Operating Systems (Client)

Lecture 2 – Process Management

Higher Cert. in Science in Computing (IT) – BN002
Bachelor of Science in Computing (IT) – BN013
Bachelor of Science (Hons) in Computing – BN104

Dr. Kevin Farrell

itb
Institute of Technology
Blanchardstown
Institiúid Teicneolaíochta
Baile Bhlainséir

Contents

Figures

Tables

License

Feedback

Constructive comments and suggestions on this document are welcome. Please direct them to:

kevin.farrell@itb.ie

Acknowledgements

Thanks in particular go to the Dr. Anthony Keane, whose lectures this material was partly based on.

Modifications and updates

Version	Date	Description of Change
1.0	1st June 2005	First published edition.
1.1	10th August 2005	Minor formatting changes
1.2	18th October 2005	Minor formatting changes

1. Overview

1.1. Lecture Summary

The most fundamental building block in an operating system is the process. This Lecture tells the story of process management in operating systems. Having first explained the process concept, we introduce the notion of multiprogramming operating systems, which allow for concurrent execution of processes. Following on from this, we explain the circumstances in which these many processes may be created and destroyed. This motivates our consideration of operating system design, and a discussion on three different process state models, which describe the life-cycle of a process in an operating system. Each model addresses the deficiencies of the previous one, until we arrive at the five-state process management model. This is a generic model which describes all possible process states, which we expect to exist in an operating system. Having established this description of the process life-cycle, proceed to a discussion on process description and control in such a system, and in particular, the implementation mechanisms for processor modes, kernels and requests for system service. Finally, we finish with quite a detailed discussion on process-related issues in the UNIX System V Release 4 flavour of the UNIX operating system.

1.2. Learning Outcomes

After successfully completing this Lecture you should be able to:

- Describe the concept of a process
- Explain the difference between a program and a process
- Justify, using basic mathematics and sound reasoning, the validity of multiprogramming as a technique for improving system performance.

- Explain the differences between the general process state management models.
- Distinguish between user mode and kernel mode
- Explain the relative merits of the system-call approach to operating system design versus the message-passing approach.
- Describe process-management in the UNIX System V Release 4 flavour of UNIX.

1.3. Reading

Material for this lecture was gathered from a number of difference sources. Some key areas are described below.

Core Texts:

- "Operating System incorporating Windows and UNIX", Colin Ritchie.
- "Operating Systems", William Stallings, Prentice Hall, (4th Edition, 2000)

Multiprogramming, Process Description and Control

- "Operating Systems – A Modern Perspective", Gary Nutt, Addison Wesley, (2nd Edition, 2001).
- "Operating Systems", William Stallings, Prentice Hall, (4th Edition, 2000)

Process Management Models, Scheduling, UNIX System V Release 4:

- "Operating Systems", William Stallings, Prentice Hall, (4th Edition, 2000)

Additional recommended reading:

- "Modern Operating Systems", Andrew Tannenbaum, Prentice Hall, (2nd Edition, 2001).
- "Operating System Concepts", Silberschatz, Galvin & Gagne, John Wiley & Sons, (6th Edition, 2003).

1.4. Suggested Time Management

This Lecture should take between 2 and 4 hours of your study time:

- Lecture Content: 2 hours
- Further reading: 2 hours

1.5. Typographical Conventions

Throughout this Operating Systems Module, I have tried to use uniform typographical conventions; the aim being to improve readability, and lend understanding:

Key/Technical Terms:	**Bold/Underline**	for eg: **Multiprogramming**
Emphasis:	*Italics*	
Command names:	**`Courier/Bold`**	for eg: **`ls -l`**
Filenames/Paths:	`Courier`	for eg: `/home/kevin/cv.doc`

2. Introduction

The fundamental task of any modern operating system is process management. The OS must allocate resources to processes, enable processes to share and exchange information, protect the resources of each process from other processes and enable synchronisation among processes. To meet these requirements, the OS must maintain a data structure for each process, which describes the state and resource ownership of that process, and which enables the OS to exert control over each process.

3. Basic Concepts

The **processor** (CPU) is the part of the machine that performs the calculations and executes the programs. A **process** is a single instance of an executable program; for example, a single mathematical calculation is a process. A process is also known as a *task* or an *activity*. A **job** (or program) in an operating systems environment is a unit of work that is submitted by the user. The job becomes a process if it is accepted by the Operating System, and is allowed to be executed. A program can be thought of as code or instructions contained in a file stored on disk, whereas a process is the transfer of (at least some of) those instructions into main memory, and their subsequent execution; a process is therefore an *active* entity, whereas a program is a *passive* one.

3.1. Multiprogramming

In many modern Operating Systems, there can be more than one instance of a program loaded in memory at the same time; for example, more than one user could be executing the same program, each user having separate copies of the program loaded into memory. With some programs, it is possible to have one copy loaded into memory, while several users have shared access to it so that

they each can execute the same program-code. Such a program is said to be **re-entrant**.

The processor at any instant can only be executing one instruction from one program but several processes can be sustained over a period of time by assigning each process to the processor at intervals while the remainder become temporarily inactive. We use the terminology, **concurrent execution**, to describe a number of processes being executed over a period of time (rather than at the same time).

A **multiprogramming** or **multitasking** OS is a system executing many processes concurrently. Multiprogramming requires that the processor be *allocated* to each process for a period of time and *de-allocated* at an appropriate moment. If the processor is de-allocated during the execution of a process, it must be done in such a way that it can be restarted later as easily as possible.

There are two possible ways for an OS to regain control of the processor during a program's execution in order for the OS to perform de-allocation or allocation:

1. The process issues a **system call**, (sometimes called a software interrupt); for example, an I/O request occurs requesting to access a file on hard disk.
2. A hardware interrupt occurs; for example, a key was pressed on the keyboard.

The stopping of one process and starting (or restarting) of another process is called a **context change**.

In many modern Operating Systems, processes can consist of many sub-processes. This introduces the concept of a **thread**. A thread may be viewed as a sub-process; that is, a separate, independent sequence of execution within the code of one process. Threads are becoming increasingly important in the design of distributed and client-server systems and in software run on multi-processor systems.

3.1.1. How Multiprogramming Increases Efficiency

A common trait observed among processes associated with most computer programs, is that they alternate between **CPU cycles** and **I/O cycles**. For the portion of the time required for CPU cycles, the process is being executed; i.e. is occupying the CPU. During the time required for I/O cycles, the process is not using the processor. Instead, it is either waiting to perform Input/Output, or is actually performing Input/Output. An example of this is the reading from or writing to a file on disk. Example 3.1 shows a simple program pseudo-code which illustrates the concept of CPU and I/O cycles.

```
READ A, B                ←        I/O cycle
C=A+B                    ⎤
D=(A*B)-C                ⎥
E=A-B                    ⎥  ←     CPU cycle
F=D/E                    ⎦
WRITE A,B,C,D,E,F        ←        I/O cycle
STOP                     ←        termination
END
```

Example 3.1: Simple program pseudo-code illustrating the concept of CPU and I/O Cycles.

