CHAPTER

3 Software Platforms

Learning Objectives

After reading this chapter, you should be able to:

- Describe the core components of software platforms.
- Discuss the different views of operating systems.
- Explain the layered structure of a typical operating system and the components of its individual layers.

3.1 Introduction

In Chapter 2 we defined two platforms of a computer system, and described some essential resources that are used to build the hardware platform. In this chapter, we discuss the other part of the system, namely the software platform. An operating system, together with the associated utility programs, provides a software platform that seems independent of the underlying hardware platform. Computer users interact with the software platform to do their work. Expert professionals design, develop, and maintain operating systems and utilities.

» The software platform sits on top of and controls the hardware platform, and simulates the environment of a software virtual machine.

While the design and development of a new operating system (that is, setting up a new software platform) is no doubt a challenge, it is no different from development of a large software product. The operating system software of a computer consists of a number of component subsystems. Each subsystem manages a specific hardware resource, and/or provides some services to the users. In this chapter, we present a typical layered structure of the operating system software, and discuss briefly the subsystems within each layer. This chapter lays the foundation for the subsequent in-depth discussion of subsystems in later chapters of this book.

3.2 Different Views of Operating Systems

» Operating systems are of different types and may be compared to the governments of different states. In principle, all governments strive to extend the best possible services to their citizens while also seeking to effectively manage and protect their resources. However, the policies they may pursue to implement these objectives and the manner of implementation may differ from one government to another. In other words, a government functions as the overall controlling authority in its state. Likewise, an operating system is a software program that controls all the hardware resources of a computer and provides servies to applications.

An operating system drives a "difficult-to-use" hardware platform, and it creates a user-friendly environment to enable users—who may not be knowledgeable about computer hardware—conveniently develop and run their applications. In other words, an operating system simulates an "easy-to-use" software platform for computer users, and also controls the usage of all the hardware resources, optimizes their performance, and resolves any access conflicts that may arise on the resources. Thus, the operating system has two primary responsibilities:

1. It acts as a "service provider" to its users, providing a set of services to accomplish their tasks.
2. It acts as an effective "resource manager", managing all resources in the computer: allocating (and reclaiming) resources to (and from) applications.

An operating system may be viewed from a few different perspectives. Each view is defined by the way a particular user operates the system in practice. Different views lay emphasis on different aspects of the system. We describe these views below:

- **Programmer's view**: Application developers view an operating system as an extension of the basic machine instruction set. The system implements a set of higher-level operations that are available through system calls, more specifically through APIs in system libraries. Application developers use these operations in their applications to obtain services from the operating system.
- **Logical view**: The logical view describes the internal functional organization of an operating system. It decomposes or breaks the system down into abstract software entities called subsystems and specifies the manner in which these entities must interact with one another to provide guaranteed services to applications, and manage hardware resources. This view defines the software architecture of the system.

» *The architecture of a system encompasses the designs of the system, its subsystems and the connections among them.*

- **Physical view**: The physical view describes the physical structures of the operating system programs and data, and provides a map of the logical software entities into their physical representations. In short, it defines the physical organizations of the various functions and data structures of the operating system.
- **Development view**: This view describes the organization of the source code in a development environment. It helps organize source code files and their compilations, and in maintaining various versions of the source code base. This is popularly known as the build environment.

» *Operating system experts focus mainly on the logical view or architecture of the operating system.*

In this book, we primarily deal with the logical view of operating systems, that is, how system designers and developers view operating systems. An operating system is partitioned into a number of subsystems to render design and development challenges manageable. Each subsystem implements a

well-defined part of the system. Without a well-defined (logical) structure in the system, it is very difficult to develop, integrate, and maintain the system codes.

Figure 3.1 presents the model of a typical (hypothetical) operating system software. Note that no real operating system is structured as depicted in the model, but the use of a simplified model allows us to present the concepts in a more coherent way. The model exhibits essential features of an operating system as if one had to build it from scratch. There are three horizontal layers in the model. In the bottom layer reside three subsystems, one for each type of hardware resource: the CPU manager, the memory manager, and device managers. The software in this layer directly manages the hardware resources. In the top layer, we have external interfaces consisting of system call-, interrupt-, and exception handling routines. In addition, there are utilities that users utilize to manoeuvre the computer system. The middle layer is the head and heart of the (hypothetical) operating system. This layer is composed of three of the most visible subsystems: (1) the process system, (2) the virtual memory, and (3) the virtual file system. These three verticals are not independent: they do interact with one another. For example, processes write data in files. This middle layer also includes a few other subsystems such as bootstrap, shutdown, time management, security, module management, etc. For the sake of simplicity, we have not shown these subsystems in the figure. We introduce them in the following sections.

3.3 Bottom-layer Subsystems

The main hardware devices, namely the CPU, the main memory, and the I/O devices, are managed by this layer. This layer also contains the system bus interface, which we do not discuss in this book.

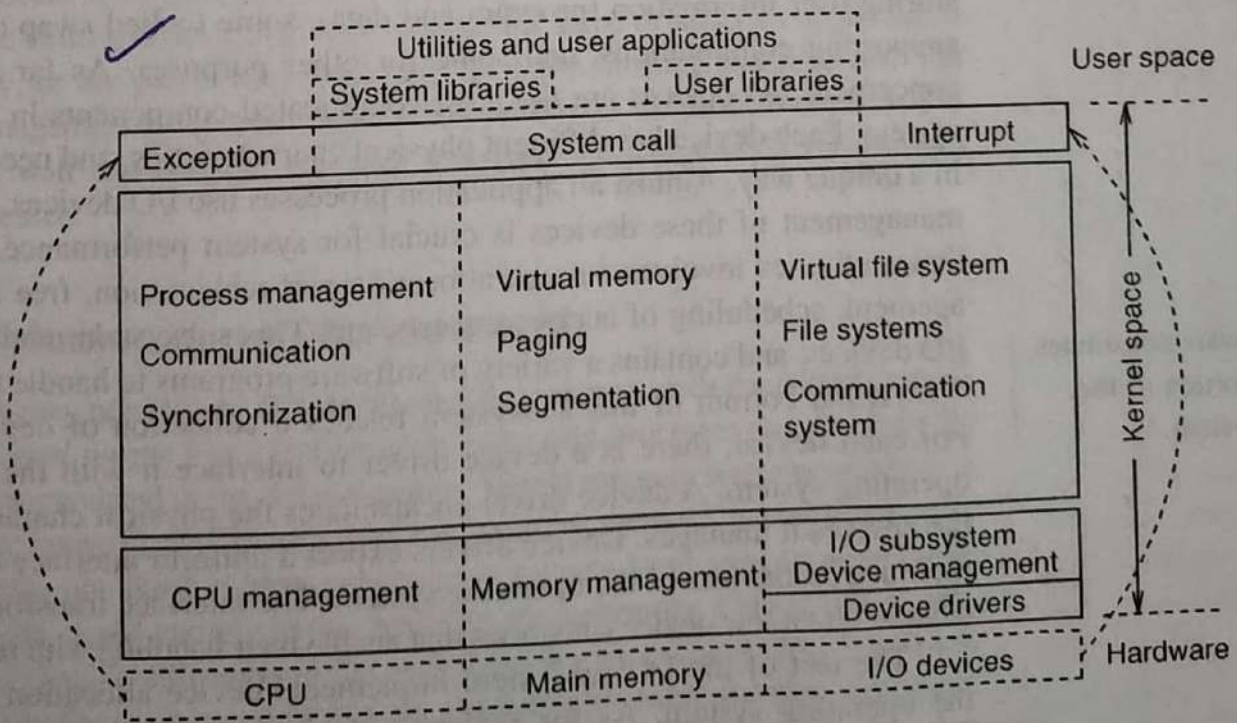


Figure 3.1: *The components of a hypothetical operating system.*

3.3.1 CPU Management

This subsystem manages the CPU, which is the most important hardware resource in a computer. The CPU manager is the core software component that (time-) multiplexes the CPU among contending processes. When the manager receives the CPU back from a process, it decides, based on the scheduling parameters, which ready process will get to use the CPU next, and for how long. The manager implements the CPU-scheduling policies of the operating system.

» Many types of operating systems such as batch, multiprogram, time-sharing and real-time are available. The CPU-scheduling policy is primarily responsible for creating these variations.

3.3.2 Memory Management

This subsystem manages the main memory, which is another important hardware resource in a computer. The application programs and the operating system (space-) share the main memory. The memory manager keeps track of where, in the memory, the operating system programs reside, and where the application programs reside. It keeps an account of the allocation of specific parts of the memory to specific processes, and of the parts that are free. It implements the memory-allocation policies of the operating system and the various schemes to protect allocated memory from unwarranted use and unauthorized accesses. It sets up shared memory regions for application processes, and connects CPU-generated addresses to the appropriate memory locations through the hardware MMU.

» Shared memory regions are used for data/code sharing and interprocess communications.

3.3.3 I/O-device Management

A computer system uses many varieties of I/O devices. I/O devices enter data into the system and retrieve them when needed. Some devices are used for storing user information (program and data), some (called swap devices) for supporting computations, and some for other purposes. As far as usage is concerned, I/O devices are the most complicated components in a computer system. Each device has different physical characteristics, and needs handling in a unique way. Almost all application processes use I/O devices. The proper management of these devices is crucial for system performance. Managing their activities involves storage allocation and reclamation, free space management, scheduling of access requests, etc. This subsystem manages all the I/O devices, and contains a variety of software programs to handle the devices.

» I/O software constitutes the major portion of the operating system.

At the bottom of this subsystem resides a collection of device drivers. For each device, there is a device driver to interface it with the rest of the operating system. A device driver encapsulates the physical characteristics of the devices it manages. Device drivers export a uniform interface to facilitate their use by the rest of the operating system. The interface transforms diverse devices into a few device categories that enable their handling with relative ease.

The rest of the I/O subsystem implements device allocation policies of the operating system. As for exclusive devices (those that are either non-sharable, or for dedicated use), it keeps an account of the specific devices

allocated to specific processes. It allocates devices to processes whenever they are needed, and reclaims them when these processes no longer require them. The I/O subsystem also implements a data cache to temporarily hold device contents for enhancing their access speed.

3.4 Middle-layer Subsystems

This layer is the head and heart of an operating system. Most users predominantly view file systems and communication systems, and some view process systems too.

3.4.1 Process Management

An operating system executes application programs in the abstraction of processes, and uses a process as a handle to manage one program execution. A process is treated as a unit for purposes of resource allocation and work scheduling. Each process needs certain resources to accomplish its task. The operating system tracks the allocation of resources to the different processes, and stores this information in process-related data structures. This subsystem also determines process priorities for resource allocation.

The process management subsystem has various components or subsubsystems. The most notable are interprocess communication and process synchronization. The interprocess communication subsystem implements a few communication primitives to facilitate communications among a group of (cooperating) processes. The synchronization subsystem implements synchronization primitives that are used to coordinate process activities relative to one another.

One process can create another process and the latter is then called the *child process*. The child and the parent may execute the same or different applications concurrently, or the parent may wait until the child finishes its execution. The process management subsystem implements primitives for process creation and destruction, wait and resume, etc., and maintains the parent-child relationship for all processes.

3.4.2 Virtual-memory Management

At times, we may need to execute processes whose size exceeds the available main memory. *Virtual memory* is a technique of executing processes that may not be entirely accommodated in the main memory. Virtual memory implementation is a logical layer between applications and the physical memory management subsystem. On the one hand, it helps processes to execute large applications even if they cannot be stored entirely in the available main memory. On the other hand, it promotes effective utilization of the main memory by keeping only those parts of processes in the main memory that are absolutely essential for their execution. The remaining parts of the processes are held in secondary storage devices that

collectively implement an auxiliary memory or swap space. Virtual memory transforms limited availability of the main memory to the size of the auxiliary memory. It treats the main memory as a (computation) cache for the auxiliary memory.

3.4.3 File-management System

Information is normally retained in a computer system for a long duration and is stored in a variety of storage devices such as disks, tapes, and CDs. These devices can retain information even when the computer is shut down. Different device have different physical characteristics, and different ways of storing and retrieving information from the storage mediums. These devices are difficult to manipulate by (naive) application developers, even through the uniform interface provided by their device drivers with the I/O subsystems.

