A Detailed Look at Operating System Processes

B. H. Venter
Department of Computer Science, University of Port Elizabeth P.O. Box 1600, Port Elizabeth, 6000 South Africa,
e-mail: csabhv@upe.ac.za

Abstract

An operating system provides, among other things, an operational definition of a process. The concept of a process is one of the fundamental concepts of Computer Science, and the designer of an operating system must strive to provide a definition that is simple to understand, does not violate the intuitive notions one has about processes, and is simple to implement efficiently on a wide range of computer systems. On the other hand, the definition should not fail to provide the functionality that existing operating systems have, by user demand, gradually evolved into providing. This paper presents a framework for discussing the operational definition of a process, and uses this framework to discuss systematically some of the more important decisions and trade-offs regarding processes, that the designer of a new operating system must make.
Keywords: *operating systems, process, light-weight process, memory-sharing, interrupts, exceptions, real-time*
Computing Review Category: *D.4.1 OPERATING SYSTEMS, Process Management*

Received June 1987, Accepted July 1987

1 Introduction

There is no universally agreed upon definition for the concept of a process, and the definitions offered in the literature are often inadequate (for example, those in [1]). And while it is difficult to fault abstract definitions such as [2], they are not specific enough to be of much use to an operating system designer.

An operating system amounts to an operational definition of a process, and no two operating systems amount to exactly the same definition. The designer of a new operating system must strive to provide a definition that is simple to understand and use, does not violate one's intuitive notions about processes, and is simple to implement efficiently on a wide range of computer systems. On the other hand, the definition should not fail to provide the functionality that existing operating systems have, by user demand, evolved into providing.

The aim of this paper is to provide a framework for discussing operational definitions, particularly with regard to functionality, and then to use this framework to discuss systematically some of the more important decisions and trade-offs regarding processes, that a designer of a new operating system must make.

The considerations outlined in this paper have formed the basis for the process concept offered by a new operating system that the author has designed [5]. The process concept of this operating system differs significantly from the definition offered by UNIX [4, 6], a system that many proponents proclaim as suitable for adoption as *the* standard operating system, but which offers an inadequate definition of a process.

However, this paper is not a critique of UNIX or any other operating system, nor a defence of the author's proposal. The paper aims to identify and discuss the general principles involved, and readers are invited to make their own comparisons and to draw their own conclusions.

2 The Framework

The most basic property of an operating system process is that it executes on a virtual processor created by the operating system. Each such virtual processor has its own set of registers and has access to a subset of the memory of the real processor used to implement it. Furthermore, the instructions of a virtual processor can be regarded as being extended with extra instructions, namely the system calls offered by the operating system, which allow it to interact with the devices and other virtual processors in its environment. A process can thus be defined as a virtual processor seen in conjunction with the set of instructions (program) that is executing.

The difference between the operational definitions offered by different operating systems can be described as differences in the capabilities of the virtual processors created by these systems.

In the following sections, the desired capabilities of a virtual processor are discussed in the light of a list of questions. Some of these questions may seem to have trivial answers, but all have been answered in different ways by different operating system designers. The list is:

- Should two different virtual processors be able to access the same area of physical memory?
- Should virtual processors use virtual or real addresses to access memory?
- Should virtual processors be able dynamically to vary the amount of memory allocated to them?
- Should a virtual processor be interruptible? That is, should a non-deterministic event in its environment be able to start the execution of an interrupt service routine?
- Should a virtual processor be able to handle exceptions caused by instructions executed by it?
- To what extent should a virtual processor be controllable by an external agent?
- What should the initial state of a newly created process be?
- Should it be possible to make assumptions about the rate of execution of a virtual processor in "real time", as well as about the elapsed "real time" needed to finish executing a routine after the occurrence of an event in its environment?

3 MEMORY SHARING

Data structures kept in shared memory areas have long been the principal concept used to achieve co-operation among different processes. However, along with the obvious advantages, allowing processes to share memory has certain disadvantages:

- Processes are implicitly given access to their private memories as part of their creation. However, to provide processes with access to shared memory areas, an operating system has to provide an explicit mechanism. Such a mechanism must provide a naming scheme and must not allow security to be compromised. Allowing memory sharing thus adds complexity to an operating system.
- Two processes cannot share a memory area efficiently unless their virtual processors can be implemented by physical processors with access to a common physical memory. This either limits the class of hardware that can be used, or necessitates an additional mechanism to control the physical processors used to implement virtual processors – thus adding more complexity to the operating system.
- Further mechanisms must be provided to allow processes to synchronize their access to shared memory areas. It is difficult, if not impossible, to make these mechanisms general, safe, and efficient at the same time. Moreover, they add yet more complexity to the operating system.