Prior to the advent of multiprogramming, computers operated as single-user systems. Users of such systems quick became aware that for much of the time that a computer was allocated to a single user, the processor was idle; when the user was entering information or debugging programs for example. Computer scientists observed that overall performance of the machine could be improved by letting a different process use the processor whenever one process was waiting for input/output. In a **uni-programming** system, if *N* users were to execute programs with individual execution times of $t_1, t_2, \ldots, t_N$, then the total time, t_{uni} , to service the *N* processes (consecutively) of all *N* users would be:

$$t_{uni} = t_1 + t_2 + \cdots + t_N$$

However, because each process contains both CPU cycles and I/O cycles, the time for which each process actually uses the CPU is a very small fraction of the total execution time for the process. So, for process, *i*:

$$t_{i(processor)} \ll t_{i(execution)}$$

where

- $t_{i(processor)}$ is the time process *i* spends using the CPU, and
- $t_{i(execution)}$ is the total execution time for the process; i.e. the time for CPU cycles plus I/O cycles to be carried out (executed) until completion of the process.

In fact, usually the sum of all the processor time, used by *N* processes, rarely exceeds a small fraction of the time to execute any one of the processes; i.e.:

$$\sum_{i=1}^{N} t_{i(processor)} < t_{i(execution)}$$

Therefore, in uni-programming systems, the processor lay idle for a considerable proportion of the time. To overcome this inefficiency, multiprogramming is now implemented in modern operating systems such as Linux, UNIX and Microsoft Windows. This enables the processor to switch from one process, *X*, to another, *Y*, whenever *X* is involved in the I/O phase of its execution. Since the processing time is much less than a single job's runtime, the total time to service all *N* users with a multiprogramming system can be reduced to approximately:

$$t_{multi} = maximum(t_1, t_2, \cdots, t_N)$$

3.2. Process Creation

When a new process is to be added to those currently being managed, the OS builds a data structure called a **Process Control Block** (PCB) (see later),

which is used to keep track of the process and the memory address space allocated to the process. These actions constitute the creation of a new process. *Four* common events can lead to the creation of a process:

- New batch job.
- Interactive logon
- Created by OS.
- Spawned by existing processes, e.g. parent/child.

3.3. Process Termination

There are many reasons for process termination:

- Batch job issues **halt** instruction
- User logs off
- Process executes a service request to terminate
- Error and fault conditions
- Normal completion
- Time limit exceeded
- Memory unavailable
- Bounds violation; for example: attempted access of (non-existent) 11th element of a 10-element array
- Protection error; for example: attempted write to read-only file
- Arithmetic error; for example: attempted division by zero
- Time overrun; for example: process waited longer than a specified maximum for an event
- I/O failure
- Invalid instruction; for example: when a process tries to execute data (text)
- Privileged instruction
- Data misuse
- Operating system intervention; for example: to resolve a deadlock

- Parent terminates so child processes terminate (**cascading termination**)
- Parent request

4. Two-State Process Management Model

The Operating System's principal responsibility is in controlling the execution of processes. This includes determining the interleaving pattern for execution and allocation of resources to processes. The first step in designing an OS—a program to control processes—is to describe the behaviour that we would like each process to exhibit. The simplest model is based on the fact that a process is either being executed by a processor or it is not. Thus, a process may be considered to be in one of two states, RUNNING or NOT RUNNING, (seeFigure 4.1).

When the operating system creates a *new* process, that process is initially labelled as NOT RUNNING, and is placed into a queue in the system in the NOT RUNNING state. The process (or some portion of it) then exists in main memory, and it waits in the queue for an opportunity to be executed. After some period of time, the currently RUNNING process will be interrupted, and moved from the RUNNING state to the NOT RUNNING state, making the processor available for a different process. The **dispatch** portion of the OS will then select, from the queue of NOT RUNNING processes, one of the waiting processes to transfer to the processor. The chosen process is then relabelled from a NOT RUNNING state to a RUNNING state, and its execution is either begun if it is a new process, or is resumed if it is a process which was interrupted at an earlier time.

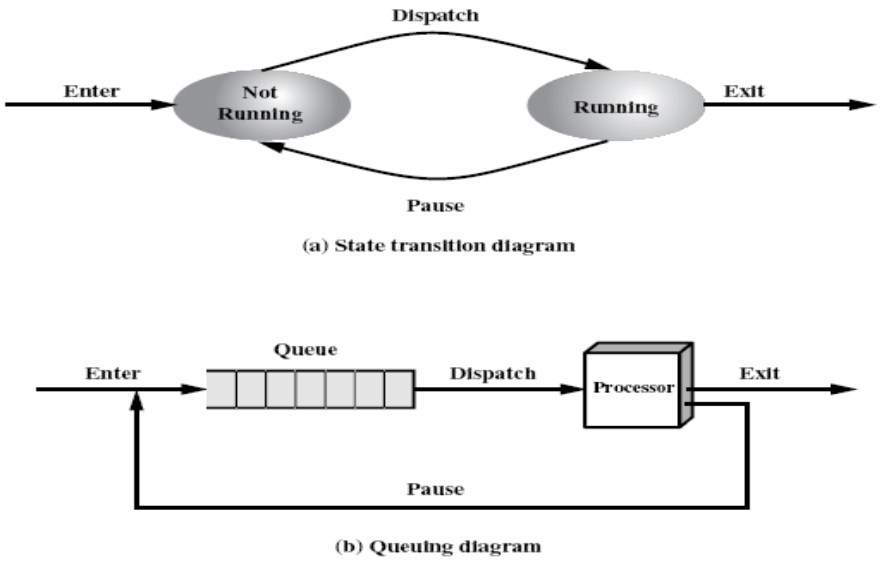

Figure 4.1: Two-State Process Management Model showing (a) transitions between states, and (b) the associated queuing diagram.

From this model we can identify some design elements of the OS:

- The need to represent, and keep track of each process.
- The state of a process.
- The queuing of NON RUNNING processes

5. Introduction to Scheduling

As stated earlier, in a multiprogramming computer with a single processor, several processes will be competing for use of that processor. It should be obvious from the Two-State Process Management Model just discussed that, at any instant, only one process will be running while the others will be queuing, waiting for the processor or on some other wait condition. This state we termed NOT RUNNING. It is also known as the READY state. The OS has the task of determining the optimum sequence and timing of assigning READY processes to the processor. This activity is called **scheduling**. The method by which processes are ordered in the READY queue, and the timing of when a process is chosen to be dispatched to the processor, is determined by the **scheduling**

policy of the OS. To aid explanation, consider the following everyday example, where a **priority-based** scheduling algorithm is used:

- Imagine you are constructing a kit that involves assembling parts by following a list of instructions.
- The first step is to join Part A to Part B with a 2-inch screw, and as you complete it you check it off your list. Next is step 2 and then step 3.
- However, just as you have just completed step 3, the doorbell rings. So you stop what you are doing, check off step 3 on the list and answer the door. Answering the door is a more immediate task than assembly of the kit.
- When finished at the door, you return to the kit, and by referring to the list, you see that the next step is 4 and so you continue until the kit is completed.