Users view stored information as sequences of records. Each of these sequences is referred to as a *file*. The file is a device-independent concept, and is a unit for information storage and maintenance. Although the users view information in abstraction of files, the devices store information in the form of physical signals in their recording mediums. A file management system bridges the gap between these two views. Files are logical entities, and are crafted from the stored physical signals by the file management system. The latter assists the operating system in mapping logical files into physical entities in the devices.

The file management system is the most visible component of an operating system. It helps users organize their files in directories. A *directory* or *folder* is the logical place to hold related files and/or directories. The file management system implements primitives for file- and directory creation, deletion, read, write, and many other operations used by applications to create, manipulate, and maintain the contents of files. It also ensures that files are not accessed by unauthorized users.

» Ethernet is a typical communication medium that physically connects two or more computers. It carries out the physical exchange of messages among the computers. The well known TCP (transmission control protocol) and IP (Internet protocol) are communications protocols o exchange messages at igher layer(s) of stract ion. Higher-level otocols help processes to change data among nselves, independent of physical characteristics e transmission ums.

3.4.4 Communication System

The communication system is the backbone in forming a computer network. A collection of computers connected to each other forms the network. These computers do not share memory or peripheral devices, and interact with one another by exchanging messages over their communication mediums. This subsystem helps processes in different computer systems exchange data among themselves. Each computer system implements logical communication ports through which processes can exchange data. A local process that wants to communicate with a remote process connects itself to a local port. To send some data to the remote process, the local process writes the data to the local port with the help of a suitable communication driver, and the operating system arranges to transport the data to the remote computer at a designated remote port there. A receiving process in the remote computer reads the data from this remote port.

3.4.5 Virtual File System

Some modern operating systems allow many different independent file management systems to coexist in the same computer system. It may be difficult for applications to handle these different (and often divergent) file systems. They access these "real" individual file management systems through a uniform interface provided by the virtual file-system switch (VFS, for short). The VFS is a thin interface layer between the applications and real file management systems. The VFS routes file-access requests to the appropriate real file management systems. Thus, real file management systems, though often varying from one another, appear identical to the rest of the operating system and applications. In some operating systems, the VFS also helps applications to treat the communication system as a kind of file system. (For example, **sockfs** in Linux.)

3.5 Top-layer Subsystems

This layer is the system software interface to applications and interrupt capable devices. It also includes the utilities and the libraries.

3.5.1 Interrupt, System Call, and Exception

Modern computers are interrupt-driven. Users interact with a computer through I/O devices such as the keyboard and the mouse. I/O devices interrupt the CPU to inform the occurrences of events (such as arrival of a new character from the keyboard). The interrupt is the only way users can interact with a computer system.

When users run an application, the application obtains services from the operating system by making system calls. The operating system normally implements a generic routine to handle all system calls. Each system call is identified by a different call number, and may need different parameters. The generic handler routine interprets the parameter values of system calls, and routes the calls to the appropriate components of the operating system.

Occasionally, applications, because of programming defects, may enter anomalous states by performing some illegal actions such as executing an unrecognized- or privileged instruction, dividing a number by zero, referring to entities outside their own address spaces, etc. Such illegal actions lead to what we term exceptional conditions, and these conditions arise during execution of applications. When an exceptional condition arises and the processor hardware cannot resolve it, the hardware raises an exceptional signal to attract immediate attention of the operating system for its resolution.

Though interrupt, system call, and exception are three different conditions, they are similar in the sense that they all need operating system services, and most operating systems handle them through the same interface. A single interrupt service system handles them all, treating each of them like an interrupt. Such uniform treatment of varied conditions makes the design of the operating system interface substantially simpler.

» In this context, the system call is termed a software interrupt, and the exception an internal interrupt. They are also called traps by some authors.

3.5.2 Utilities

» Utilities are like handy tools. They can do amazing things such as finding and fixing files, tuning systems for fast executions, and protecting user process and data, cleaning up and backing up disks, etc., for users and applications.

Utilities are standalone executable programs in computer systems. They come with the operating system distribution from vendors. They are special-purpose applications, and are often called system programs. They help users in manoeuvring the computer with relative ease. The utilities in a computer include compilers, assemblers, logins, shells, database management systems, etc., and provide a friendly and convenient environment for the users and define the "user interface" to the computer system.

Utilities are not really part and parcel of the core of the operating system. They are like any other user application. For example, in UNIX, when a terminal is not in use the operating system executes a login program in an application process. This process prompts for a login name and password on the terminal, and waits to receive a response from a user. On receiving the information, it verifies the name and password, and sets up an initial working environment for the user. This step entails executing a standard shell program (sh, csh, ksh, bash), which is another utility. Utilities except shell are not discussed in this book. They are treated merely as any other user application.

3.5.3 Command Interpreter or Shell

» A shell may be compared to a receptionist in an organization whose main duty is to direct the visitor's (user's) requests to the appropriate professionals in the organization (service routines inside the operating system).

The shell is the single most important utility in modern operating systems such as UNIX. These systems support many types of shells that are command line interpreter programs. Shells help users execute their applications and utilities with relative ease. When a user logs into a computer system, a shell program is executed on her behalf and we say that she has started a session with the system. In UNIX, the shell designated to run the session is specified in the **/etc/passwd** file. The shell acts as the main interface between the user and the computer system, even though the true interface is the system call. (An operating system may support many shells, and they do not need to run in the kernel mode.) The function of a shell is very simple—to read a new command (from the keyboard), execute the command read, and report the execution status and output (to a monitor). As long as the shell is executing, the user remains logged into the system. When the shell terminates, the login session, too, ends.

» *Microsoft Windows operating systems provide GUI-based menu-driven systems. These systems are driven by mouse clicks. Traditionally, they support a command line interface.*

3.5.4 System Libraries

braries allow greater arity, faster ation, and easier nance of application s.

A library is normally defined as a collection of information, resources, and services. What exactly are these in the context of a computer system? A *program library* (or simply a library) is an entity containing machine-language code that may be incorporated into application programs at link time, load time, or runtime. A library is a binary image that is executed by the CPU. However, unlike utilities, a library is not a standalone application program. A library contains a set of related functions that provide some specific services to applications that incorporate the library in

classified into three categories based on the time of their attachment to applications: (1) static libraries, (2) shared libraries, and (3) dynamically loaded/linked libraries.

A *static library* is one that is installed in an application executable before that executable is loaded into the memory. The installation is actually done at the application-link time, and the binding between program identifiers in the application and those in the library is done at the link time. Every application has its own copy of a static library. One unavoidable problem here is that when a static library is upgraded, all the applications referring to it need to be recompiled and/or re-linked to re-install the upgraded library in them. Static libraries are not very popular these days because of the advantages shared libraries (described in the next paragraph) offer to application developers.

A *shared library* is not installed into an application executable, but is linked to the executable at load time. The binding of program identifiers in the application and the library is done at the load time, and before the program execution begins. A shared library can be shared by many different applications. Unlike static libraries, a shared library can be upgraded without any recompilation of application programs that reference the library. Thus, applications are insulated from future upgrades of shared libraries as long as the specifications of APIs that the libraries support remain unchanged.

A *dynamically loaded library* (DLL) can be used by an application at any time while it is running. It is not installed in the application at link time, nor is it added at load time. The DLL is not really a different kind of library format compared to shared libraries. The difference is in how programmers use DLLs. Unlike shared libraries that are linked to applications at load time by the system, DLLs are linked to an application only when it needs them. However, DLLs require a little more work to use; they have to be explicitly opened when an application needs them. The binding of program identifiers in the application and a DLL is done at runtime (at the library open time or when the identifiers are referenced for the first time). The program can explicitly close the DLLs or the operating system implicitly closes them when the program execution terminates.

» The libraries shown in Fig. 3.1 are shared and DLL libraries. Normally, user libraries sit on the top of system libraries. They can form a hierarchy of libraries. Library stacking is a good way of structuring applications.

3.6 Other Subsystems

We have clubbed together remaining small but important subsystems here. Except for the security subsystem, the others are not studied in later chapters.

3.6.1 Bootstrap

When a computer is powered on, almost all hardware devices including the main memory are in inconsistent states. The operating system is not available in the main memory at that juncture and, consequently, the CPU cannot execute the operating system programs to initialize the hardware devices.

When a system is powered on or forced to reset itself, the CPU needs an initial program to execute. This initial program is embedded in a ROM, and is

called a *bootstrap program* or *bootstrap loader*. It is a very simple program that initializes system hardware (such as processor registers, cache memory, main memory, and I/O controllers). The bootstrap loader knows from where to obtain a copy of the operating system and how and where in the main memory it is loaded. The bootstrap program, after loading the operating system, transfers the execution control to the operating system. The operating system initializes hardware devices appropriately. This initialization is done before users begin using the computer.

Bootstraps are highly dependent on processor architecture, and as usual, are specific to individual operating systems. Most modern computers implement multilevel bootstrapping, and incorporate a tiny bootstrap loader in the ROM. The loader finds a full bootstrap program from a booting device, and loads it to initialize the operating system. We will discuss more about bootstrap in the context of the Linux operating system in Section 17.3.1.

3.6.2 Time Management

For various reasons, the operating system needs to know the current real time. For example, it may like to know the creation time of a process to find out whether the process has already spent too much time in the system. Numerous system activities are driven by real time- and interval time events. For example, automatically switching off the screen is an interval timer-driven activity. The operating system also maintains many interval timers that are used to generate events in the system when their time settings expire. Modern processors have a built-in timer device that ticks at regular intervals. At each tick, the operating system updates all the software timers it maintains.

3.6.3 Protection and Security

In multiuser systems, a user may not be authorized to use all the hardware- and software resources, especially the files that belong to other users. Consequently, it is necessary to continuously monitor activities of application processes run by users. A mechanism needs to be employed by the operating system so that these processes do not take undue possession of unauthorized resources. Protection is a mechanism to control an application process's accesses to various system resources. Every access to a resource is checked by the operating system to verify whether the requesting process possesses sufficient rights on the resource to carry out the accessing operation. The operating system prohibits unauthorized accesses to resources. Such protection helps in the smooth functioning of the computer system, and protection-related checks, in the general, improve reliability of the computer system. Protection mechanisms are embedded in almost every subsystem.

Security involves protecting a system from external interference by outsiders (usually illegal users). Each legal user stores her programs and data in her private storage space. The operating system must ensure the privacy and

security of such private storage spaces. The private information must not be viewable by or inferable to unauthorized users. The operating system must prevent illegal users from using the computer.

3.6.4 Module Management

It is not expected that an operating system knows how to interact with new types of devices that did not exist at the time the operating system was created. Further, it may not be worth including in an operating system interfaces for all types of devices known to the world. Creating an operating system with built-in functionalities for all kinds of devices is not practicable because its size would bloat enormously. Furthermore, operating system designers cannot foresee the details of all future devices. Most modern operating systems allow interfaces for addition of new types of devices to the kernel at runtime. This is enabled by adding new device drivers to the kernel in the form of kernel modules at runtime. A *kernel module* is an executable object that can be linked to, and unlinked from, the kernel at runtime. Dynamically loading and unloading a system code on demand is an attractive feature as it helps maintain the kernel size to a minimum. A code not immediately required can be offloaded from the kernel. Modules can also be useful for developing and testing new kernel codes without having to rebuild the entire kernel and reboot the system every time the kernel code is modified.

» A module is like a DLL; it is not a standalone program, but can be incorporated and linked into the kernel at any point after the system bootstrapping is complete. It can also be unlinked from the kernel and removed when the system no longer needs it.

A kernel module consists of a section of statically allocated data and functions. Some of these functions are exported to the kernel. The kernel invokes these functions to obtain services from the module. There are two specific functions, namely *init* and *cleanup*, that each module must have. The former function is executed by the operating system when the module is loaded into the kernel. The *init* function initializes module data structures, and may also activate some device hardware that the module is required to manage. The *init* function registers the module with the kernel so that the rest of the operating system becomes aware of the new functionalities in the system. The *cleanup* function is executed by the operating system when the module is unlinked from the kernel. The *cleanup* function then deregisters the module from the kernel, and releases all the resources that the module had acquired.