It is possible for an operating system designer to accept these disadvantages as unavoidable and to provide generalized mechanisms that allow arbitrary processes to share memory (as is done in UNIX System V [6]). However, since non-shared memory multi-processors (for example, hundreds of workstations inter-connected by means of a high-speed local-area network) are likely to become more common, if not prevalent, it makes sense to avoid memory sharing among processes. It therefore also makes sense to ask whether it is necessary to provide complex, generalized mechanisms to allow arbitrary processes to share memory.

Memory sharing only makes sense if the sharing processes are

a) executed by a single physical processor, or

b) executed by separate physical processors that share access to a common physical memory.

a) Assuming that the processes that are co-operating via memory sharing are all executed by the same physical processor, the question arises why they should not be consolidated into a single process. The multiple processes cannot together execute faster than a single process since the available real processor cycles remain fixed. In fact, the overhead incurred by extra context switches and process synchronization make such multi-process formulations slower than single-process formulations.

The main argument for having multiple co-operating processes on the same physical processor is that multiple-process solutions can sometimes be simpler to program than single-process solutions. A typical case is a server process that may receive requests from different clients in quick succession, each of which may involve waiting for some event to happen in the environment of the server (for example the completion of a disk access). Unless a single-process server explicitly breaks up each request into multiple subrequests, none of which require waiting, and interleaves the execution of different requests on this complicated basis, the single-process server will be slower than a multi-process server since context switching among the various co-operating processes automatically achieves the desired interleaving.

However, it is straightforward to incorporate a local scheduler into the code of a single-process server. This makes it possible to formulate the server as a set of memory-sharing, co-operating "light-weight" processes, while actually executing the server as a single operating system process. Since the "light-weight" scheduler is "user code", it can dispense with many of the precautions that the operating system "heavy-weight" scheduler must take. It can therefore be much more efficient to construct a set of memory-sharing, co-operating processes without the aid of expensive operating system mechanisms.

There will, of course, be cases where memory-sharing, co-operating processes need the separate identities and relative isolation available only to operating system, "heavy-weight" processes. However, these cases should be comparatively rare, and confined mainly to "systems programming" situations. It thus seems reasonable to require these processes to be declared as "privileged and trusted" (by suitably authorized users) and to be explicitly tied to specific physical processors. It is then possible to provide a simple mechanism that dispenses with most of the complications of generality and security checking to enable such privileged processes to obtain access to shared memory areas.

Another use for memory sharing among processes is to allow them to access common blocks of code, such as language-provided run-time systems. However, such code sharing need not affect the logic of a process, and the issue can thus be relegated to a convention between the linker and loader, rather than be treated as part of the definition of a process. In some systems, memory sharing is also used by debuggers. However, such debuggers can be accommodated by providing "privileged and trusted" server processes that allow them to achieve the same ends.

b) Assuming now that there are multiple physical processors that have access to a shared physical memory, and therefore that the memory-sharing processes can be executed in true parallel, the "light-weight" process solution is no longer applicable, and the "privileged and trusted" solution too primitive and restrictive. Furthermore, the existence of "teams" of closely co-operating processes executing in true parallel opens up new opportunities and complications, such as "co-scheduling" [3] and non-blocking ("spin-lock") forms of synchronization.

It is, however, possible to accommodate such hardware without resorting to generalized mechanisms by extending the "light-weight" process solution. On non-shared memory systems, the routines for implementing a "light-weight" scheduler will be provided as a standard library, to be linked into the code of the "heavy-weight" process hosting the collection of "light-weight" processes. However, on shared-memory multi-processors these routines will simply invoke "hidden" system calls, which see to it that the "light-weight" processes are executed on different physical processors, while otherwise being part of a single "heavy-weight" process.

The discussion on shared memory can be summarized as follows:

It is easier and more efficient to implement a process concept that does not allow processes to share memory. Furthermore, an application formulated as a set of processes that do not share memory can be executed on a wider range of computer systems, which is highly desirable.