In OS terminology, you played the part of the CPU, the assembly of the kit was job A and answering the doorbell was job B. The doorbell was an interrupt and answering the door was a higher priority program. When you were interrupted, you performed a context switch by marking off step 3 as the last completed instruction and stopping work. Answering the door was the execution of program B. The end of both jobs was the termination. The example above can be summarised as a set of instructions as follows:

- Get the input for job A
- Identify resources
- Execute the process
- Interrupt
- Context switch to job B
- Get the input for job B
- Terminate job B
- Context switch to job A
- Resume executing process
- Terminate job A

Three levels of scheduling *may* exist in any OS (although some modern OSes only implement two of them). The three levels are as follows:

- **High Level Scheduling** (HLS): deals with adding new jobs to system.
- **Medium Level Scheduling** (MLS): concerned with the decision to temporarily remove a process from the system or to re-introduce a process.
- **Low Level Scheduling** (LLS): handles the decision of which READY process is to be assigned to the processor. This level is often called the **dispatcher** and refers to the actual kernel activity of transferring control to the selected process.

As explained earlier, a common trait found among processes associated with most computer programs, is that they alternate between CPU cycles and I/O cycles. Scheduling takes advantage of this property. For the portion of the time required for CPU cycles, the process is assigned the RUNNING state. However, as you may already have deduced, the Two-State Process Management Model does not possess a state which describes a process in I/O cycles; i.e. when the process does not require the use of the processor. A new model —the Three-State Process Management Model—overcomes this short-coming (as we will see shortly) by introducing a third state termed the BLOCKED state. The times spent by the process in the RUNNING and BLOCKED states are of different duration.

One interesting empirical observation regarding processes relates to the distribution of CPU cycle times for a sample of processes in a system. Figure 5.1 shows that the the distribution of CPU cycle times for processes in a system follows a Poisson distribution. This graph shows that a greater proportion of processes possess short CPU cycles and a lesser proportion of processes possess long CPU cycles. When the computer system is over-loaded, the MLS will swap jobs out of memory to allow other jobs to be completed faster.

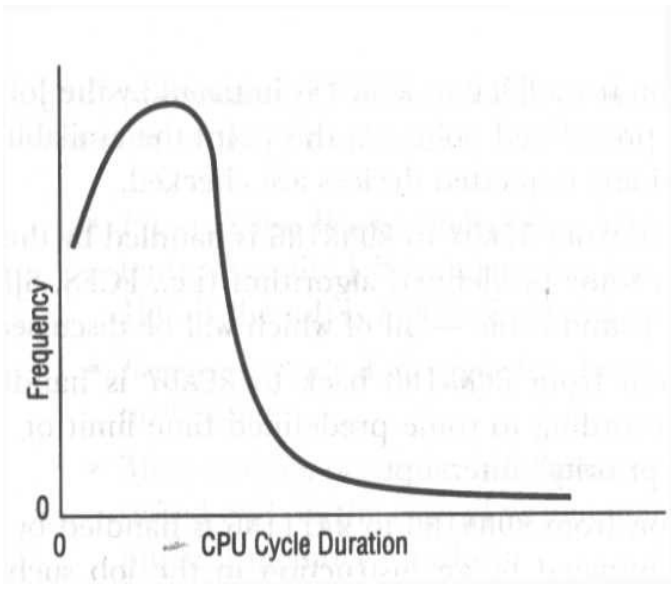

Figure 5.1: Observation of the CPU cycle lengths of a sample of processes shows that they follow a Poisson Distribution.

6. Three-State Process Management Model.

Although the Two-State Process Management Model is a perfectly valid design for an operating system, the absence of a BLOCKED state means that the processor lies idle when the active process changes from CPU cycles to I/O cycles. This design does not make efficient use of the processor. As stated in the previous section, the Three-State Process Management Model[1] is designed to overcome this problem, by introducing a new state called the BLOCKED state. This state describes any process which is waiting for an I/O event to take place. In this case, an I/O event can mean the use of some device or a signal from another process. Figure 6.1 shows the diagram of the Three-State Process Management Model. The ellipses represent process states, and the arrows represent transitions between states in the direction indicated.

1 Some authors refer to the Three-State Process Management Model as a Five-State Model, where the two additional states are NEW and EXIT (See Figure 6.1). However, we will ignore these, as a process does not actually exist in the system when in the NEW state, and is no longer active in the system when in the EXIT state.

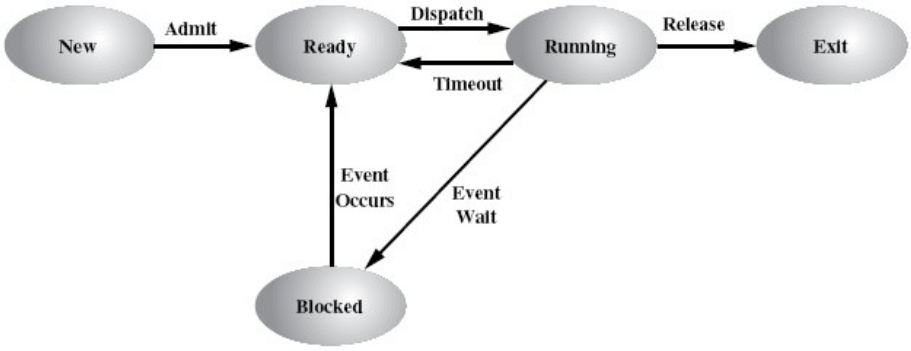

Figure 6.1: The Three-State Process Management Model. The ellipses represent process STATES, and the arrows represent transitions between states in the direction indicated.

In summary, the three states in this model are:

- **RUNNING:** The process that is currently being executed.
- **READY:** A process that is queuing prepared to execute when given the opportunity.
- **BLOCKED:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.

At any instant, a process is in *one* and *only one* of the three states. For a single processor computer, *only one* process can be in the RUNNING state at any one instant. There can be *many* processes in the READY and BLOCKED states, and each of these states will have an associated queue for processes. Points to note are:

- processes entering the system must go initially into the READY state
- Processes can only enter the RUNNING state via the READY state.
- Processes *normally* leave the system from the RUNNING state.
- For each of the three states, the process occupies space in main memory.

While the reason for most transitions from one state to another might be obvious, some may not be so clear. Consider the following transitions:

6.1. RUNNING → READY

The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution; i.e. time-out occurs. Other reasons can be the imposition of priority levels as determined by the scheduling policy used for the LLS, and the arrival of a higher priority process into the READY state.

6.2. RUNNING → BLOCKED

A process is put into the BLOCKED state if it requests something for which it must wait. A request to the OS is usually in the form of a system call, (i.e. a call from the running process to a function that is part of the OS code). For example, requesting a file from disk or a saving a section of code or data from memory to a file on disk. Figure 6.2 shows examples of different queues of BLOCKED states which may exist in a system.