A kernel module may reference kernel data and functions. When the module is loaded into the kernel, all unresolved references to the kernel symbols are fixed by the module loader. The kernel maintains a symbol table for this purpose. Even if there is a single unresolved symbol, the module cannot be loaded in the kernel. To load a module, the operating system first reserves a contiguous area in the kernel space. The module code is relocated accordingly to the reserved space as the loader loads the module. A module may export its own symbols to the kernel symbol table so that other modules that are dependent on it can be loaded later. Module stacking is a good way of organizing kernel codes.

» Kernel modules must be written with extreme care as they are known to be a major source of system crashes. They are not normally as well-tested as core kernels are. A rogue module can even damage hardware units.

Summary

An operating system simulates a software platform on top of the hardware platform, which the computer users can use with relative ease. Design and development of a new operating system is a challenging task, but the challenge is made tractable by compartmentalizing the system. This chapter describes four different views that look at operating systems, namely the programmer, the logical, the physical, and the development views. As operating system experts, we always look at the logical view. This view is also called the software architecture of the computer system.

This chapter presents a simple architectural model of a hypothetical operating system. The model has three layers. (1) The bottom layer has three components, namely the CPU, the main memory, and the I/O-device management. The CPU manager multiplexes the CPU among processes. The memory manager allocates and deallocates the memory to the application processes and the kernel based on their needs. The I/O subsystem is overall incharge of allocating I/O devices to the processes and scheduling I/O requests to devices. (2) The middle layer has three components, namely the process management-, the virtual-memory management-, and the file management systems. The process management system creates, maintains, and destroys processes according to requirement. It also supports primitives for interprocess communications and process synchronization. The virtual memory system helps users to run applications whose size can be larger than the size of the main memory. It simulates a very large memory system by using a backing storage device such as a disk. The file management system helps users store and manipulate their information in the abstraction of files. A computer system can have many different file management systems. These file-management systems are hidden under a virtual file-system switch (VFS). The VFS may also handle inter-computer message communications. (3) The top layer provides interfaces for system call-, interrupt-, and exception services. Application processes avail services of the operating system by means of system calls. I/O devices attract the attention of the CPU using interrupts. Whenever a program execution reaches some unusual situation, the internal processor hardware raises exceptions to draw the attention of the operating system to the situation. This chapter briefly discusses these components and prepares the ground for the remaining part of the book. It also discusses the utility, shell, library, bootstrap, and kernel module subsystems in sufficient detail; these concepts are not explored further in later chapters.

We follow some guidelines professed by Bovet and Cesati (2001) to present the subject material of this book in the later chapters. We adopt a segmented bottom-up approach to present the subject material. We discuss topics bottom-up from the three vertical segments shown in Fig. 3.1, and from the left to right order in the following chapters.

Literature

Many operating systems have been developed over the past several decades. UNIX (HP UX, IBM AIX, Linux, SUN Solaris, etc), MS-DOS, Microsoft XP, IBM OS/2, Apple Macintosh are only a few we mention here. These operating systems are commercially very successful. Many more either proved commercially unsuccessful or became obsolete over time.

Modern general-purpose operating systems are time-sharing systems. Strachey (1959) originally proposed the notion of time-sharing systems. Time-sharing systems were developed in CTSS at MIT (Corbato et al. 1962), in SDC Q-32 (Schwartz et al. 1964), and in many others. These operating systems are decomposed into several subsystems for easier development and maintenance. Hierarchical decomposition of complex systems was strongly advocated by Dijkstra (1968b; 1972) and also by Parnas (1972). A well-structured

modularized system is easier and faster to build, test, and maintain.

The "THE" operating system described by Dijkstra (1968b) used a layered composition of operating system software; it had six layers: hardware, CPU scheduler, memory management, console management, I/O system, and application programs. Later the layered design approach was adapted to other systems such as Venus (Liskov 1972), and had seven layers. Such an excessively layered approach was later found to be less efficient, and few modern operating systems support it. Modern operating systems support only a limited number of layers. Hansen (1970) constructed the core kernel on which other subsystems were built. His is considered the first microkernel-based system even though the terminology was not conceived then (see Liedtke 1995).

Bourne shell (by S.R. Bourne) (/bin/sh) was the earliest command line interpreter. Later came C-shell (/bin/csh) from the University of California, Berkeley, which had better job control and aliasing features making it more interactive; its programming resembled C language. Korn shell (/bin/ksh) by David Korn from AT&T Bell Laboratories was put together from the best features of both Bourne shell and C-shell, but K shell was not free. Bash shell (/bin/bash—the Bourne again shell) was created as a part of GNU to be free and to replace K shell.

Exercises

1. What is a software platform and why do we need it? How is it different from a hardware platform?
2. What are the components of the software platform in a typical computer system? What purposes do they serve?
3. What are the two main responsibilities of an operating system?
4. What are the different perspectives to view operating systems?
5. What are the interfaces by which external devices can interact with the operating system?
6. What is a utility program? Give some examples of utilities.
7. Is utility a part of the operating system? How is a utility different from other (user) application programs?
8. Can we get rid of all utilities from a computer system? Justify your answer.
9. What is a command interpreter? To which part of the operating system does it belong? How is it different from other utilities? Explain your answers.
10. What are the special characteristics of a command interpreter? Why is not a command interpreter built into the kernel?
11. What is a library? Explain the types of libraries available in a typical system. How are they different from one another?
12. Compare and contrast utilities and library.
13. Describe what happens when a process calls a routine in a dynamically linked library for the first time.
14. What is a bootstrap loader? How is it different from a bootstrap program?

CHAPTER

5 CPU Management

Learning Objectives

After reading this chapter, you should be able to:

- Describe how multiprocessing is achieved by switching the CPU among processes.
- Explain what a context switch is and how it is performed.
- Describe process preemption and its complexity.
- Discuss the merits and demerits of various CPU-scheduling algorithms.
- Explain thread scheduling in user- and kernel spaces.

5.1 Introduction

The central processing unit (CPU) is the most valuable hardware resource in a computer system and it is the only resource that actually executes processes. One objective of operating systems is to raise the effective utilization of the CPU as much as possible. To do this, multiple processes are allowed to co-exist in the system. That is, at any given point of time, it is most likely that some process will be using the CPU. (Almost all modern operating systems allow multiple processes to co-exist; hence, unless otherwise specifically stated, hereafter the term operating system refers to a multiprocess operating system.) While processes share the CPU, they can only use it mutually exclusively. Therefore, CPU allocation forms the basis for any multiprocess operating system, and the system allocates the CPU to ready processes in an orderly manner. By switching the CPU among processes, the system ensures that all processes have a fair share of the CPU time. In this chapter, we study various CPU allocation algorithms, and the pros and cons of each algorithm. We also discuss thread scheduling as done in multithreaded systems. Before going on to do so, we discuss the goals of scheduling algorithms and the various metrics to evaluate their performance.

5.2 Scheduling Objectives

In a computer system, many application- and kernel processes can run concurrently to improve the utilization of system resources. Processes use various resources to accomplish their tasks. The use of resources by concurrent processes often leads to conflicts over resources, and unless resolved by the operating system, the reliability of the system might be jeopardized. The system needs to resolve access conflicts by allocating resources to processes in an orderly manner. The set of rules by which the system evaluates who is given a resource, when, and for how long is called a resource allocation- or scheduling policy. The management of resource time is called *resource scheduling*. The goal of resource scheduling is to provide a predictable level of service to all the processes that compete for a resource.

Resource scheduling is purely an operating system activity, and applications are not involved in the related decision-making tasks. That is, they do not include routines for resource scheduling in their programs. The system must follow good resource allocation policies to assign resources to processes in a fair- and rational manner. Otherwise, the performance of the system as well as that of individual processes may be degraded considerably.

In general, when the operating system attempts to allocate a resource among many processes waiting for it, the system has to decide in a finite amount of time to which process will the resource be allocated next. The system follows a set of rules to reach the decision. An implementation of such decision rules is called a *resource-allocation algorithm* or *request-scheduling algorithm*. A scheduling problem is to determine a proper order in which to serve the requests that are waiting. For every resource, there is an allocator or manager that supervises the resource. The manager implements allocation and deallocation algorithms for scheduling and releasing requests for the resource. In this chapter, we discuss a few CPU-allocation algorithms. (Other resource-allocation algorithms are discussed in later chapters.)

» Resource allocation is a problem of a generic nature and that of scheduling can be considered to be a special case. Allocating a complete resource to a process for exclusive access is known as scheduling the resource to the process. The CPU has to be allocated and used exclusively whereas resources such as the memory, the disk space, and the files may be partially allocated or used. This is why, in the context of the CPU, it is called a scheduling problem, and in other contexts it is called an allocation problem.

Most operating systems create a large number of processes that are executed on a small number of CPUs (one CPU in uniprocessor systems and multiple CPUs in multiprocessor systems). Our objective is to rationally distribute CPU time among the processes. Because many processes can be ready for execution at the same time and a single CPU can execute only one process at a time, the operating system uses various criteria to select the best process then available for execution. The system allocates the CPU to processes either by executing them one at a time to completion or by executing several of them in an interleaved manner, each one for a short period of time in rotation. (In the latter case, we say the CPU is *time-multiplexed* among the processes. The processes feel that they each have a dedicated CPU for their use.) In either case, the CPU would pause from normal activities at different points in time to make its scheduling decision. The decision would be to either continue with the currently running process or switch to a different one. The events that trigger a scheduling decision are the completion of the current process, a state transition of the current process (to wait, sleep, or blocked states), a timer interrupt, arrival of a higher priority process, and so forth.

» In a single-CPU system, once the CPU is allocated to a process for a time duration of t units, no other process can do any useful work in the time t. Any other process has to wait for the expiry of time t to obtain the use of the CPU. Therefore, the problem of scheduling the CPU may also be viewed as a time-scheduling problem.

The operating system program that implements CPU-allocation policies is called the *CPU scheduler* or *CPU manager*. The CPU scheduler is a very important component of an operating system even though it occupies a tiny fraction of the operating system code. When a CPU becomes idle or the system decides to preempt the running process, the system invokes the scheduler to select a different process to run on the CPU. Then, the system invokes a "context switch" routine to start the execution of the selected process.

» During the execution of a process, both CPU and process need each other to do any useful work. Therefore, scheduling a process to a CPU has the same impact as scheduling a CPU to a process and, therefore, we consider both to mean the same.

In most operating systems, CPU scheduling decisions are made at two levels of abstractions. At the higher level of scheduling, the objective is to control the degree of multiprocessing. It is called *long-term* scheduling. How many processes and which of them will be brought into the main memory for execution and which process executions will be deferred are decided at this level. At the lower level, the objective is to allocate a CPU to processes no sooner the CPU becomes free. It is called *short-term* scheduling. In each invocation, the scheduler chooses the best process from all those that are ready. In this chapter, we will study short-term scheduling algorithms. We consider only deterministic algorithms in which the values of all the parameters that control scheduling decisions are known before attempting the execution of the algorithms. The goals of short-term scheduling are improving CPU utilization, increasing throughput, reducing average response time, reducing the variance of response time, and so forth. We define performance metrics next.

» Some time-sharing systems such as UNIX do not have a long-term scheduler. Once a process is created, it becomes immediately ready.

5.3 Performance Metrics

The main objective of a short-term scheduler is to offer "good" services to all processes in the system. Many metrics are used to evaluate the "goodness" requirement. In the early days when hardware resources were expensive, the objective was to get as much work done as possible. The unit of work could be as small as one instruction or as large as one complete process. The measure used was *throughput* and is defined as the number of work units completed per unit time. For example, the number of application processes completed per second is a measure of the throughput of the system. To get as much work done as possible, we may require keeping all system resources, particularly the CPU, as busy we possibly can.

» Performance metrics are measures to assess how well a system is performing or meeting the needs and expectations of its users. An accurate measurement of these metrics can expose the strengths and weaknesses of a system.

Keeping a resource busy certainly raises its *utilization*, which is defined in percentage as the ratio of the busy time to the total time. If the busy time equals the total time, then the resource is considered one hundred per cent utilized. The non-busy time is called *idle time*. Although keeping a resource busy might increase its utilization, it may not increase the throughput. For example, a CPU may be busy in executing some instructions, which are not from any of the application processes. Such work is called system *overhead*. Therefore, a related objective is to reduce the system overhead to increase its throughput.