However, when it is essential to exploit a shared-memory multi-processor, this can be achieved by introducing "light-weight" processes that are "internal" to normal "heavy-weight" processes. These "light-weight" processes can also be supported on a single processor by incorporating an internal "user-code" scheduler into the code of a single "heavy-weight" process. Thus, applications that explicitly exploit shared-memory multi-processors can still be ported to other kinds of computer systems, albeit with a performance penalty in some cases.

It should be noted that the concept of a "light-weight" process is independent from the concept of a "heavy-weight" process. The rest of this paper is exclusively concerned with "heavy-weight" processes.

4 VIRTUAL ADDRESS SPACES

A virtual address space typically allows a virtual processor to view its subset of the physical memory as one or more contiguous blocks of memory, starting at fixed addresses such as zero.

Providing virtual processors with such virtual address spaces has several disadvantages: It limits the class of suitable hardware, most processor caches must be invalidated after each context switch, and the operating system is complicated by the need to cope with different address spaces.

However, appropriate memory management hardware units are becoming common, and virtual address spaces facilitate the writing of compilers. Furthermore, virtual address spaces allow a process to comprise a number of disjoint areas of physical memory and make it possible to provide processes with virtual memories that are larger than the physical memory (however, this is becoming increasingly less important as physical memories become larger).

Moreover, widely-used operating systems such as UNIX provide processes with virtual address spaces. It may thus be more difficult to port programs developed for existing operating systems to a new operating system that does not provide processes with virtual address spaces.

Thus, the advantages or providing virtual address spaces appear to outweigh the disadvantages.

5 DYNAMIC MEMORY ALLOCATION

Allowing a process to change the amount of memory available to it dynamically is highly desirable. For example, a process such as a compiler then need use no more memory than is required by the input of a particular run, and a process such as a sort utility can use as much (or as little) memory as is available during a particular run.

However, to be implemented efficiently, dynamic memory allocation requires memory management hardware that allows the virtual processor to access a potentially large number of disjoint segments of real memory. Furthermore, it complicates both the operating system resource allocation policies and the code of the dynamically sized processes, since the amount of memory needed by a process is not known in advance and it may be impossible to grant a request for extra memory.

These difficulties can be ameliorated effectively be requiring the linker to supply the loader (via parameters in the process image) with the maximum amount of extra memory a process may request, as well as the minimum amount of extra memory the operating system must guarantee. These parameters allow particular implementations of an operating system to overcome limitations of the memory management hardware by pre-allocating or partially pre-allocating extra memory for a process when it is created. They also facilitate the writing of dynamically sized processes as a minimum amount of memory can be guaranteed to be dynamically allocatable.

6 VIRTUAL INTERRUPTS

Whether or not to make virtual processors interruptible is a somewhat contentious issue. Anyone who has programmed an interruptible processor will recall at least one extremely hard-to-find error caused by subtle interactions between non-deterministic interrupts. None but the best and bravest (or most foolish) programmers will venture anything of substance that the final "debugged" system running on an interruptible processor will be completely free of errors.

The main argument for having interrupts is that they allow fast responses to external events and avoid the need to check repeatedly whether the event has occurred. However, these properties can be obtained in a non-interruptible system by packaging interrupts as arriving messages, by providing a process with suitable mechanisms for suspending its execution pending the arrival of a message, and by providing mechanisms that enable a process to respond quickly to the receipt of a message.

The only processes for which these "no interrupt" mechanisms are clearly less suitable than interrupt-based mechanisms are the device driver processes that must ultimately deal with the real interrupts on the real machine. These interrupts should preferably be dealt with as rapidly as the hardware allows, and the less software packaging is involved, the better. However, since device driver processes can reasonably be required to be designated as "privileged and trusted", it suffices to give such processes access to the real interrupts via a simple (but dangerous) mechanism involving no overhead for other processes.

As will be seen below, the programmer who really wants an ordinary process to be asynchronously interruptible can readily achieve this by means of a protocol between the process and its parent. Furthermore, the entire mechanism and the overhead of the mechanism are firmly under the control of the programmer.

7 EXCEPTIONS

Real processors often use the same mechanism to handle exceptions and interrupts. However, since exceptions are usually caused by a processor trying to execute an erroneous instruction, it makes no sense for an operating system to package an exception as a message to the process, and then to allow the virtual processor to carry on executing until the process explicitly accepts the message.

Clearly, after an exception, the virtual processor must suspend the execution of the instruction stream that caused the problem, and either carry on with a different instruction stream, or have the process terminated. The mechanism for doing this should preferably involve minimal overhead for the ordinary process, which should not generate exceptions and should be terminated when it does. The mechanism should also be machine independent so as not to decrease portability, and should provide the programmer or language runtime system with complete control over what happens when an exception is generated.