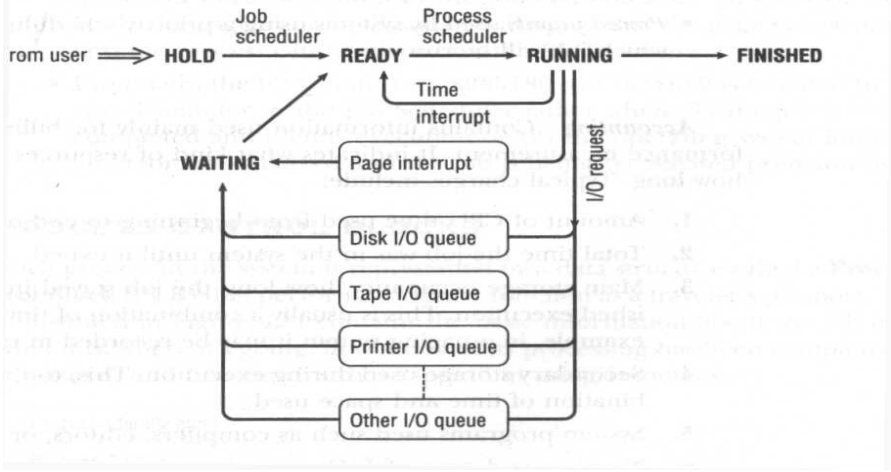

Figure 6.2: Examples of different queues of BLOCKED states.

7. Five-State Process Management Model

While the three state model is sufficient to describe the behaviour of processes with the given events, we have to extend the model to allow for other possible events, and for more sophisticated design. In particular, the use of a portion of the hard disk to emulate main memory—so called **virtual memory**—requires additional states to describe the state of processes which are **suspended**[2] from main memory, and placed in virtual memory (on disk). Of course, such processes can, at a future time, be **resumed** by being transferred back into main memory. The Medium Level Scheduler controls these events.

Figure 7.1 shows the Five-State Process Management Model[3]. The ellipses represent process STATES, and the arrows represent transitions between states in the direction indicated. As can be seen from this figure, a process can be suspended from the RUNNING, READY or BLOCKED state, giving rise to two other states, namely, READY SUSPEND and BLOCKED SUSPEND. Note

2 In UNIX/Linux, the term **stopped** is often used instead of the term **suspended**. This should not be confused with the concept of a **terminated** process, the latter which has finished execution entirely.

3 Some authors refer to the Five-State Process Management Model as the Five-State Model with Suspend States; the two extra states being “New” and “Exit” - see Figure 7.1.

that a RUNNING process that is suspended becomes READY SUSPEND, and a BLOCKED process that is suspended becomes BLOCKED SUSPEND.

A process can be suspended for a number of reasons; the most significant of which arises from the process being swapped out of memory by the memory management system in order to free memory for other processes. Other common reasons for a process being suspended are when one suspends execution while debugging a program, or when the system is monitoring processes.

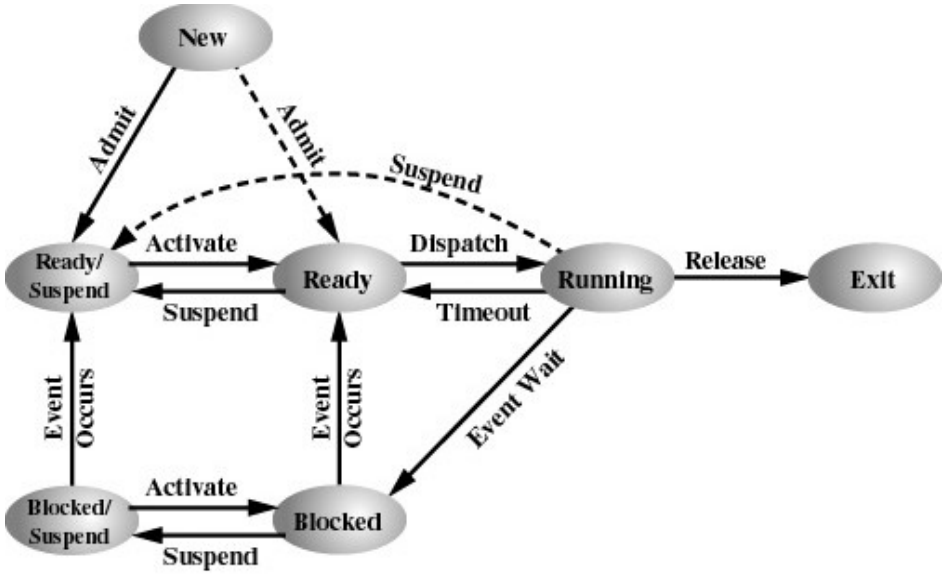

Figure 7.1: The Five-State Process Management Model. The ellipses represent process STATES, and the arrows represent transitions between states in the direction indicated.

For the Five-State Process Management Model, consider the following transitions described in the next sections.

7.1. BLOCKED → BLOCKED SUSPEND

If a process in the RUNNING state requires more memory, then at least one BLOCKED process can be swapped out of memory onto disk. The transition

can also be made for the BLOCKED process if there are READY processes available, and the OS determines that the READY process that it would like to dispatch requires more main memory to maintain adequate performance.

7.2. BLOCKED SUSPEND → READY SUSPEND

A process in the BLOCKED SUSPEND state is moved to the READY SUSPEND state when the event for which it has been waiting occurs. Note that this requires that the state information concerning suspended processes be accessible to the OS.

7.3. READY SUSPEND → READY

When there are no READY processes in main memory, the OS will need to bring one in to continue execution. In addition, it might be the case that a process in the READY SUSPEND state has higher priority than any of the processes in the READY state. In that case, the OS designer may dictate that it is more important to get at the higher priority process than to minimise swapping.

7.4. READY → READY SUSPEND

Normally, the OS would be designed so that the preference would be to suspend a BLOCKED process rather than a READY one. This is because the READY process can be executed as soon as the CPU becomes available for it, whereas the BLOCKED process is taking up main memory space and cannot be executed since it is waiting on some other event to occur. However, it may be necessary to suspend a READY process if that is the only way to free a sufficiently large block of main memory. Finally, the OS may choose to suspend a lower-priority READY process rather than a higher-priority BLOCKED process if it believes that the BLOCKED process will be ready soon.

8. Process Description and Control

8.1. Process Control Block

By this stage, you may have already asked yourself the question: how does the operating system keep track of all the processes in the system? The answer to this question is quite simple! Each process in the system is represented by a data structure called a **Process Control Block** (PCB), or **Process Descriptor** in Linux, which performs the same function as a traveller's passport. The PCB contains the basic information about the job including:

- What it is
- Where it is going
- How much of its processing has been completed
- Where it is stored
- How much it has “spent” in using resources

Figure 8.1 gives some common data held in a PCB. They are described in greater detail as follows.

- **Process Identification**: Each process is uniquely identified by the user’s identification and a pointer connecting it to its descriptor.
- **Process Status**:This indicates the current status of the process; i.e. READY, RUNNING, BLOCKED, READY SUSPEND, BLOCKED SUSPEND.
- **Process State**: This contains all of the information needed to indicate the current state of the job.
- **Accounting**: This contains information used mainly for billing purposes and for performance measurement. It indicates what kind of resources the process has used and for how long.