Throughput is a metric for measuring system performance. The related metric from the users' perspective is *turnaround time*—the time between the submission and completion of a job. One objective is to minimize the average turnaround time (including the waiting time and the actual processing time). For interactive systems, the *response time*, the time taken to respond to an

» Interactive applications are primarily reactive in nature. They iteratively get inputs from users, process the inputs, produce outputs, and respond back to the users. Thus, turnaround time is not an effective performance measure for these applications.

interactive request, is considered more important than the overall turnaround time. *Proportionality* and *predictability* are the higher-level metrics related, respectively, to turnaround time and response time. Proportionality is a measure of the amount of expected time relative to its size, and predictability to that of the expected quality of service. It is natural to expect good service to guarantee that smaller jobs stay in the system for a smaller time than larger jobs.

Statistical measures such as minimum, maximum, average, variance, etc., are important performance metrics for response time. Scheduling algorithms do not directly affect a process's waiting time for an event or I/O operation, but they do affect the amount of time a process may wait in the CPU queue. Therefore, reducing the average wait time in the queue is also be an objective of the CPU scheduler. Variance of response time is viewed a more appropriate measure of the *quality of service* (predictability). Minimum variance implies a higher quality of service. *Meeting deadlines* and predictability are the two dominant performance metrics in real-time systems. Failing to meet deadlines in many life-critical real-time systems can result in catastrophic consequences. We will study CPU scheduling for real-time systems in Chapter 15.

5.4 A System Model

» Note that all CPUs share the same ready queue. It may so happen that more than one CPU accesses the ready queue simultaneously to select processes from the queue. The CPUs must synchronize their accesses to the queue. This is called CPU or processor synchronization. Processor synchronization and other general synchronization problems are separately addressed in Chapter 7. For the purpose of this chapter, we assume that CPUs access the ready queue mutually exclusively. In fact, we can assume we are in a single CPU system.

To study scheduling algorithms and compare them with one another, we need a model to abstractly represent the system. For the purpose of this chapter, we consider a tightly coupled multiprocessor computer system. There are a finite number of identical CPUs in the system. They all are connected to the same main memory. We assume a symmetric multiprocessor system where all CPUs execute the same operating system programs, and make their own decisions. The main memory is assumed to hold all ready processes. Figure 5.1 presents a model displaying CPU allocation to running processes; CPU1 is executing Process 1, CPU2 Process 2, and so forth. Normally, there are more ready processes in the system than there are CPUs. Consequently, many of these ready processes have to wait for their turn for allocation of a CPU. All ready processes are tracked down through a data structure called the *CPU queue* or the *ready queue*.[1] The CPU scheduler, when executing, selects ready processes for execution in an orderly manner, based on the short-term CPU scheduling policies of the operating system. The process selection is based on criteria, called priorities.

Typical processes alternate between their computations and waiting events such as input/output activities. A process starts its life with a CPU burst, then waits for an I/O operation or some other event, then another CPU burst, then another wait, and so forth. It alternately receives a CPU burst and waits for some event. In its lifetime, a process moves through a number of queues—the CPU queue, device queues, and event queues—where it receives various

[1] Note that term

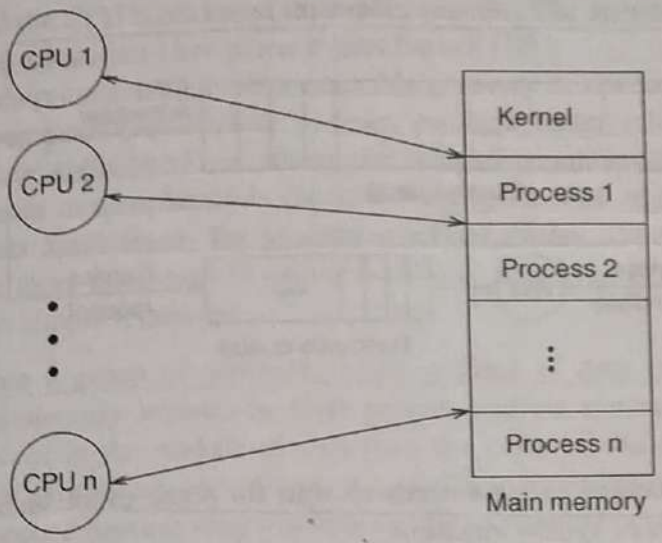


Figure 5.1: *The connection between the CPUs and the processes.*

services or simply waits for some events to occur. Finally, the process completes its execution with a final CPU burst.

Figure 5.2 presents a schematic representation of how processes flow in a typical computer system. (It is also known as the queuing model.) The figure is one way of realizing the process state transition diagram of Fig. 4.8 on page 97. For the sake of simplicity, we have shown only a single-resource queue in the figure. The arrows in the figure show the direction of process flow in the system. A new process initially joins the ready queue, and waits there for a CPU allocation. Once the operating system allocates a CPU to the process, it starts running (i.e., the CPU starts executing instructions on behalf of the process). After a while, the process either (1) leaves the system, or (2) waits to obtain service from another resource, or (3) is temporarily suspended by the operating system. In any of the cases, the operating system reclaims the CPU from the process, and allocates the CPU to another process. In the first case, we say that the process execution is complete; in the second, the process waits in a resource queue or an event queue; and in the third, we say the CPU is *preempted* from the process and the process waits at the ready queue until the system allocates a CPU to it again. For the second case, the process eventually comes out of the waiting state, and joins the ready queue and waits there for the next CPU allocation to it.

» The current state information of each process is stored in the state component of the process descriptor (see Fig. 4.2).

» The moments in time when context-switching calls are made are called *preemption points*. Alternatively, a process may yield to another process either explicitly or implicitly when it does not require the CPU (e.g., when it needs to wait for a resource). This way of giving up the CPU is common in thread scheduling and it invokes preemption by calling a yield function.

5.5 Two Broad Scheduling Classes: Preemptive vs Non-preemptive

When a scheduling decision is to be made for a CPU, the operating system invokes the CPU-scheduling algorithm to choose a new process to run on the CPU. The system may invoke this algorithm in the following situations.

1. The running process finishes its execution. This situation occurs when the process executes a system call to exit or the operating system forcefully terminates the process.

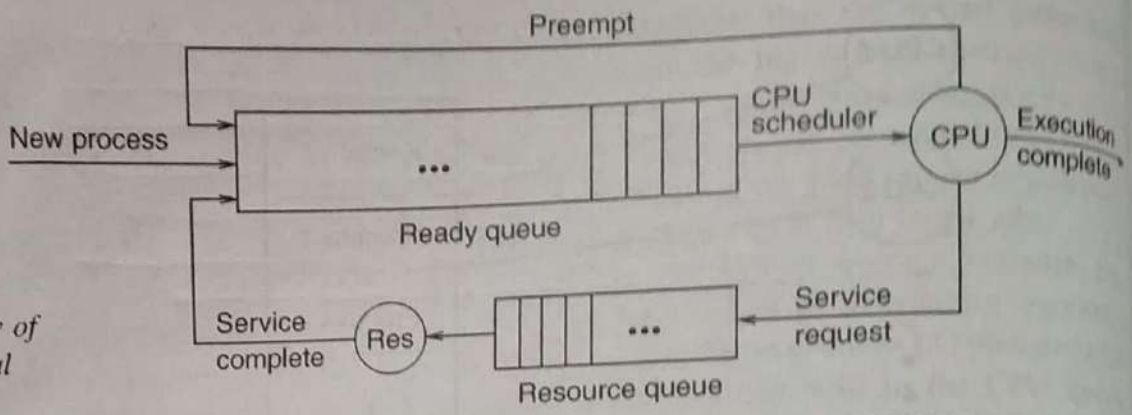


Figure 5.2: *The flow of processes in a typical computer system.*

» We need to understand the difference between preemption and interruption especially when the kernel is running. When a kernel path is running and an interrupt occurs, we have three choices.
1. We may ignore the interrupt and continue with the kernel path. In such an event, there is no preemption of the CPU from the running process or from the kernel path.
2. We may start a new kernel path to serve the interrupt. We then have preemption of the former kernel path, but not that of the running process. The new kernel path runs in the context of the same process.
3. We may suspend the running process. We have preemption of the running process, and we resume another process. We also have preemption of the kernel path if the latter process executes a different kernel path.

2. The running process needs to wait for some event such as I/O completion or synchronization.
3. The running process has exhausted its allocated time.
4. A waiting process becomes ready to run. This situation arises when the event for which the process has been waiting occurs.
5. A new process arrives in the system. This situation occurs when the currently running process has created a new process.

For situations (1) and (2), the currently running process voluntarily releases the CPU and the system has no choice but to select another process to run, or else the CPU remains idle. However, for other situations, there is a choice for the system to decide whether or not to continue with the running process. If the system permits CPU-scheduling only under the former two situations, it is a *non-preemptive system*. Otherwise, it is a *preemptive system*. Preemption means forced suspension of the current process execution before it voluntarily releases the CPU. The preempted process resumes its execution later when it receives a CPU back.

CPU scheduling policies are broadly classified into two categories: preemptive and non-preemptive. In non-preemptive scheduling, when a process is allocated a CPU, it runs as long as it needs the CPU. In preemptive scheduling, a running process may be suspended in the middle of its execution, and the operating system may reclaim its CPU to preempt the process and allocate it to another process with higher priority or merit. The system reallocates a CPU to the preempted process later. (Some authors call this procedure preemptive scheduling with resumption.)

In non-preemptive scheduling schemes, when a process is allocated a CPU, it continues using it until it releases it voluntarily. A major shortcoming with non-preemptive scheduling is that when a long-running process is allocated a CPU, it monopolizes the CPU until it voluntarily releases it. An even more dangerous situation occurs when, because of programming errors, a process enters into an infinite loop, and thereby, perpetually occupies the CPU, causing the system to stall. In contrast, in a preemptive-scheduling scheme, a running process may be preempted from using the CPU at any point in time. The preempted process is not terminated; it is temporarily suspended from its

execution, and the CPU is allocated to another process. The suspended process will resume its execution later when it gets back a CPU.

CPU preemption seems to be a desirable property of operating systems. In general, preemptive-scheduling policies produce better scheduling outcomes than those produced by non-preemptive scheduling policies. Nevertheless, preemption leads to complexity in the kernel design as well as in the design of multiprocess applications for process synchronization. Thus, preemptive policies cause more overheads in decision-making activities. Let us study the following two simple examples.

1. Suppose a group of processes share a piece of data (by setting up shared-memory regions in their private address spaces). Suppose a process is in the middle of changing the value of the data when its CPU is preempted and allocated to another process from the group. This second process may manipulate the same shared data and put the data into an inconsistent state, and alter the integrity of the data.
2. A process makes a system call, and the kernel is busy handling it by initiating a kernel path. (The kernel path executes in the context of the caller process.) The system call may involve changing the values of some kernel data structures. If the process (and the kernel path) is preempted in the middle of these changes and another kernel path alters the same data structures, the entire system may collapse.

In the first example, processes manipulate data in the shared part of their private address spaces; and in the second example, kernel paths manipulate data in the common kernel space. In either case, we have data corruption problems. We cannot let this happen in the system. We need a kind of coordinated preemption. Modern operating systems (such as Linux) allow arbitrary preemption when processes run in the user space, but not when they run in the kernel space. That is, a CPU from an application process in the user space can be preempted at any time. Application processes need to synchronize accesses to shared data on their own. (This is one instance where the CPU scheduling policy forces application redesign.) However, the system does not arbitrarily take away CPUs from processes when they are executing in the kernel space. The system normally provides a limited number of points where preemptions can safely take place. The preemption points are usually those places in the kernel from where the system is about to return to the user-operating mode from a system call or an interrupt service, or the current process voluntarily releases the CPU on its own.

Preemption is certainly good for interactive systems where users submit short requests and expect replies immediately. Preemption is necessary to ensure faster response to urgent processes. We, however, do not like to debate the advantages and disadvantages of preemptive and non-preemptive scheduling systems here. We assume that at some points of time the kernel will call the CPU scheduler and "context switch" to allocate a CPU to a new process. We first study the context switch in the next section, and various scheduling schemes in the following section.

» In an operating system with a non-preemptive kernel, a process is preemptive only when it voluntarily releases the CPU or when it is about to return to the user mode. In a preemptive kernel, in addition, process preemption may occur in other parts of the kernel.

