Why Speedos executes Threads entirely in-process

J.L.Keedy
keedy@jlkeedy.net
formerly Professor/Honorary Professor at the University of Newcastle, NSW and Monash University, Melbourne
the Technical University of Darmstadt, the University of Bremen and the University of Ulm in Germany

Abstract

The paper examines the issue of processes and their threads, explaining the superiority of the in-process method with respect both to the level of parallelism attainable and to its protection advantages. It then points out that synchronisation is necessary and also that installations with special needs with respect to CPU scheduling of threads can replace the User Thread Scheduler. Finally the highly secure Speedos technique for logging users in and out is explained.

1 Introduction

As Lauer and Needham pointed out [1] there are two fundamental models for decomposing an operating system into processes. The more obvious of the two, which is used in most operating systems, involves having a separate process for carrying out each operating system activity. When a user process wishes to use a service of a particular operating system module it sends a message from its own process to the module offering the required service. We refer to this approach as *out-of-process* because the service is carried executed out of the user's process. This is illustrated in Figure 1.

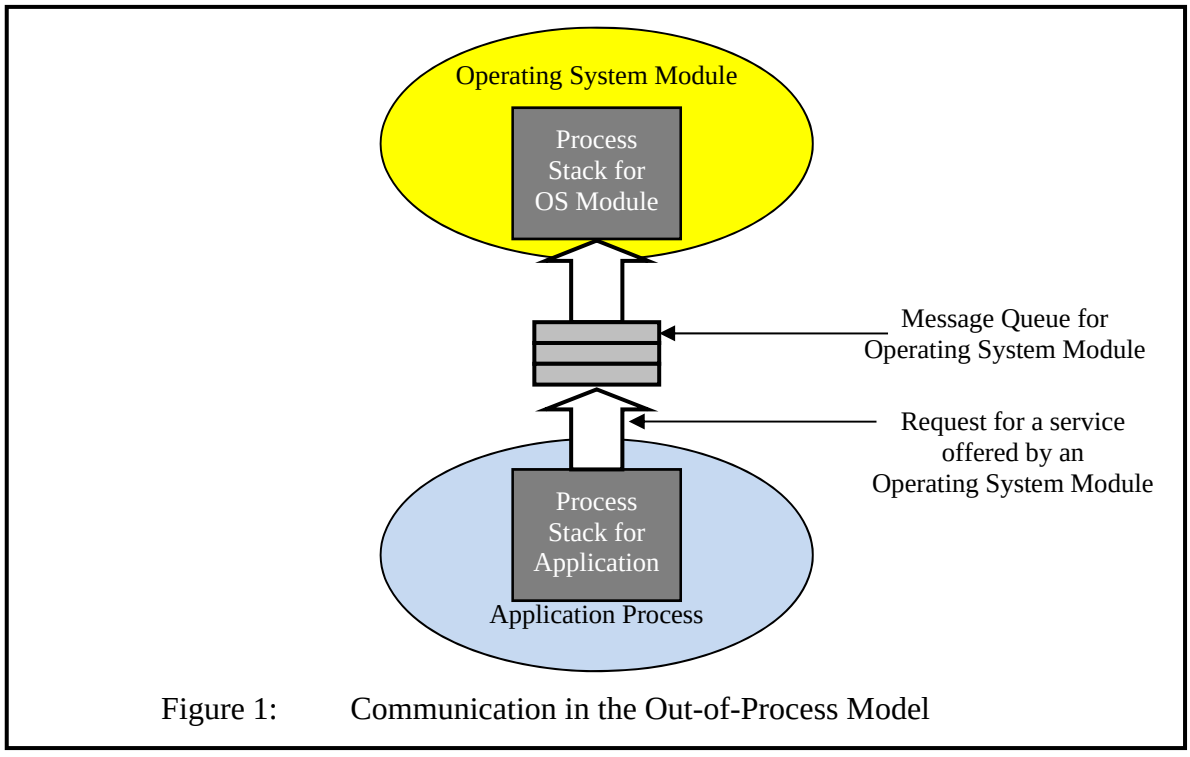

Figure 1: Communication in the Out-of-Process Model

In the second model operating system services are provided in a 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 on the stack of the user requesting the operating system service as a procedure call (see Figure 2).