The simplest mechanism that will do the job is as follows: When a process generates an exception, it is suspended, and the event is packaged into a message that is sent to its parent process. The parent of a process can then cause its suspended subprocess to be restarted at any point within the subprocess address space. Alternatively, the parent can simply terminate the subprocess.

This mechanism allows the operating system to handle exceptions using a simple non-interrupting scheme that exacts no overhead other than the cost of sending a message (the minimum cost in a non-shared memory multi-processor). The policy decisions on what to do about an exception are relegated to a machine-independent protocol between the process and its parent process. Moreover, the mechanism provides the programmer and/or language run-time system with complete control over what happens when a process generates an exception.

8 CONTROL BY EXTERNAL AGENTS

External agents, such as devices and users, can always be represented by corresponding processes. Thus the issue is the extent to which one process can control another. As previously indicated, a process can restart a suspended subprocess, as well as terminate a subprocess. It can also be informed whenever a subprocess terminates itself or is suspended.

It may seen unreasonable to restrict the privilege of controlling a process to its parent process. However, restricting the privilege avoids the need to introduce an explicit "right to control" granting mechanism and makes it easier to check whether a process has the right to make a control request (especially in a distributed implementation of the operating system). Also, a programmer can synthesize control facilities in non-parent processes by letting the parent execute requests communicated to it by these processes. The restriction, therefore, simplifies the operating system without impairing functionality. Furthermore, the overhead of providing extended control rights is limited to those applications for which these are required.

An additional control feature often provided by operating systems is the ability to cause an interrupt or exception in another process. This can be achieved by allowing a process to suspend a subprocess, whereafter it can restart the subprocess at the address of the interrupt/exception handler. This allows, for example, a process to extricate a subprocess from an endless loop, and to restart it inside a "cleanup and reset" procedure.

Allowing a process to be suspended at an arbitrary point in its execution, and then to be restarted at another, causes some complications. Firstly, the "interrupt" handler entered when the suspended process is restarted may eventually wish to resume the interrupted control flow (in true interrupt handler fashion). Thus, when a process is suspended, the address of the instruction that was about to be executed must be saved on the process stack. Secondly, the interrupt handler may need to know the reason for the interrupt. Consequently, the parent must supply a reason code when it suspends a subprocess; this code is then stacked along with the "return" address. (When a subprocess suspends itself, it too supplies a reason code, and when it is suspended because of an error, the system provides a reason code.)

Suspending a subprocess so that it can be resumed cleanly from the point of interruption can be very difficult if the subprocess is inside a system call at the time that the suspension request is received (which can be at any time on most multi-processors). Consequently, a suspension request issued by a parent while the subprocess is inside a system call takes effect only when the system call returns. If the parent proceeds to restart a subprocess that is still waiting to be suspended, the completion of the parent's restart request is delayed until the subprocess can be suspended.

However, if a subprocess is inside a system call that may never return, for example when it is waiting for an input/output operation that will never be completed, then such a scheme will cause the parent to be delayed indefinitely when it tries to extricate its subprocess form an indefinite delay. Consequently, such system call must be interruptible.

When interrupted inside such calls, the stacked reason code is modified, and additional information is placed on the stack. An interrupt handler that wishes to return to the point of interruption can use this stacked information to resume the interrupted system call. On the other hand, an exception handler that must extricate the process form an erroneous control flow, rather than resume it, must be able to regard an interrupted system call as completed.

The complications that arise when a parent is allowed to suspend its subprocesses raise the question whether it is possible to dispense with this mechanism. However, the only alternative to suspending and restarting a run-away subprocess is to terminate it. When a process is terminated, all of its communication links are severed and all its subprocesses are terminated – otherwise, other complications would arise.

The cascading effects of terminating a process forming part of a set of co-operating processes may be too severe to be acceptable to a multi-process, real-time, fault-tolerant application. Albeit somewhat complicated, the semantics of suspend/restart seem preferable. And, even though complicated to describe, the suspend/restart mechanism can be implemented without exacting much overhead cost.

With regard to the automatic termination of subprocesses, note that a process can nevertheless create another subprocess that will survive itself. A process would do this by not creating the new process as its own subprocess, but by requesting an operating system provided non-terminating "server" process to create and control a subprocess on its behalf. Such a server would provide the functions of the background batch queues found in many operating systems.