» Kernel in a conceptual shell:

```
On each processor
do forever {
    Run the current process;
    Choose the next process;
    ContextSwitch(current
    process, next process);
    /*Next process now
    becomes the current*/
}
```

5.6 The Context Switch

It is mentioned in Section 4.4 that each process is modelled by a process descriptor. The execution context of a process is stored in its descriptor. Process context has a hardware dependent part (see Fig. 4.2 on page 88). When a process is running on a CPU, the process owns some hardware resources such as the CPU registers, the memory management registers, etc., and the register values (at least those of general-purpose registers) are normally different from those stored in the descriptor. For example, the latest value of the PC register is not available in the descriptor. Consequently, if the kernel decides to suspend the current process, the latest hardware context value must be saved in the process descriptor. Later, when the kernel again decides to resume the suspended process, the saved context is reloaded from the descriptor before the process resumes its execution. The task of switching the CPU from one process context to another is called a *context switch* (also known as a *task switch*, *process switch*, *process dispatching*, or *context exchange*). Each execution of the context switch first saves the hardware context of the currently running process, and then restores that of the next process to be executed. The operating system invokes the context switch right after the CPU scheduler selects a different process to run. Normally the scheduler is invoked only when a context switch is anticipated.

» Every context switch incurs overheads by a way of saving- and restoring the hardware contexts. The overhead depends on the size of the hardware contexts. Switching between processes is costlier than that between sibling threads. This latter switching saves and restores fewer registers.

» As both processes involved in a context switch are in the kernel space, the context switch does not lead to an immediate address space switch. However, if the two processes are in two different combined-kernel spaces, and there is certainly an address space switch too at the time of the context switch.

» Context switching may not be that straightforward in reality. Before we change the PC value, we change the SP to operate on the kernel-stack of the selected process. The loading of the new PC value is simulated by a return from the context-switch function call.

It is worth noting that process descriptors reside in the kernel space. Hence, context switching is a kernel activity, and happens only in the kernel space. The two processes involved are in the kernel mode. Unlike other kernel executions (due to interruption or exception), a context switch is accomplished in two kernel paths, and these two paths run on behalf of two different processes in sequence. The first kernel path starts in the context of the currently running process. It saves the hardware context in the process descriptor first. It then loads the hardware context of the next process to run. Figure 5.3 shows how the operating system does a context switch in two kernel paths. When the program counter value from the next process descriptor is loaded in the PC register of the CPU, the second kernel path starts running on behalf of this newly selected process. The selected process resumes its execution from where it was stopped, as if coming back to life from hibernation. Note that the switched out process remains waiting in the kernel in the privilege mode.

5.7 Scheduling Schemes

When there are many processes waiting in the ready queue and the CPU becomes free, the operating system selects a process from the queue a

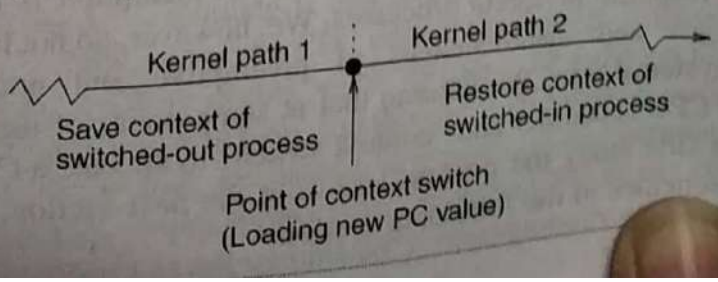


Figure 5.3: *Context switch activity.*

allocates the CPU to the selected process.[2] A scheduling algorithm is executed to make this selection. Scheduling algorithms differ from one another based upon the performance metric they try to optimize. Further, a particular algorithm may concentrate on performance improvement for a specific category of processes. Thus, scheduling algorithms differ from one another only in the choice of processes given preferential treatment. The performance improvement (e.g., reduction of response time) of one category of processes can be achieved only at the expense of performance degradation of processes in other categories. There is no single best scheduling algorithm that is good for all processes. Each operating system employs a different flavour of scheduling algorithm.

» Scheduling algorithms do not alter the execution time of processes. Irrespective of the order the processes are executed, the CPU consumption time of each process to execute its instructions remain the same. However, the CPU consumption time may vary from one CPU to another.

The CPU scheduler occupies a tiny portion of the operating system, but executes very frequently, especially in time-sharing systems. As pointed out previously, every execution of a scheduling algorithm incurs some overhead. Therefore, the algorithm should reach a decision quickly, for which it must be simple and efficient. A large number of scheduling algorithms were invented in the early era of operating system development. In the following subsections we will study some popular scheduling algorithms.

5.7.1 First-come First-serve (FCFS) Scheduling

The first-come first-serve scheduling algorithm is the simplest of all algorithms. It implements the ready queue as an ordinary first-in first-out queue. Processes that request a CPU enter at the tail end of the queue. They are dequeued from the head end of the queue. The scheduler always allocates an available CPU to the process at the head of the ready queue. Essentially, the CPU is allocated to the processes in the order of their making requests for the CPU. The FCFS algorithm is very simple to implement; the ready queue can be constructed as a linked list of process descriptors.

When a CPU is allocated to a process, it uses the CPU for as long as it needs the CPU. There are two important variations possible when the process needs to wait for some event. It could simply hold the CPU in its scope until the event occurs. This satisfies the true spirit of FCFS, but wastes CPU time. As another variation, the process releases the CPU for other processes to use and waits for the event to occur. On occurrence of the event, the process rejoins the ready queue at the tail end, and waits there for its turn to get a CPU (see Fig. 5.4). This increases the CPU utilization with a slight compromise of the FCFS discipline; this version is discussed further.

Let us first study a concrete example to understand the FCFS scheduling discipline better. Let P1, P2, P3, P4, and P5 be five processes that have become ready at the same time. They have joined the ready queue in the said order. Suppose they need 10, 6, 20, 2, and 4 units of time, respectively, to complete their executions. For the sake of simplicity, we assume that they are

[2] If no process is ready to run, a predetermined idle process is run on the CPU. This is true for all practical general-purpose operating systems we know of.

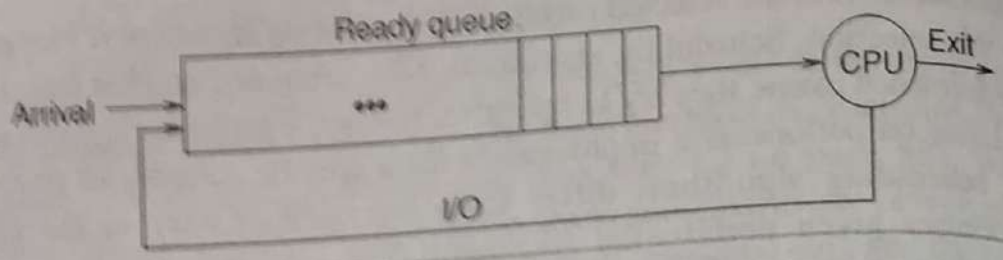


Figure 5.4: *The queuing model of FCFS scheduling with I/O.*

purely CPU-bound processes and they do not execute any waiting instructions. In addition, we ignore the context-switching time. The operating system executes processes P1, P2, P3, P4, and P5, one after another in the said order. Figure 5.5 presents a Gantt chart of CPU allocation to processes. As shown in the figure, their turnaround/response times are 10, 16, 36, 38, and 42 time units, respectively. Hence, their average response time is 28.4 time units. If the processes were executed in the order P4, P5, P2, P1, and P3 (see Fig. 5.6), their response times would be 2, 6, 12, 22, and 42 time units, respectively, and the average response time 16.80 time units.

» A process is regarded as *CPU-bound* if it mainly performs computational work and occasionally uses I/O devices. On the contrary, a process is *I/O-bound* if it spends more time using I/O devices than the CPU. Generally, an I/O-bound process has a shorter CPU burst than a CPU-bound process.

FCFS is a non-preemptive CPU-scheduling discipline. Processes are executed in the order of their arrival to the ready queue. The scheduling discipline favours the longest waiting processes irrespective of their processing time requirements. The above example shows that the FCFS discipline may not yield the best average response/turnaround time. Moreover, if an erroneous program puts a process into an infinite loop, the rogue process will never release the CPU, and the entire system stalls. Therefore, we need to monitor the time consumed by each process. Further, when a process exhausts its time limit, we need to forcefully terminate it.

» FCFS scheduling is the most popular service strategy in practice and is considered fair. It works well for the situations where all processes require similar service times. It is favourable for processes with longer execution times, and not attractive for short processes.

5.7.2 Shortest-job-first (SJF) Scheduling

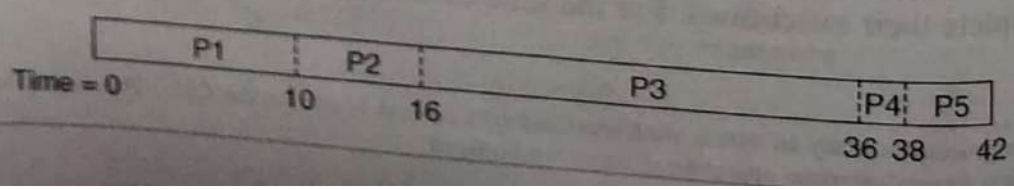
As elucidated in the preceding examples, FCFS, though simple, is not very effective. The efficacy of scheduling algorithms may be improved if we know the execution time requirements of the processes in advance. As the name implies, the SJF scheduling algorithm when executed schedules the process with the shortest execution time, and hence, it favours processes with shorter execution times.

When a process requests a CPU, it must inform the system for how long it wants to use the CPU. When a CPU becomes available, the system allocates it to the process with the then least expected execution time. To break ties (when many processes have the same least execution time requirement) it follows the FCFS discipline. Figure 5.6 is the SJF version of Fig. 5.5.

This discipline achieves the best average turnaround time. Nevertheless, in practice, it is very difficult to predict process execution times. As this scheduling discipline favours processes with shorter execution times, there

» Among the non-preemptive-scheduling algorithms, SJF gives the minimum average turnaround time and is, therefore, optimal for the metric.

Figure 5.5: *The Gantt chart of FCFS scheduling of processes.*



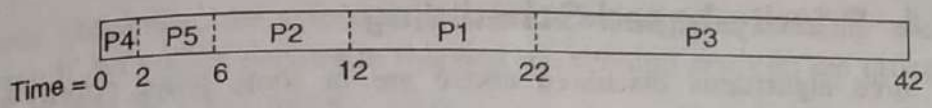


Figure 5.6: *The Gantt chart of a different scheduling of the example of Fig. 5.5.*

a possibility of process starvation happening in such systems, because short processes may repeatedly overtake longer ones. SJF is a non-preemptive CPU-scheduling discipline, and hence, as in FCFS, an erroneous process can perpetually take over the CPU.

» *Starvation* is a situation in which one or more agents wait indefinitely to do something because other agents are overtaking and preventing them start their work.

5.7.3 Shortest-remaining-time-next (SRTN) Scheduling

Although SJF gives the best average turnaround time among non-preemptive-scheduling algorithms, it is not the best overall. Consider a simple example where we have three CPU-bound processes: P1 with execution time 300 s, P2 with execution time 10 s, and P3 with execution time 10 s; they arrive in that order at time moments 0, 1, and 2 seconds, respectively. Since SJF is non-preemptive, these three processes are executed one by one with P1 completing at the 300th second, P2 at 310th second, and P3 at 320th second. The turnaround time for P1, P2, and P3 are 300 s, 309 s, and 318 s, respectively. Thus, the average turnaround time is 309 s. This turnaround time can be reduced if the scheduler is forced to re-examine the execution times of the processes at the arrival of every new process and schedule the process with the least execution time required to complete it. In that case, when P2 arrived at time 1st second, it would be scheduled immediately, then P3 after P2 completes, and finally P1 is resumed to complete. The turnaround time for P1, P2, and P3 are 320 s, 10 s, and 19 s, respectively. Thus, the average turnaround time is 116.33 s. This way of scheduling is called shortest remaining time next (SRTN) and it is the preemptive version of SJF. (See Fig. 5.7 for a comparison of SRTN and SJF for this example.)

» SRTN, the preemptive version of SJF, is overall optimal for average turnaround time assuming that the cost of context switch is zero.