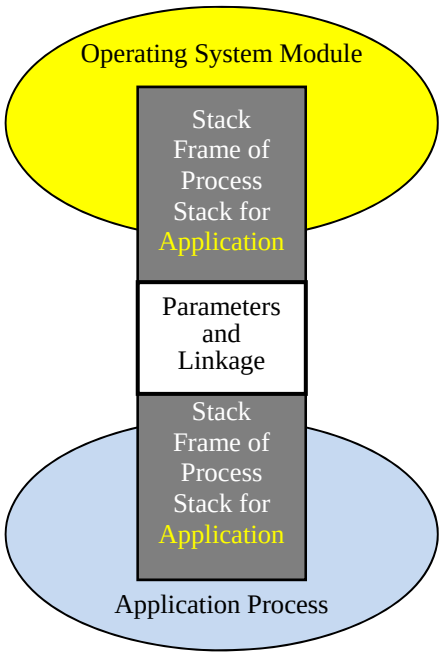

Figure 2: Communication in the In-Process Model

Lauer and Needham reached the conclusion that these two approaches are duals of each other. However, they overlooked two significant points, which are discussed in detail in my former student Kotigiri Ramamohanrao's PhD thesis [2]. This work was carried out in the context of the MONADS project, and the MONADS-PC computer was based on the in-process approach [3]. This paper looks more closely at the advantages of this approach, which is now also an integral part of the Speedos design [4].

2 Comparison of the Dynamic Properties of both Approaches

The two models are not equivalent in terms of their dynamic properties, e.g. with respect to the level of parallelism attainable in each.

2.1 Runtime Parallelism

Since a process is a unit of execution 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 out operating system tasks. In fact this is quite the opposite of the truth. In a system in which a process is statically assigned to provide a particular service (i.e. the out-of-process model), this service must normally 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[1].

By 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 processes executing in the same module need to access the same data structure concurrently. If these processes are only *reading* the data there is no problem, but if they have to *modify* it, then the data structure could potentially become incon-

[1] It would be possible to associate more than one process with each operating system module, but that would create more problems than it solves.

sistent, leading to wrong results. This issue was resolved in MONADS using semaphores [5].

The important point here is that processes executing in the same operating system 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 local data on their private process stacks, which are not shared and therefore cause no problems. But even if they are accessing shared data they may not be modifying it but only reading it. So in practice more parallelism can be achieved in an in-process design.

2.2 Charging for the Use of the CPU

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, 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.

2.3 Priority Scheduling of the CPU

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.

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.

3 Specific Advantages for Speedos

From the general properties of the two process structuring models so far discussed the superiority of the in-process model has been clearly established. But it also turns out that this model also has a number of specific advantages for Speedos.

Like its forerunner Monads, Speedos is a persistent system. It does not have a file sys-

tem in the conventional sense[2]. Its persistent virtual memory [6] is populated by information-hiding modules [7, 8, 9] with potentially multiple entrypoints which can be invoked using an in-process *inter-module call* kernel instruction. If the destination module is located on a different node from the caller, the kernel implements this as a *remote inter-module call.* Inter-module calls are all executed using the in-process model and the necessary linkage is held on a kernel managed stack.

A process in Speedos is actually an information-hiding module in the persistent memory, which holds a number of threads. These threads carry out computations for the owner of the module. Note therefore the change in terminology from this point. What in sections 1 and 2 were referred to as "processes" are now called "threads". A Speedos process is simply a persistent file module containing thread stacks and related information used primarily by the system kernel. The threads themselves are executable units which are scheduled at appropriate times by a "User Thread Scheduler" (UTS). Each Speedos node has its own UTS. These are fully independent of each other.

The kernel has a thread stack for each thread, which inter alia is used to pass and return parameters for inter-module calls. A thread can only access its parameters via special parameter registers. The thread stack also holds other security-sensitive information about the thread, including for example a thread security register (see volume 2 chapter 22 of [4]).

3.1 Identification of Users and their Processes

From the protection viewpoint the most important point is that by rigorously using the in-process model the unique identification of every user of any Speedos system is possible, since his unique identifier is based on the identifier of the first container which is created for him. There are no anonymous users, and knowledge of his identifier makes it possible always to trace him. This implies, for example, that Speedos users should not be troubled by junk mail. But far more important, if they do manage to gain access to a Speedos system and attempt to carry out some criminal activity such as attempting to steal information or blowing up a power station they will not only be unsuccessful[3] but they will leave telltale information about their identity. How this can happen is explained in [10], which gives examples of how protection problems can be solved.