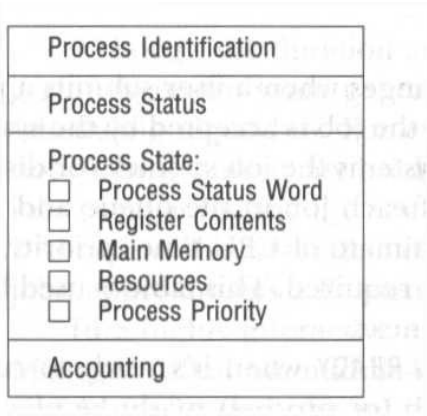

Figure 8.1: Some common elements contained in a Process Control Block (PCB).

8.2. Processor Modes

Contemporary processors incorporate a mode bit to define the execution capability of a program in the processor. This bit can be set to **kernel mode** or **user mode**. The kernel mode is also commonly referred to as supervisor mode, protected mode or monitor mode. In kernel mode, the processor can execute every instruction in its hardware repertoire, whereas in user mode, it can only execute a subset of the instructions. Instructions that can be execute only in kernel mode are called **kernel**, **privileged** or **protected** instructions to distinguish them from the user mode instructions. For example, I/O instructions are privileged instructions. So, if an application program executes in user mode, it cannot perform its own I/O. Instead, it must to request the OS to perform I/O on its behalf.

The system may logically extend the mode bit to define areas of memory to be used when the processor is in kernel mode versus user mode – see Figure 8.2. If the mode bit is set to kernel mode, the process executing in the processor can access either the kernel or user partition of the memory. However, if user mode is set, the process can reference only the user memory space. We fre-

quently refer to two classes of memory **user space** and **system space** (or kernel, supervisor or protected space).

In general, the mode bit extends the operating system's **protection rights**. The mode bit is set by the user mode **trap instruction**, also called a **supervisor call instruction**. This instruction sets the mode bit, and branches to a fixed location in the system space. Since only system code is loaded in the system space, only system code can be invoked via a trap. When the OS has completed the supervisor call, it resets the mode bit to user mode prior to the return.

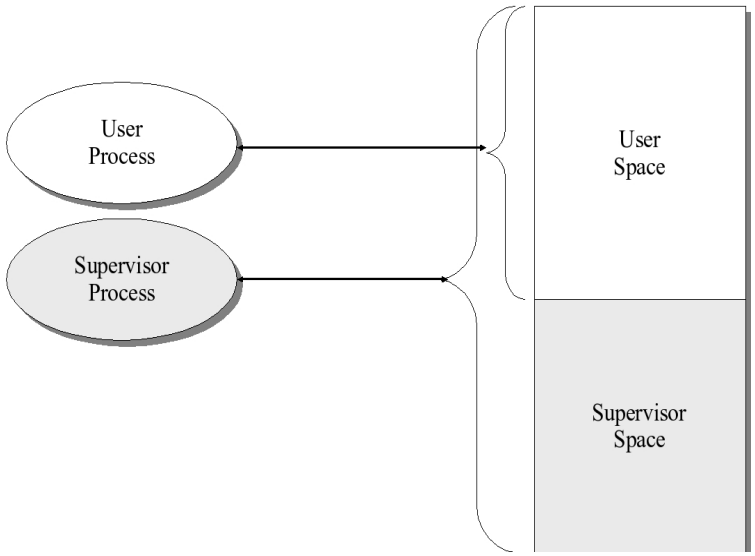

Figure 8.2: Supervisor (Kernel) and User Memory Space.

8.3. The Kernel Concept

The parts of the OS critical to its correct operation execute in kernel mode, while other software (such as generic system software) and all application programs execute in user mode. This fundamental distinction is usually the irrefutable distinction between the operating system and other system software. As you may have already deduced, the part of the system executing in kernel or

supervisor state is called the **kernel**, or nucleus, of the operating system. The kernel operates as trusted software, meaning that when it was designed and implemented, it was intended to implement protection mechanisms that could not be covertly changed through the actions of untrusted software executing in user space. Extensions to the OS execute in user mode, so the OS does not rely on the correctness of those parts of the system software for correct operation of the OS. Hence, a fundamental design decision for any function to be incorporated into the OS is whether it *needs* to be implemented in the kernel. If it is implemented in the kernel, it will execute in kernel (supervisor) space, and have access to other parts of the kernel. It will also be trusted software by the other parts of the kernel. If the function is implemented to execute in user mode, it will have no access to kernel data structures. However, the advantage is that it will normally require very limited effort to invoke the function.

While kernel-implemented functions may be easy to implemented, the trap mechanism and authentication at the time of the call are usually relatively expensive. The kernel code runs fast, but there is a *large performance overhead in the actual call*. This is a subtle, but important point.

8.4. Requesting System Services

There are two techniques by which a program executing in user mode can request the kernel's services:

- System call
- Message passing

Generally, operating systems are designed with one or the other of these two facilities, but not both. Figure 8.3 summarises the differences between the system call and message passing techniques. First, assume that a user process wishes to invoke a particular target system functions (represented as an annotated shaded rectangle in the figure). For the **system call** approach, the user process uses the trap instruction (described in previous sections). The idea is that the system call should appear to be an ordinary procedure call to the application program; the OS provides a library of user functions with names corresponding to each actual system call. Each of these **stub functions** contains a trap to the OS function. When the application program calls the stub, it executes the trap instruction, which switches the CPU to kernel mode, and then branches, (indirectly through an OS table) to the entry point of the function which is to be invoked. When the function completes, it switches the processor to user mode and then returns control to the user process; thus simulating a normal procedure return.

In the **message passing** approach, the user process constructs a message, A, that describes the desired service. The it users a trusted **send** function to pass the message to a trusted OS process. The **send** function serves the same purpose as the trap; that is, it carefully checks the message, switches the processor to kernel mode, and then delivers the message to a process that implements the target functions. Meanwhile, the user process waits for the result of the service request with a message **`receive`** operation. When the OS process

completes the operation, it sends a message (B in the figure) back to the user process.

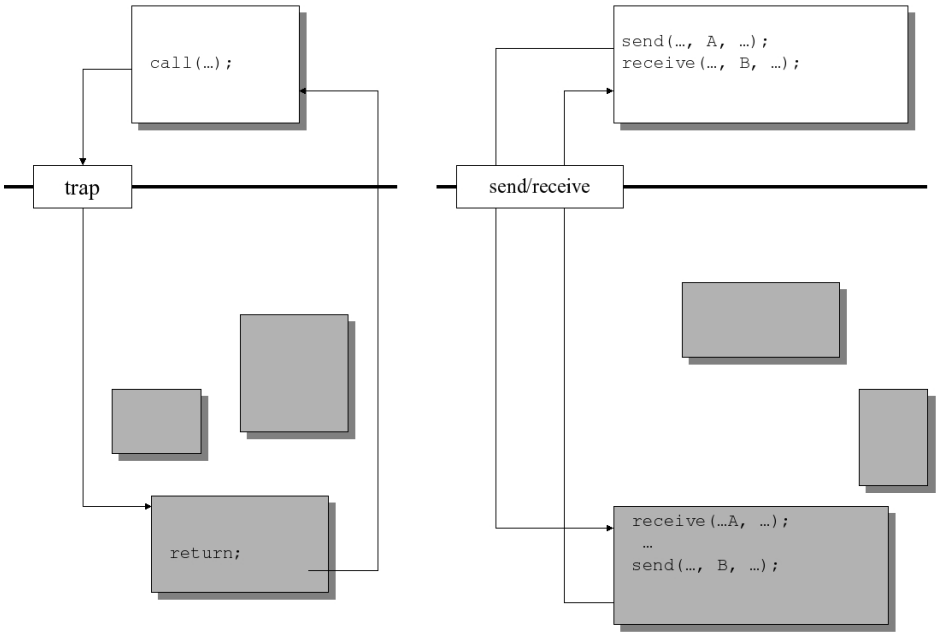

Figure 8.3: Procedure Call (Left) and Message Passing (Right) Operating Systems.