SRTN has the same limitation as that of the SJF, namely, the scheduler must know the execution times of the processes in advance. Therefore, from the practical point of view it is hard to implement. However, it has the theoretical appeal that it assures the minimal average turnaround time.

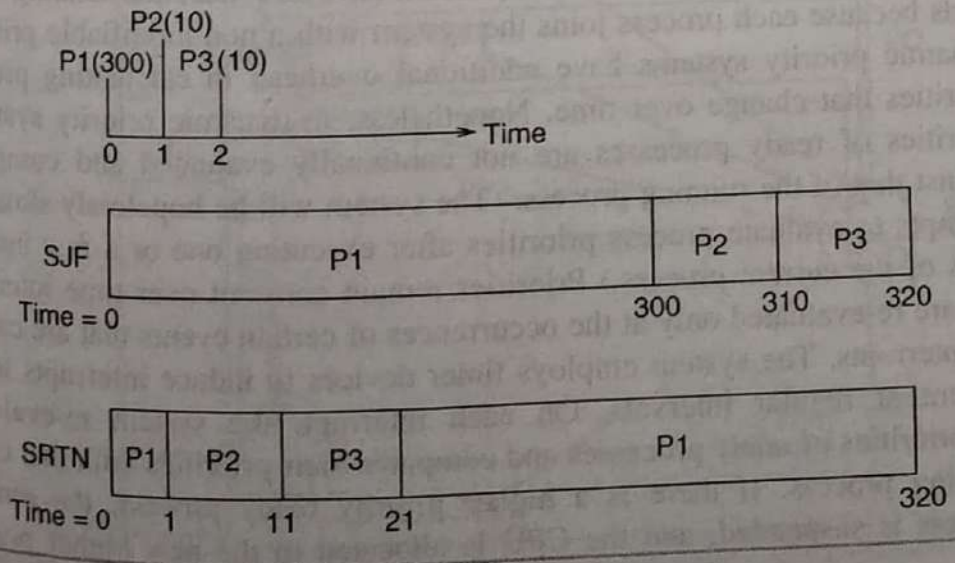


Figure 5.7: *A comparison of SRTN and SJF for the example in Section 5.7.3.*

5.7.4 Priority-based Scheduling

The three algorithms discussed above are in some sense priority-based algorithms even though we did not explicitly state this. In the case of FCFS, the priority of a process is determined based on the time-instant the process makes its request for a CPU. In SJF, execution time of the process is used to determine its priority: the lower the execution time, the higher the priority. In these two schemes, priorities are considered as static in that once a priority is assigned to a process it does not change thereafter. In contrast, in SRTN, the priorities are computed based on the CPU time required for each process at that point of time. Since this requires changes of execution timing over real-time for the processes, priorities are re-estimated and used at various points in time. Therefore, SRTN is a dynamic priority-based scheduling. These three are special cases of general priority-based algorithms.

» Most scheduling algorithms use some form of priority. There are two parts in priority-based scheduling algorithms: priority computation and scheduling decision. Once the priorities are computed, the scheduling decision becomes simple, enabling the algorithm to select a process with the highest priority for execution. Ties are brokenin FCFS order.

As observed in the above discussion, priority is the relative importance of processes for CPU allocation. In practice, the execution time requirements of processes may not be known in advance and process priorities are determined differently. The system makes its allocation decisions after determining the "priorities" at that moment of all the processes that are ready. Priority determines how quickly a process's demand for a CPU will be granted if other processes make competing demands. Each process is assigned a priority number for the purpose of CPU scheduling. The priority number is normally a nonnegative integer number, with lower numbers signifying higher priority or greater importance. Priority assignment however is not an integral part of a scheduler, and it is done by some other part(s) of the operating system. When needed, the scheduler consults those parts to determine the current priority level of a process.

» Priority determination is based on two factors: (1) external priority is based on administrative or business requirements, and (2) internal priority is based on resource usage.

In priority-based systems, the operating system assigns a priority to a process when the latter is created. In static priority systems, priority of a process does not change in its lifetime. In dynamic priority systems, the priority changes over time. When the system executes the scheduling algorithm for a CPU, the CPU is allocated to a ready process with the highest priority. FCFS discipline is followed to break any ties that may arise. The rule is that the running process is always a highest-priority process.

Static priority systems are simple and have low decision making overheads because each process joins the system with a non-modifiable priority. Dynamic priority systems have additional overhead in calculating process priorities that change over time. Nonetheless, in dynamic priority systems, priorities of ready processes are not continually evaluated and compared against that of the running process. (The system will be hopelessly slow if it attempts to evaluate process priorities after executing one or a few instructions of the current process.) Priorities remain constant over time intervals, and are re-evaluated only at the occurrences of certain events that are caused by interrupts. The system employs timer devices to induce interrupts in the system at regular intervals. On each interrupt, the system re-evaluates the priorities of ready processes and compares their priorities with that of the running process. If there is a higher priority ready process, the running process is suspended, and the CPU is allocated to th

» If the system is non-preemptive, then the new higher priority process has to wait until the current process releases the CPU.

process. Otherwise, the system continues executing the currently running process. If the system decides to suspend the running process, the process is put back into the ready queue.

Priority Implementation

To implement a priority-based CPU-scheduling scheme, the scheduler may maintain a separate queue for each priority level supported by the operating system. The scheduler iterates over priority queues, and always selects processes from higher priority queues before those from lower priority queues. For each priority level in the queuing model depicted in Fig. 5.8, there is a FCFS queue. A ready process joins at the end of the appropriate queue. That is, the ready queue is actually a logical concatenation of FCFS queues for the increasing priority levels.

Priority Aging

A priority-scheduling scheme favours higher priority processes. Consequently, it may induce process starvation that is caused by an indefinite delay in allocating CPUs to lower-priority processes. That is, higher priority processes may perpetually overtake a lower-priority ready process. Starvation reduces resource utilization as starved processes continue to occupy some of the resources without making progress in their computations. A technique called *aging* is superimposed on priority algorithms to avoid starvations. The scheduling algorithm does not change though. The operating system gradually increases priorities of processes that wait for too long a time in the ready queue. Thus, a low priority process will eventually achieve high priority status, and will get a CPU soon. However, when a process is allocated a CPU, the process falls back to its original (low) priority, because otherwise, there will be a priority inversion that would cause a low-priority process perpetually occupying the CPU.

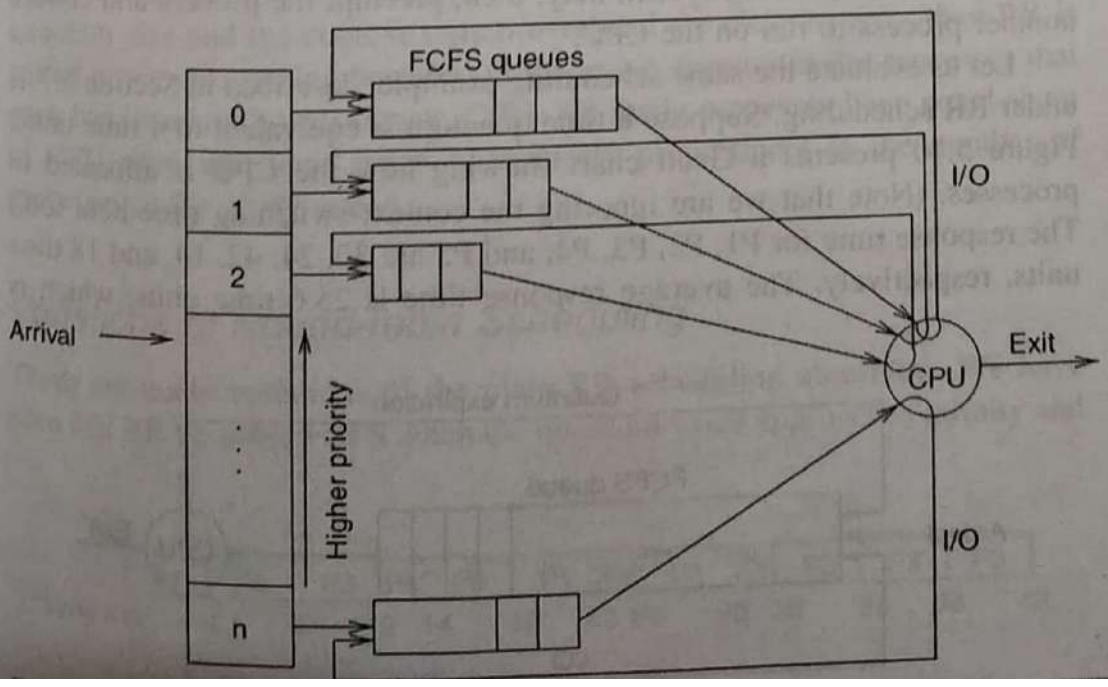


Figure 5.8: *The queuing model of static priority-queue-based scheduling.*

5.7.5 Round-robin (RR) Scheduling

The CPU-scheduling schemes presented above are not very effective for interactive systems, where users mostly submit short requests and expect quick responses back from the system. The main problem in those systems is with occasional long running processes. Once such a process is allocated a CPU, it monopolizes the CPU for an arbitrary span of time. The same also happens in both non-preemptive and priority-based scheduling schemes.

RR scheduling is specially designed for interactive time-sharing systems. It is a gross implementation of CPU multiplexing to ready processes. It ensures faster response time to shorter processes. It is a kind of preemptive FCFS scheduling discipline. A process that requests a CPU joins at the tail end of a FCFS ready queue. Like FCFS the CPUs are allocated to processes from the head end of the queue. However, unlike the pure FCFS discipline, a process may not get a CPU for an arbitrary length of time. Instead, when selected, a process can use the CPU for at the most a predetermined length of time called *time quantum* or *time slice*. No process can get a CPU for more than one time quantum in each selection. That is, a process, when selected, may get uninterrupted use of the CPU for at most a single quantum of time. If the process cannot complete its execution (or it does not execute a wait instruction) before the allocated time quantum expires, the process execution is interrupted and the CPU is preempted from the process. The process rejoins the ready queue at its tail end and waits there for its next turn. A queuing model of RR scheduling is shown in Fig. 5.9. A waiting process that becomes ready also joins at the tail end of the ready queue. In due course, when its turn comes up again, the same procedure is repeated until its completion.

» A time quantum is generally a small amount of time, in the order of tens of milliseconds. A quantum though is normally not a fixed value, and may vary with process priority, time of the day, or some other factor. How the quantum value is determined is a policy decision of the operating system, and not of the scheduler.

» As discussed in Section 2.2.3, a *timer* is a hardware device that can be set to interrupt the CPU after a specified length of time.

The RR scheduling needs special support from processor hardware. Such systems need a timer. When a CPU is assigned to a process, the timer is set to interrupt the CPU after a time quantum. If the process does not complete its execution before its allocated quantum expires, the timer will interrupt the execution. The operating system may, then, preempt the process and choose another process to run on the CPU.

Let us evaluate the same scheduling example, described in Section 5.7.1, under RR scheduling. Suppose a time quantum is equivalent to 4 time units. Figure 5.10 presents a Gantt chart showing how the CPU is allocated to processes. (Note that we are ignoring the context-switching time here too.) The response time for P1, P2, P3, P4, and P5 are 30, 24, 42, 14, and 18 time units, respectively. The average response time is 25.6 time units, which is

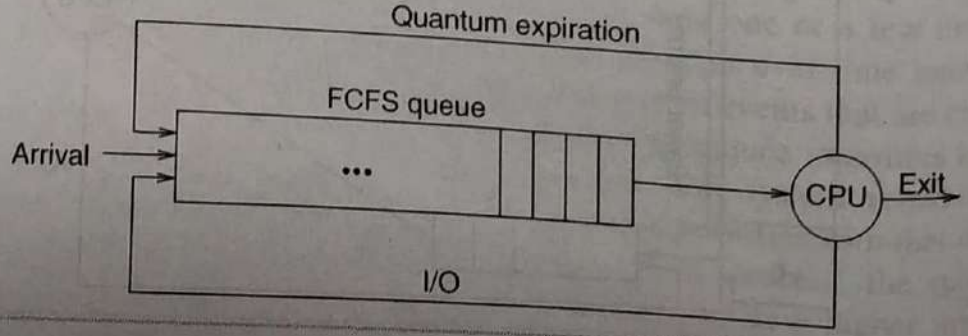


Figure 5.9: *The queuing model of round-robin scheduling.*

better than that (of 28.4) for FCFS scheduling. Nevertheless, RR scheduling leads to more context switches.

