] J. L. Keedy, K. Espenlaub, G. Menger and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology,* vol. 3, no. 1, pp. 101-121, 2004.

[123] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Call-out Bracket Methods in Timor," *Journal of Object Technology,* vol. 5, no. 1, pp. 51-67, 2006.

[124] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Persistent Objects and Capabilities in Timor," *Journal of Object Technology,* vol. 6, no. 4, pp. 103-123, May-June 2007.

[125] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Persistent Processes and Distribution in Timor," *Journal of Object Technology,* vol. 6, no. 6, pp. 91-108, 2007.

[126] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, "Security and Protection in Timor Programs," *Journal of Object Technology,* vol. 7, no. 4, pp. 123-138, 2008.

[127] J. L. Keedy, G. Menger, C. Heinlein and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Net.ObjectDays*, Erfurt, Germany, 2002.

[128] J. L. Keedy and I. Richards, "A Software Engineering View of Files," *Australian Computer Journal*, vol. 14, no. 2, pp. 56-61, 1982.

[129] 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.

[130] F. Henskens, A Capability-Based Distributed Shared Memory, Newcastle, N.S.W.: Ph.D. thesis, University of Newcastle, NSW, Australia, 1991.

[131] J. L. Keedy and B. Freisleben, "On the Efficient Use of Semaphore Primitives," *Information Processing Letters*, vol. 21, no. 4, pp. 199-205, 1985.

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 companies and then a Freelance Management Adviser in Munich in 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.

Valuable contributions to TIMOR were made by

– 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.