3.2 Synchronisation and Thread Scheduling

As indicated above, using the in-process model requires that when two or more threads are active in a module some form of synchronisation is necessary. How this functions in Speedos is described in volume 2 chapter 21 of [4]. From the programmer's viewpoint it is primarily a matter of placing synchronisation primitives at the correct point in the code.

A central scheduling module in any system is probably the module which is most activated. The primary job of the UTS in Speedos is to schedule threads (i.e. to determine which threads can use the CPU in what order). In an out-of-process system a process switch is necessary each time a user sends a message to an operating system. Such overheads are entirely avoided in Speedos, since the same thread continues to execute when an operating system module is called. The kernel manages a change of environment, but this occurs without the central scheduler being involved, i.e. without any scheduling activity.

---

[2] This is possible because it uses 256 bit virtual addresses which are unique across all Speedos systems. How these can be efficiently translated into main memory addresses is explained in chapter 26 of volume 2 of [4].

[3] provided of course that the owners of the resource being attacked have taken steps to prevent this.

3.3 Logging In and Out

Because the entire content of the virtual memory is persistent, not only files and programs but also processes and their threads are persistent. This means that the current state of a process and its threads is also preserved automatically when users log out. Thus when a user wishes to log in again, he can in principle simply resume his work in the same state that he had left it. Consequently there is no reason for the system automatically to delete his process on logout and to start a new process for him when he logs in again. That is clearly more efficient and more convenient for users.

But the idea of persistent processes brings a further advantage. It opens up the way for adding greater security to the login mechanism. If the final action which a user takes before logging out is to call a "logout" module (which he must do anyway to warn the process scheduler that his process should be temporarily deactivated) he can do this from a logout module *of his own choice*. Such a module (which is owned by the user, who can also determine what it does) can contain arbitrary checks devised by the user to check his own identity. This need not be a simple password, it can for example be a dynamic password, a cognitive password and/or whether the person attempting to log in has to conform to some required actions[4]. The kernel's part in the login and logout mechanism is trivial. In the case of logging in simply advises the process scheduler that the user is active. This then activates the user's thread in the latter's logout module, which then validates the user or, if the checks fail, informs the process scheduler to deactivate the process again. Notice also that there is no central file which can be hacked to obtain login information.

It will eventually be generally recognised that Speedos can really prevent criminality, stop the theft of important files and prevent damage to infrastructure, etc. [10]. Once Speedos has become established, I believe that a software components industry will develop, as was originally envisaged by [11]. One of the many possibilities for such an industry will be to develop logout modules.

Funding: This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors

[4] Login Security checking is discussed in Making Computers Secure, volume 1, chapters 4 and 15, and in volume 2, chapter 22 which can be downloaded from the Speedos Website https://www.speedos-security.org/

References

[1] 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.

[2] K. Ramamohanarao, "A New Model for Job Management Systems," *PhD. Thesis, Monash University, Australia,* 1980.

[3] J. Rosenberg and D. A. Abramson, "MONADS-PC: A Capability Based Workstation to Support Software Engineering," *Proceedings of the 18th Hawaii International Conference on Systems Sciences,* pp. 515-522, 1985.

[4] J. L. Keedy, Making Computers Secure, Speedos Website, https://www.speedos-security.org/, 2021.

[5] 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.

[6] J. L. Keedy, "Persistent Programming with Speedos and Timor," in *SPEEDOS Website (https://www.speedos-security.org/),* 2024.

[7] D. L. Parnas, "Information Distribution Aspects of Design Methodology," in *Proceedings of the 5th World Computer Congress*, 1971.

[8] 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.

[9] D. L. Parnas, "A Technique for Module Specification with Examples," *Communications of the ACM,* vol. 15, no. 5, pp. 330-336, 1972.

[10] J. L. Keedy, "Protecting and Confining Information with Speedos," *Speedos Website,* 2024.

[11] M. D. McIlroy, "Mass Produced Software Components," in *Software Engineering: Concepts and Techniques, Petrocelli-Charter*, New York, 1968.

[12] 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.