The main advantage of RR scheduling is its simplicity and freedom from starvation. It is one of the best-known scheduling policies for achieving a good- and relatively evenly distributed response time. It behaves uniformly and evenly with all processes provided they are alike in their execution patterns. For example, if all the processes are completely CPU-bound, the system behaves fairly with individual processes.

It is to be noted that it is preferable to have a balanced mixture of CPU-bound and I/O-bound processes in a system to optimize the utilization of both the CPUs and the I/O devices. Unfortunately, in such a mixed environment the system does not behave uniformly (evenly and fairly) with individual processes. In such a mixed environment, an I/O-bound process uses a CPU for a short period, releases the CPU, waits for I/O to be completed, joins back the ready queue at its tail end, and waits there for the next allocation of a CPU. On the contrary, a CPU-bound process uses a complete time quantum at almost every selection. In other words, CPU-bound processes monopolize the CPU, and I/O-bound processes receive unfair treatment. This results in poor performance of I/O-bound processes, poor utilization of I/O devices, and an increase in the variance of response times.

» When all processes have the same execution time requirement, RR may not be better than FCFS. In fact, the average response time will be worse in RR. Here is a small exercise that you may workout: five CPU-bound processes all require 10 units of execution time, and the time quantum is 2 units of time.

» An inherent weakness of the RR discipline is that it forgets the total amount of time a process spends on CPUs while making a scheduling decision.

The Effect of Time Quanta

The performance of RR scheduling depends on the size of time quantum. If the quantum value is too large, it behaves as if it is a FCFS scheduling algorithm. If the quantum value is too low, context switches are too frequent causing additional burden of workload on the CPU. Note that the context switch time is purely an overhead because in this time period no purposeful computation is carried out for any process. The usual range for the quantum value is between 20 to 50 times of the context switch time. If both the quantum size and the context switch overhead are close to zero, then RR is called *processor sharing* because, in theory, it appears to the processes that each has its own, though slower, CPU. All ready processes have equal share of CPU time, and their speed is inversely proportional to the number of processes in the ready queue.

Variations of Round-robin Scheduling

There are many variations of the plain RR scheduling algorithm. We have seen that RR becomes FCFS when the quantum value approaches infinity and

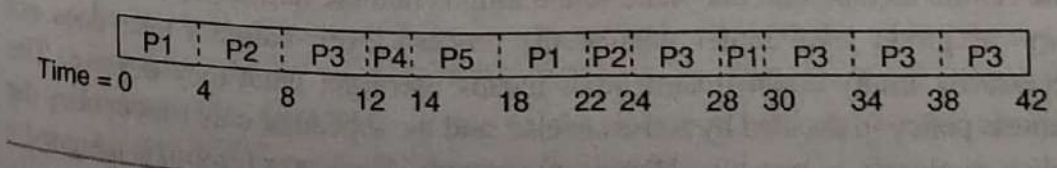


Figure 5.10: *The Gantt chart of RR scheduling of the example of Fig. 5.5 (with quantum = 4).*

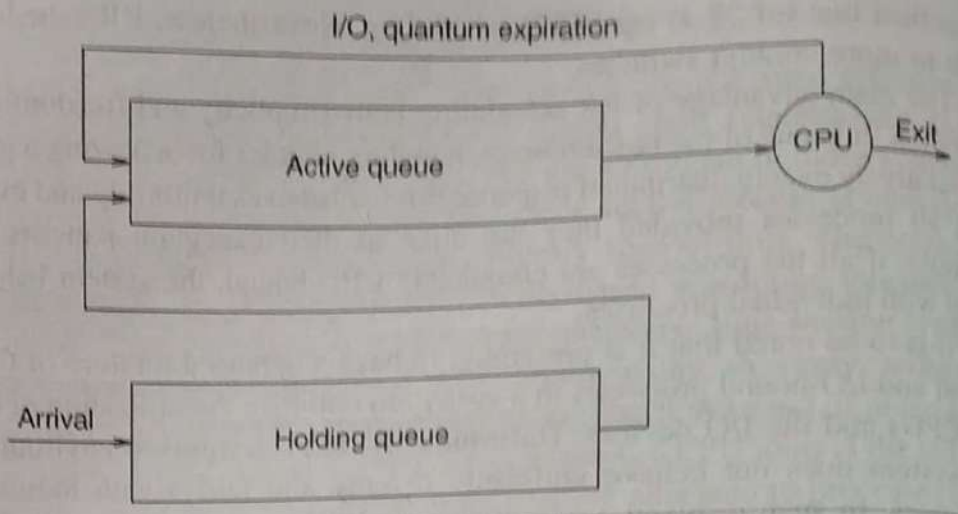


Figure 5.11: *The queuing model of selfish round-robin scheduling.*

it becomes processor-sharing when the quantum value approaches zero. We next introduce two more variations of RR called selfish round-robin and virtual round-robin.

SELFISH ROUND-ROBIN (SRR) SCHEDULING. In the plain RR, all the processes are treated equally when they are ready. SRR uses aging to gradually increase the priorities of processes over time. It uses two queues: the active queue and the holding queue. New processes enter the holding queue and reside there until their priorities reach the level of priorities of the processes in the active queue. At this point, they leave the holding queue and enter the active queue. The priority of a process increases at a rate a while in the holding queue, and at a rate b while in the active queue, where $a \geq b$. In general, SRR favours older processes over the processes that have just entered the system (see Fig. 5.11).

VIRTUAL ROUND-ROBIN (VRR) SCHEDULING. VRR also uses two queues: the auxiliary queue and the ready queue. The auxiliary queue is used to increase the fairness among CPU- and I/O-bound processes and it has a higher priority than the ready queue. In the plain RR, when a process goes for an I/O wait, its remaining quantum is ignored and it joins back at the end of the ready queue when it returns from the I/O wait (see Fig. 5.9). In VRR, a process returned from an I/O wait joins at the end of the auxiliary queue to use its remaining quantum before it returns to the ready queue (see Fig. 5.12). This way, processes get a little boost every time they return from an I/O wait. When the quantum eventually expires, the process rejoins at the end of the ready queue.

5.7.6 Fair-share Scheduling

The central idea behind fair-share scheduling is that the fairness is elevated at the user- or group level rather than at the process level. Fairness here does not necessarily imply equal treatment; it means users get what they deserve. The fairness policy in decided by someone else, and the scheduler only tries to obey the policy as closely as possible. Here is an example. Each user (group) is assigned a

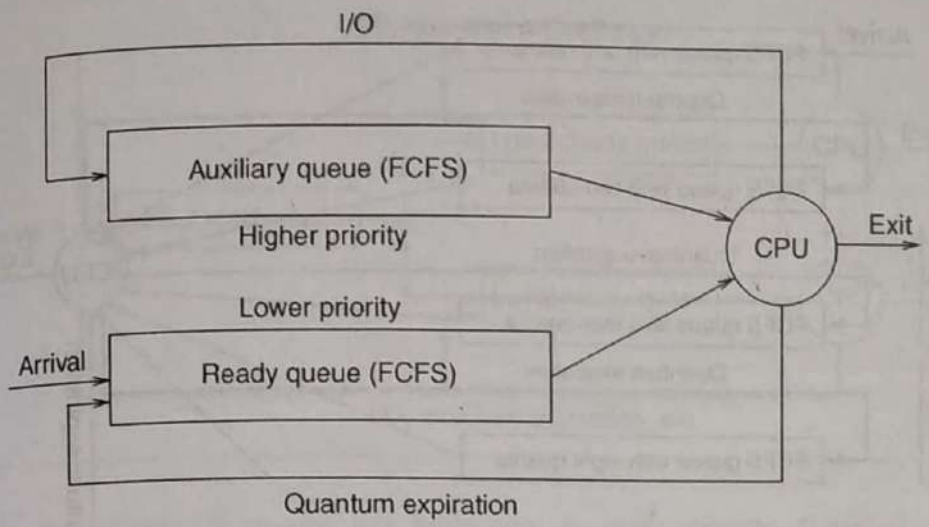


Figure 5.12: *The queuing model of virtual round-robin scheduling.*

share of the CPU and this share is then distributed among the processes of that user (group). Here is a simple example. Assume that user *A* has two processes and user *B* has four. In fair-share scheduling, users *A* and *B* get the same share while a process of *A* gets twice the share of CPU time compared to a process of *B*. RR is applied within the processes belonging to same user (group). Most practical scheduling algorithms use priority to classify groups and then use either RR or FCFS within each group.

5.7.7 Multilevel or Multiple-queue-based Scheduling

The aim of multilevel scheduling is to use multiple priority queues to hold processes. The processes are always scheduled from the non-empty queue with the highest priority. Within each queue, either RR or FCFS is used. (Interactive applications prefer RR, while some real-time applications prefer FCFS.) Multilevel scheduling is the most generic scheduling strategy. There are many possible ways to implement such scheduling. Some representative implementations are as follows.

1. Each process joins a fixed queue based on its initial priority and stays there until completion. This produces multilevel scheduling algorithms with static priorities (see Fig. 5.8 on page 123).
2. Each process initially joins the highest priority queue and moves down after it completes its execution in that queue for a certain amount of time. Allocation of processing time in each queue is determined based on the time quantum value. The processes always move downwards, and the quantum size for each queue is fixed and increases downwards (see Fig. 5.13; it only shows CPU-bound process flow). In the figure, processes in the highest priority queue run for one quantum, processes in the next queue run for two quanta, processes in the next queue run for four quanta, and so forth. Whenever a process uses up all the quanta allocated to it, it moves down one level. The lowest level is always FCFS without forwarding quantum.

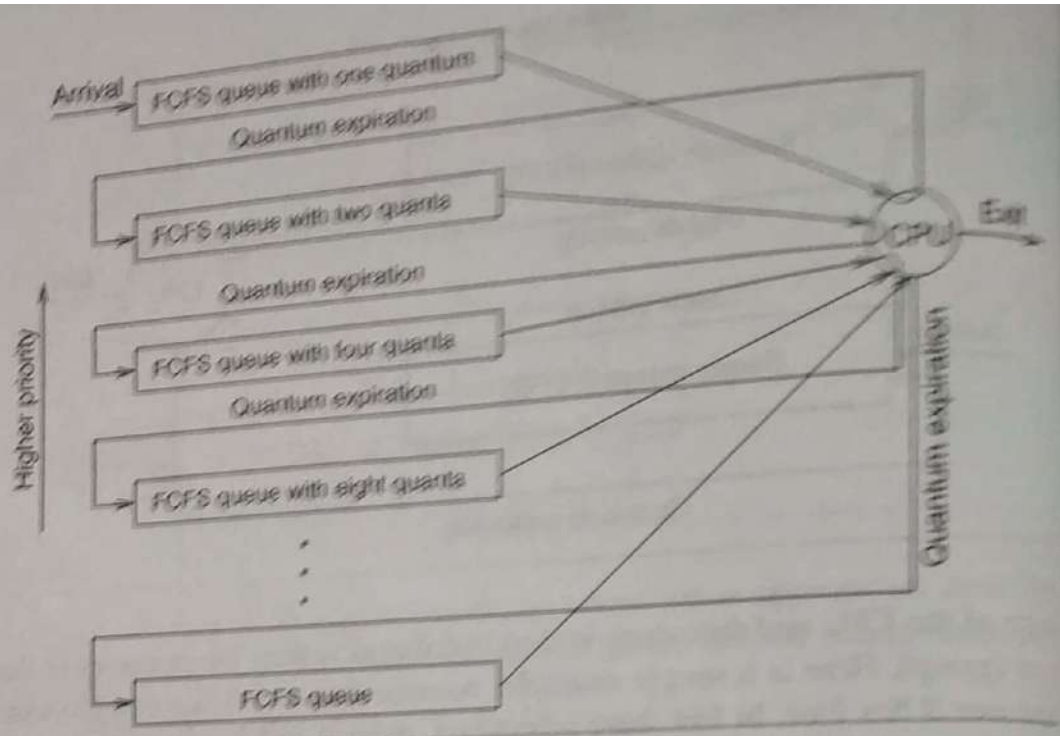


Figure 5.13: *Multilevel scheduling with progressive priority degradation.*

3. Like static priority queuing, processes may join any priority queue. However, they can move up or down depending on their priority recalculation. For example, a process returning from an I/O operation gets its priority boosted and, therefore, moves up, and when it has finished its quantum it moves down. Such a technique where the processes move up or down in the priority queues is called *feedback*.