The distinction between the two approaches has important consequences regarding the relative independence of the OS behaviour, from the application process behaviour, and the resulting performance. As a rule of thumb, operating system based on a system call interface can be made more efficient than those requiring messages to be exchanged between distinct processes. This is the case, eventhough the system call must be implemented with a trap instruction; that is, even though the trap is relatively expensive to perform (as we stated earlier), it is more efficient than the message passing approach, where there are generally higher costs associated with process multiplexing, message formation and message copying.

The system call approach has the interesting property that three is not necessarily any OS process! Instead, a process executing in user mode changes to kernel mode when it is executing kernel code, and switches back to user mode when it returns from the OS call. If, on the other hand, the OS is designed as a set of separate processes, it is usually easier to design it so that it gets control of the machine in special situations, than if the kernel is simply a collection of functions executed by users processes in kernel mode. Even procedure-based operating system usually find it necessary to include at least a few “system processes”—called daemons in UNIX—to handle situation whereby the machine is otherwise idle, scheduling, handling the network, etc.

9. Case Study: The UNIX Operating

Now let us examine a specific example of an operating system. We pick UNIX because it is a system that runs on a wider variety of computers than any other operating system. Strictly speaking, UNIX is not one operating system. There are many **flavours** (versions) and clones of UNIX, including AIX, BSD, HP-UX, Linux, MINIX, SCO UNIX, System V, Solaris and various others. This sometimes presents problems. Fortunately, the fundamental principles and system calls are very similar for all of them. This is by design. Also, the general implementation strategies, algorithms and data structures are also similar. But, of course, there are some differences, which we will not go into here. Later, in this section, we will examine UNIX System V Release 4, which is one of the most common flavours of UNIX.

But first, since UNIX has a long and turbulent history, we start our discussion there.

9.1. Brief History of UNIX

It wasn't until the early 1960s that the first time-sharing operating systems began to emerge. (Up until that time, batch systems were used). The first such

system was developed at Dartmouth College and MIT. This system, called the **Compatible Time-sharing System** (**CTSS**), was an enormous success among the scientific community, who were the main users of computers. By 1965, researchers at MIT joined forces with Bell Labs and General Electric (then a computer vendor) and began to design a second generation time-sharing system called **MULTICS** (**MULTiplexed Information and Computing Service**). Although, Bell Labs was one of the founding partners in the project, it later withdrew, which left one of the the Bell Lab researchers, Ken Thompson, looking around for something to do! He eventually decided to write a stripped-down MULTICS by himself, on a discarded PDP-7 minicomputer. Despite the (relatively) small size of the PDP-7, Thompson's system actually worked, which facilitated him in further developing the system. Consequently one of his colleagues at Bell Labs, Brian Kernighan, somewhat jokingly call this new system, UNICS, for **Uniplexed Information and Computing Service**. Despite puns about "EUNUCHS" being a castrated MULTICS, the name stuck, although, the spelling changed to UNIX. Thompson's work impressed his colleagues at Bell Labs so much, that he was soon joined by Dennis Ritchie, and later by his entire department! The first major development in UNIX then concerned the *porting*[4] of this operating system to newer and more modern hardware. The second major development concerned the language in which UNIX was written. Since the original system was written in the language of **assembler**, which is hardware-dependent, this created difficulties porting the system to different types of computers. It also made maintenance of the code difficult – assembler is a cumbersome language in which to program. Therefore, Thompson decided to rewrite UNIX in a high-level language of his own design, called *"B"*. However, this was not particularly successful, due to the limited nature of the *B* language. Later, Ritchie designed a successor to *B*, called *"C"* (of course!), accompanied by an excellent compiler, which facilitated development. This led to the rewriting of UNIX in *C*. As you may be aware, this programming language has dominated system programming ever since; successors to *C* being, *C++* and *Java* (which is written in C++).

4 "Porting" means the rewriting of all or part of the code, so that the system runs on the different hardware.

In 1974, Ritchie and Thompson published a landmark paper about UNIX, called *"The UNIX Timesharing System"*[5]. The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee. UNIX quickly replaced the (dreadful) operating system that ran on the PDP-11, which was the dominant machine used in universities. The fact that this machine was widely used across the sector was a stroke of luck. For, UNIX was supplied with its source code, which allowed university researchers and students to tinker with the code, and improve on it. One of these universities, which acquired UNIX Version 6 early on was the University of California, Berkeley. Berkeley was able to modify the system substantially. Its first version was called 1BSD (First Berkeley Software Distribution). This has developed to this day, where we have 4.4BSD.

In 1984, the AT&T monopoly was broken up with the US government, and the company was then legally free to set up as a computer subsidiary. Within a short time, after some unsuccessful attempts, AT&T released its a commercial UNIX product[6], UNIX System V.

This meant that by the late 1980s, two different, and somewhat incompatible, versions of UNIX were the two *main* flavours in widespread use: 4.3BSD and System V Release 3. By this stage there were many other flavours also in use – see Figure 9.1.

5 "The UNIX Timesharing System", Commun. of the ACM, Vol. 17, D. Ritchie & K. Thompson
6 Systems I – III were not well received. Nobody knows what happened to System IV.