9 THE INITIAL STATE OF A PROCESS

There can be little argument that it is desirable to control fully the initial state of a process. However, there are at least two ways of achieving this. One is to allow the system linker to create a fully specified process image in a file, and for the loader to load the image, exactly as specified, into memory. Another way, taken by UNIX, is to duplicate the current image of a process when it creates a subprocess, and also to allow a process (usually a new subprocesses) to replace its current image with one specified in a file.

However, creating a new process by duplicating the memory image of the parent process is not sensible if the subprocess immediately replaces the image with one from a file (by far the most common case). Furthermore, such an unnecessary duplication can be very expensive operation if the image must be copied from one physical memory to another via a network.

Once a new process is created, it can either immediately start executing, as is the case for a UNIX process, or it can remain in a suspended state until explicitly started by its parent process. Such an initial suspension allows a parent process to create several subprocesses, and to set up communication links between them, while they are in a known state. A parent can also connect the communication ports of subprocesses to servers, files, or devices, or transfer the use of some of its own communication links to subprocesses. Initial suspension is thus preferable.

Another initialization issue concerns the interface of a new process to its environment. A new UNIX process inherits copies of all the file descriptor of its parent. This is fine when the descriptors provide read-only access to actual files. However, if one of the file descriptors represents a terminal and both parent and child use it simultaneously, chaos results. In the operating system designed by the author, the UNIX concept of a file descriptor is superceded by a generalized "communication link", and it can be both complicated and undesirable automatically to provide a new process with duplicates of all the communication links of its parent. Consequently, a new process is created without any communication links, and it is up to its parent explicitly to provide the links that the new process expects to be able to use without establishing them itself.

10 RATE OF EXECUTION

If the operating system is to support real-time applications, or even just interactive applications, there is little choice but to allow a programmer to make some assumptions about the rate of execution of a process. It is also necessary to allow the rate of execution to be influenced by the programmer.

Most operating systems allow priorities to be attached to processes, and some implement feedback schemes that dynamically adjust the priorities of processes. While workable, and often successful, this approach does suffer from the drawback that priorities must be chosen and controlled carefully. When the number of real-time processes become large, it becomes very difficult to use priorities with predictable effect.

A simpler approach is to allow the rate of execution of a process to be specified directly, rather than be influenced indirectly via priorities. For example, the speed of a processor can be rated on some absolute scale and the rate of execution of a process expressed as a number on the same scale, with the following interpretation: Let Y be the speed of the processor and X the desired rate of execution of a process. The process should then receive at least X/Y of the processor's cycles, say every 100 milliseconds, while the process remains ready to execute.

Of course, to guarantee that a process will progress at least as rapidly as at its desired rate of execution, the sum of the rates of all the processes that can be ready at the same time must not exceed the speed of the processor. However, it should be easier for a real-time application designer to ensure that this is true, than it would be to ensure that a set of priorities will result in the desired progress by each process.

To aid real-time application designers further, an operating system should also be able to distinguish among "real-time" and "ordinary" processes, with real-time processes preempting ordinary processes. A real-time application designer then need not take non-real-time processes into account at all.

11 CONCLUSION

In striving to provide the simplest possible definition of a process, while providing as much functionality as possible, the impact of various trade-offs must be evaluated carefully.

This paper has identified a number of the more important aspects of functionality and has discussed the issues and trade-offs involved, particularly in terms of the simplest mechanisms that can be used to provide the needed functionality.

By providing the necessary functionality from the start, it is possible to avoid the problems associated with piecemeal evolution. In this regard, it is instructive to compare the complex definition of a process offered by UNIX System V to the simple, but incomplete, definition offered by early versions of UNIX.

References

[1] Deitel H. M., An Introduction to Operating Systems, Addison-Wesley, Reading Mass., 1984

[2] Horning J. J. and Randell B., Process Structuring, *ACM Computing Surveys,* **5**,1, 5-30, March 1973

[3] Ousterhout J. K., Medusa: a distributed operating system, UMI Research Press, Michigan, 1981

[4] Ritchie D. M. and Thompson K., The UNIX time-sharing system, *The Bell System Technical Journal,* **57**, 6, 1905-1929, July 1978

[5] Venter B. H., The design of a new general-purpose operating system, Ph.D. Thesis, University of Port Elizabeth, South Africa, 1987

[6] XELOS Programmer Reference Manual, Perkin-Elmer Corporation, New Jersey, 1984