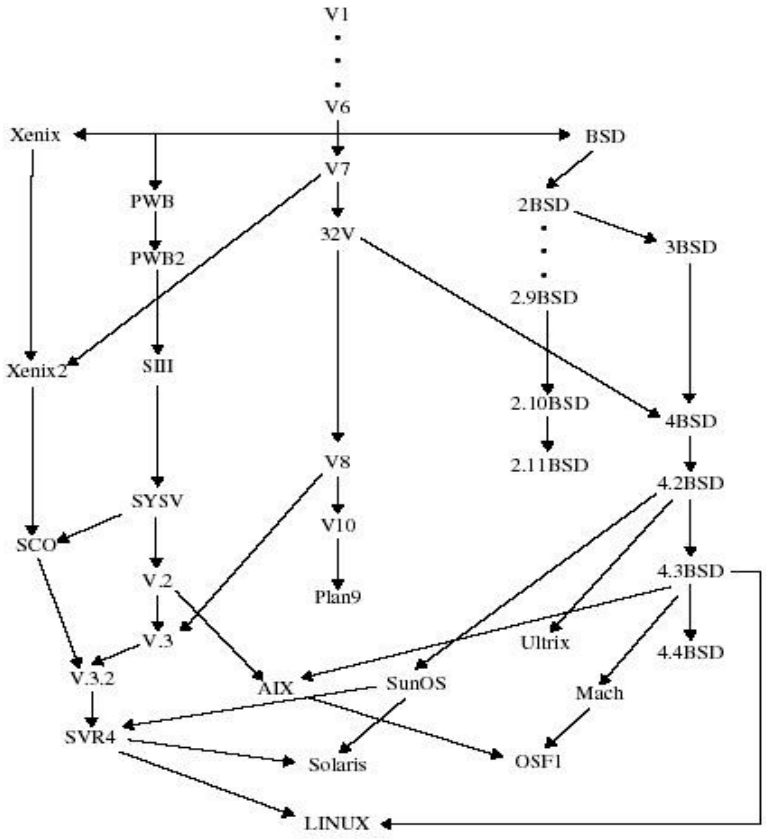

Figure 9.1: The evolution of UNIX from Version 1 to the present day. This shows the forking of UNIX into many flavours. Note the System V and BSD branches. Ultimately, these led to Linux.

The first serious attempt to reconcile these two main flavours of UNIX was initiated under the auspices of the IEEE Standards Body. This resulted in the POSIX project. **POSIX** stands for **Portable Operating Systems Interface** (the IX at the end was added to make the name “UNIXish”). This standard defines a set of library procedures that every *POSIX-compliant* UNIX system must supply. Most of these procedure invoke a system call, but a few can be implemented outside the kernel. The idea of POSIX is that a software vendor who writes a program that uses only the procedures defined by the standard knows that this program will run on every compliant UNIX system. Today, both 4.4BSD and System V Release 4 are (generally) POSIX-compliant.

9.2. Features of the UNIX Operating System

The features of UNIX can be summarised as follows:

- Multi-user
  - more than one user can use the machine at a time
  - originally supported via terminals (serial or network connection)
- Multi-tasking
  - more than one program can be run at a time
- Hierarchical directory structure
  - to support the organisation and maintenance of files
- Portability
  - only the kernel (<10%) written in assembler
- Tools for program development
  - a wide range of support tools (debuggers, compilers)

The *three* main aspects of the operating system are:

- kernel:
  - schedules tasks
  - manages data/file access and storage
  - enforces security mechanisms
  - performs all hardware access
- shell:
  - presents each user with a prompt
  - interprets commands types by a user
  - executes user commands
  - supports a custom environment for each user
- utilities:
  - file management (**`rm, cat, more, ls, rmdir, mkdir`**)
  - user management (**`passwd, chmod, chgrp`**)
  - process management (**`kill, ps`**)

- printing (**`lpr, lpq`**)

9.3. UNIX Command Structure

Commands in UNIX all have the same structure. The number of commands is kept small, but each command is extended by using option switches. Options are preceded by the "-" symbol. Figure 9.2 illustrates this further.

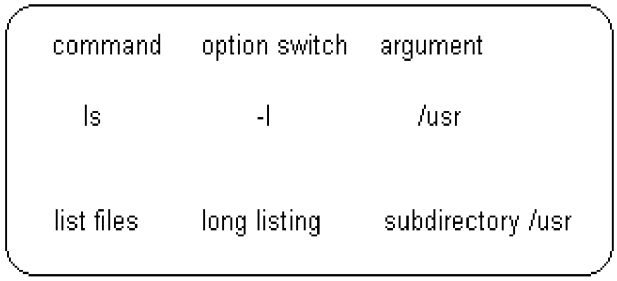

Figure 9.2: UNIX command structure consists of generally short command names, with zero or more options and zero or more arguments.

9.4. The UNIX Shell

As stated above, the UNIX user command line interface is known as the **shell**, because of the structure of traditional UNIX, whereby the interface surrounds the operating system like a shell – See Figure 9.3. The shell which runs when a user logs on, enables users to run application programs. There are eight features of the UNIX shell:

1. Interactive: The shell presents a prompt and waits for the user to enter a command. After the return key is pressed, the shell processes the command and when the command is finished, the shell re-displays the prompt. This process continues until the user exits the shell, by typing exit or by pressing CTRL-d, at which time the user is logged out of the UNIX host.

2. Runs programs in the background. Non-interactive tasks which do not require keyboard input or display output can be run in the background as a separate task. The user continues working with other inter-active programs. Examples of this are printing or sorting files.
3. Input/Output redirection: Programs designed to use the standard input device (keyboard) and standard output device (display) can have their input and output devices redirected to other devices. For example, a program which generally reads from the keyboard can be redirected to read from a file instead. A program which writes its output to the display can be instructed to redirect its output to the printer or a file.
4. Programs can be chained or connected together via pipes: The output of one program can be fed directly into another program by connecting the two programs via a pipe. This allows a user to create powerful new commands by chaining existing commands together.
5. Wild-card characters are supported in file-names: The handling of files is simplified by using wild-card characters to match files which match particular patterns. Common operations can thus be performed on a group of common files using a single command.
6. Script files simplify repeating command sequences: A number of commonly used commands can be stored in a file, which when executed, runs each command as though it has been typed from the command line. A sequence of commands can be executed by executing the file which contains the command. This simplifies repetitious commands.
7. Environment variables: The user can customise and control the behaviour of the shell by using special variables that the shell supports. The variables can also be used by application programs and shell script files to control their behaviour. An example of a shell variable is the prompt string used to display the shell prompt sign ($).
8. A macro language for building shell scripts: The shell supports a simple language definition for creating shell script files. These can be used to generate very complex command sequences.

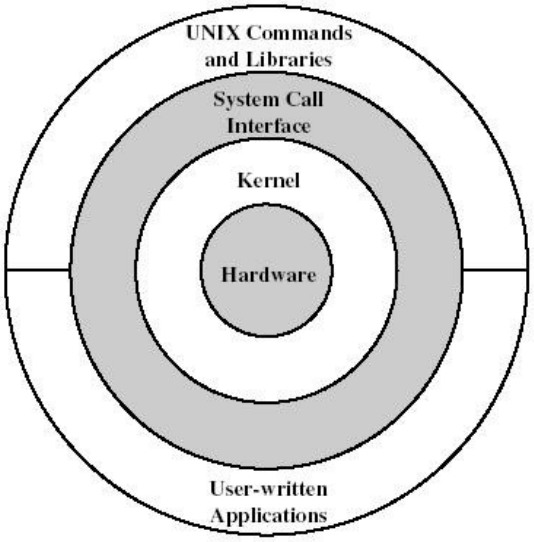

Figure 9.3: UNIX operating system structure.

9.5. UNIX System V Release 4

UNIX System V makes use of a simple but powerful process facility that is highly visible to the user. UNIX follows the model shown in Figure 9.4, in which most of the operating system executes within the environment of a user process. Thus, two modes, **user mode** and **kernel mode**, are required. UNIX uses two categories of processes: **system processes** and **user processes**. System processes run in kernel mode and execute operating system code to perform administrative and housekeeping functions, such as allocation of memory and process swapping. User processes operate in user mode to execute user programs and utilities, and switch to kernel mode to execute instructions belong to the kernel. A user process enters kernel mode by issuing a system call.

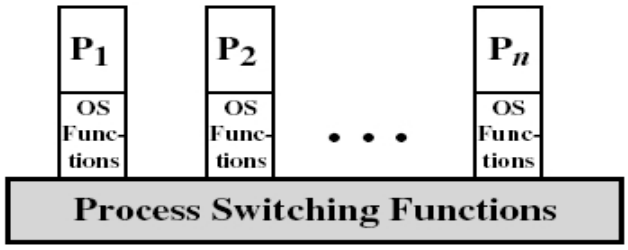

Figure 9.4: In UNIX System V Release 4, the OS executes within the environment of the user process.

9.5.1. Process States

A total of *nine* process states are recognised by the UNIX operating system; these are explained in Table 9.1 below. The corresponding state transition diagram is shown in Figure 9.5. This figure is similar to Figure 7.1, with the two UNIX sleeping states corresponding to the two blocked states. The differences are summarised as follows:

- UNIX employs two Running states to indicate whether the process is executing in user mode or kernel mode.
- A distinction is made between the two states: (Ready to Run, in Memory) and (Pre-empted). These are essentially the same state, as indicated by the dotted line joining them. The distinction is made to emphasise the way in which the pre-empted state is entered. When a process is running in kernel mode (as a result of a supervisor call, clock interrupt, or I/O interrupt), there will come a time when the kernel has completed its work and is ready to return control to the user program. At this point, the kernel may decide to pre-empt the current process in favour of one that is ready and of higher priority. In that case, the current process moves to the pre-empted state. However, for purposes of dispatching, those processes in the pre-empted state and those in the Ready to Run, in Memory state form one queue.

Pre-emption can only occur when a process is about to move from kernel mode to user mode. While a process is running in kernel mode, it may not be pre-empted. This makes UNIX unsuitable for real-time processing.

State	Description
User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Pre-empted	Process is returning from kernel to user mode, but the kernel pre-empts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect (or mourn its death!)

Table 9.1: The nine process states of UNIX System V Release 4.

Two processes are unique in UNIX. Process 0 is a special process that is created when the system boots; in effect, it is predefined as a data structure loaded at boot time. It is the **swapper** process[7]. In addition, process 0 spawns process 1, referred to as the **init** process; all other processes in the system have process 1 as an ancestor. When a new interactive user logs onto the system, it is process 1 that creates a user process for that user. Subsequently, the user process can create child processes in a branching tree, so that any particular application can consist of a number of related processes.

7 In fact, the swapper process is not a true process; it is a pseudo-process.

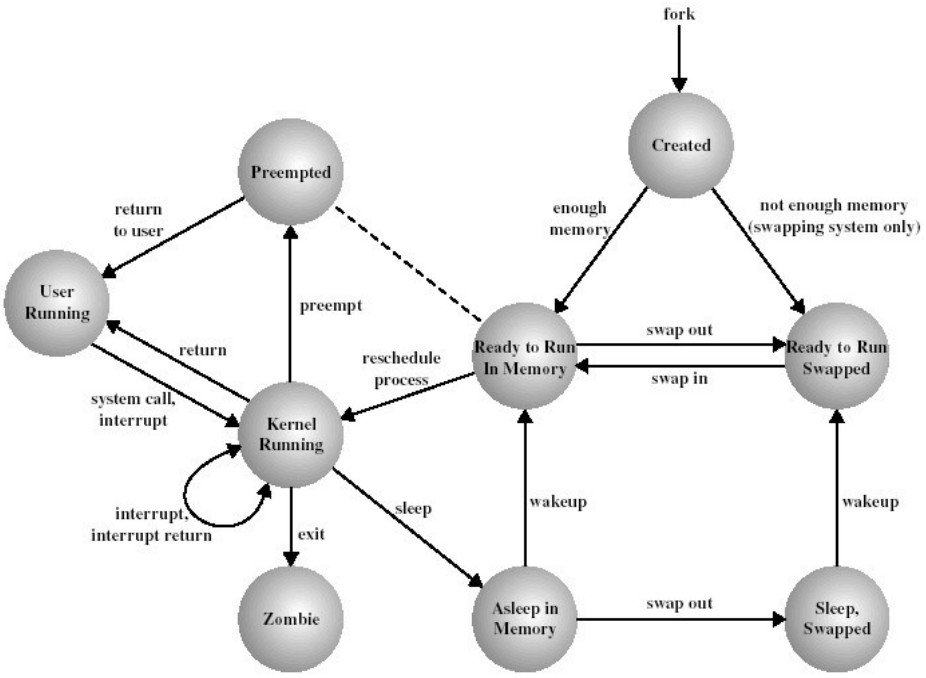

Figure 9.5: UNIX System V Release 4 Process State Model.

9.5.2. Process Control

Process creation in UNIX is made by means of the kernel system call, **`fork()`**. When a process issues a **`fork`** request, the operating system performs the following functions:

1. It allocates a slot in the process table for the new process.
2. It assigns a unique **process ID** to the **child process**.
3. It makes a copy of the **process image** of the parent, with the exception of any shared memory.
4. It increments counters for any files owned by the parent, to reflect that an additional process now also owns those files.
5. It assigns the child process to a Ready to Run state.
6. It returns the ID number of the child to the parent process, and a 0 value to the child process.

All of this work is accomplished in kernel mode in the parent process. If a different process is to run as the child process, then the system invokes the **`exec()`** system call, which overwrites the duplicate (child) process with the one passed as the argument to **`exec()`**. When the kernel has completed these functions it can do one of the following, as part of the dispatcher routine:

1. Stay in the parent process. Control returns to user mode at the point of the fork call of the parent.
2. Transfer control to the child process. The child process begins executing at the same point in the code as the parent, namely at the return from the fork call.
3. Transfer control to another process. Both parent and child are left in the Ready to Run state.

It is perhaps difficult to visualise this method of process creation because both parent and child are executing the same passage of code. The difference is this: when the return from the fork occurs, the return parameter is tested. If the value is zero, then this is the child process, and a branch can be executed to the appropriate user program to continue execution. If the value is non-zero, then this is the parent process, and the main line of execution can continue.

10. Exercises

1. Describe what the concept of a **process** is, and how a process is different to a **program**.
2. What is meant by the statements that a process is:
   - in the READY state?
   - in the BLOCKED state?
3. Why is there only one process in the RUNNING state at any one instant?
4. Draw a process state transition diagram using five states and explain the interpretation of each transition.
5. Distinguish between **user mode** and **kernel mode** in an Operating System.

6. In UNIX, describe the six tasks which the Operating System performs to deal with a **`fork()`** request issued by a process. What are the three possible choices the kernel has, once those tasks are completed?
7. It was stated in this Lecture, that UNIX is unsuitable for real-time applications because a process executing in kernel mode may not be pre-empted. Elaborate.