The highest priority queue normally holds the kernel- or system server processes. The next priority level is allocated to real-time processes while other interactive processes are at lower levels on the priority list. Normally, background processes and processes from low-end users are assigned lower priorities.

» We would like to emphasize the word 'feedback'. A multilevel queue without feedback is the static priority system (see Fig. 5.8 on page 123). With forward feedback, it is one-way flow of processes in the ready queue. Their priorities are progressively lowered (see Fig. 5.13). This may cause starvation for long processes. We need a feedback mechanism to increase process priorities too to avoid starvation. We need a two-way flow of processes. Linux 2.6 and Windows XP use multiple level priority queues with feedback. They also support higher priority real-time processes and low priority normal processes. At each priority level, the service discipline can be exclusively RR, or FCFS, or a mix of the two.

5.7.8 Multiprocessor Scheduling

CPU scheduling in (symmetric) multiprocessor systems is not much different from that in uniprocessor systems. Normally a single global ready queue is shared by all CPUs. In addition, to avoid high contention on the global ready queue, there might be an individual/local ready queue for each CPU. Processes from the global ready queue are distributed (by way of balancing workload on CPUs) to local queues, as shown in Fig. 5.14. Not shown in the figure, a process that requests a CPU may have an affinity to a particular or a few CPUs; in that case, the process is scheduled only to those CPUs. For example, if a process accesses a device that is connected to a particular CPU, then the process is said to have an affinity to that CPU.

5.7.9 Thread Scheduling

Threads are becoming a necessary component of modern operating systems. As mentioned in Chapter 4, a thread is an execution flow of a part of a sequential

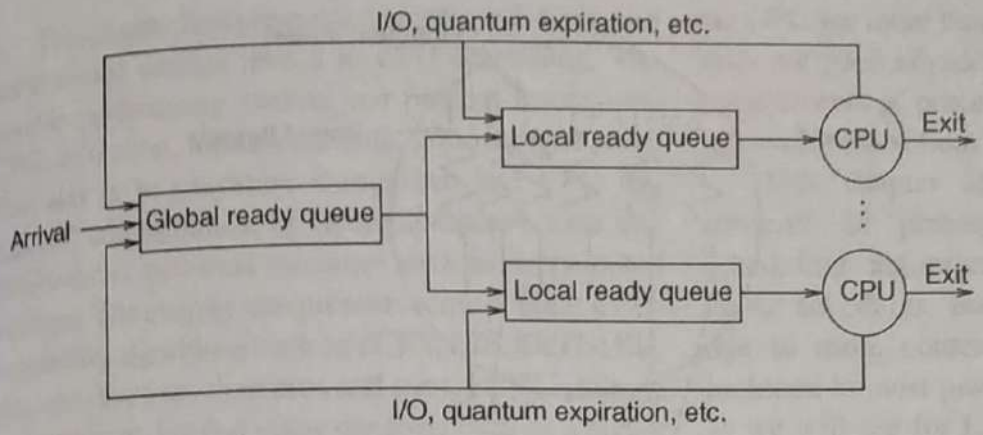


Figure 5.14: *A typical queuing model in multiprocessor systems.*

code of a process. A process can have one- or more threads. Creating- and synchronizing threads are the responsibilities of the applications. Typically, threads are scheduled in two ways for execution: (1) the CPU is directly assigned to threads (one-level scheduling, fundamentally the same as traditional CPU scheduling), and (2) the CPU is first assigned to processes in the system and then the threads within each process share the CPU for the time allocated for that process (two-level scheduling in which each process has its own thread-scheduling scheme). Thread management and scheduling is done either at application level (user-level threads) in the form of including suitable thread libraries or at the operating system level (kernel-level threads).

In user-level thread scheduling (as shown in Fig. 5.15), once an application process acquires the CPU, the threads in the process are scheduled using its own scheduling policy until the CPU is taken away from that process or there are no ready threads. Usually, no process preemption is necessary during thread scheduling within a process because only another thread of the same process will replace a preempted thread. User-level thread scheduling has the advantage of having flexible scheduling policies specific to applications and does not have to incur the cost of trapping to the kernel space during thread switching. This scheduling flexibility can be achieved to some extent in

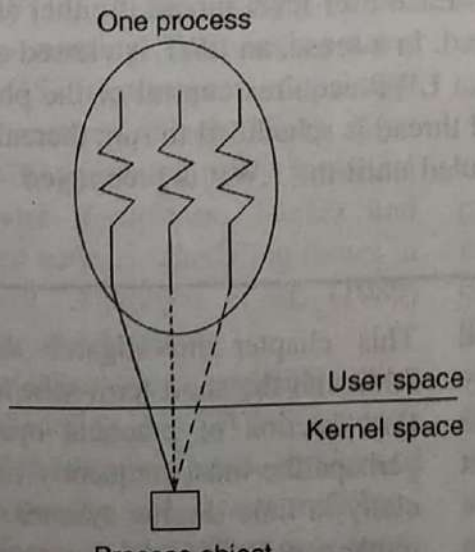


Figure 5.15: *Two-level thread scheduling.*

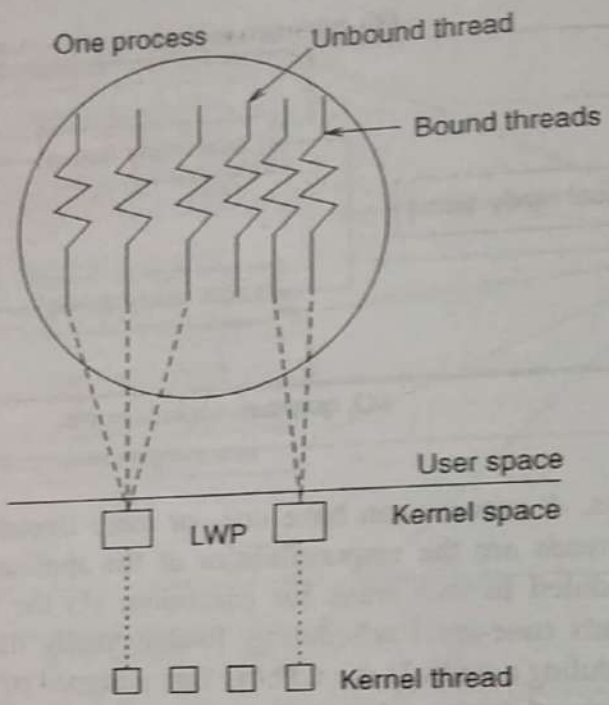


Figure 5.16: *Thread scheduling in Solaris 2.*

kernel-level thread implementations by defining interfaces that allow each application to select its thread-scheduling policy. Again two-level thread scheduling can be considered merely as a logical scheduling routine and it can be actually implemented using one scheduler within the operating system or two schedulers—one within the application and another within the operating system. In either case, at the most one thread of a process can be in the running state.

Solaris 2 supports both user level and kernel level threads (see Fig. 5.16). As discussed in Section 4.12.5, middle level abstraction called lightweight process (LWP) is employed to schedule user level threads. LWP is manipulated through the library that is used to create and manage user level threads. The kernel-level scheduler schedules kernel-level threads and each LWP is associated with a unique kernel-level thread. Some kernel-level threads are designed only to run on behalf of the kernel and, therefore, they do not have any associated LWP. Each user-level thread is either associated with at least one LWP or is blocked. In a sense, an LWP is viewed as a "logical CPU" for user-level threads. An LWP acquires control of the physical CPU when the corresponding kernel thread is scheduled to run; thereafter, the threads under its control are scheduled until the LWP is preempted.

Summary

Scheduling algorithms, although simple compared to the other components of an operating system, play a crucial role in the operating system. In some practical systems, scheduling decisions are made at two levels: (1) long-term scheduling controls the degree of multiprocessing; and (2) short-term scheduling allocates CPUs to competing processes. This chapter investigates short-term scheduling. Although the short-term-scheduling code occupies a tiny fraction of practical operating systems, it is perhaps the most frequently executed routine, especially in time-sharing systems. It is the only way to implement multiprocess systems by multiplexing the CPU to proces-

CHAPTER

7 Process Synchronization

Learning Objectives

After reading this chapter, you should be able to:

- Explain process synchronization and the need for it.
- Describe some fundamental synchronization problems.
- Discuss some useful synchronization tools.
- Describe solutions to synchronization problems.
- Explain deadlocks and their solutions.

7.1 Introduction

Processes in a computer system execute programs to manipulate data. One process may or may not interact with another process to accomplish its task, and as explained in Section 6.3.3 such interactions take place only through shared data. If two processes use two distinct sets of data items, then they are not considered interacting processes and their activities do not affect each other. Their behaviours are independent and do not influence each other. Whereas, if a process A modifies a data item that another process B reads, then the behaviour of B may depend on activities of A. Here we say the two processes are interacting with one another through shared data. The behaviour of these interacting processes then depend on two factors: the relative speeds of the processes, what they do with shared data and how they access the shared data. If the activities of interacting processes are not controlled suitably, their behaviour may not be as "consistent" as expected from their specifications.

» The operating system or any application execution may be considered to be a collection of processes in which some interact and some others do not.

The theme of this chapter is the coordination of interacting processes, that is, the orderly execution of their accesses to shared data. Unlike interprocess communication (discussed in Chapter 6), process interaction is somewhat indirect. In many situations, a process may complete its own execution even if other processes are absent. Processes in general are independent, but they

» Synchronization is the single most important topic in highly concurrent systems such as operating systems, databases, and networks.

with that interrupt to handle it. Usually, the device drivers are the interrupt handlers and are kept in the user space.

15.5.4 The Monolithic-kernel-based Model

No standards exist for this model. Any design not following the above philosophies may be considered under this model. Typically, added functionalities in this model are the following:

- sophisticated CPU scheduling;
- solutions for priority inversion issues;
- improved memory management—allocation and protection (isolating the applications software from the kernel by supporting the two operating modes, and adding an isolation barrier between the individual tasks);
- file handling;
- graphics handling; and
- networking.

» Despite the appeal and advantages of the microkernel paradigm, it was not widely accepted until recently. Further, most of the earlier microkernels were evolved from monolithic kernels and, therefore, did not have many essential characteristics of the real microkernel.

Generally, an operating system may be considered as its kernel with the remaining part. In this model, since there is no separation within the operating system as kernel and the remaining part, the entire operating system may be viewed as a monolithic structure and hence referred to as *monolithic kernel based systems*. RT-Linux is a monolithic kernel based real-time operating system and Linux kernels are highly portable and easily configurable.

In summary, multitasking and preeemption, predictable performance and synchronization, support for a range of priority levels and priority determination, and bounded latency on task switching and interrupt handling are some of the basic operating system requirements for real-time systems.

With this introduction to REOS, we next focus on individual topics such as CPU scheduling, task synchronization, memory management, and file systems.

» Real-time software designers generally perceive separate kernel spaces and separate process address spaces as disadvantageous for time-critical applications.

15.6 CPU Scheduling

Real-time task scheduling is one of the interesting topics and is deeply studied in the context of real-time systems. The interest in the topic started with the seminal work of Liu and Layland in 1973. Since then, many algorithms have been proposed for real-time scheduling. Recently, the field is receiving renewed interest due to the pervasiveness of embedded devices in the consumer market and the advancement of technological innovations.

Real-time task scheduling is an important responsibility of real-time systems. Examples of real-time tasks include control of temperature in a chemical plant, collecting readings from sensor nodes periodically, monitoring systems for nuclear reactors, etc. Based on their importance, real-time

» Priority is very important in REOSes, and nothing should prevent the execution of the highest priority tasks in the system.

tasks are usually prioritized. Since timely execution of these tasks is of paramount importance, the real-time CPU scheduler must make sure that the tasks meet their deadlines. (A *deadline* is a relative or an absolute time by when a task must complete.) Schedulability analysis is a fundamental aspect of real-time scheduling. That is, checking whether a set of given tasks can be executed in the system without missing the deadlines is crucial for many life critical systems before the tasks are actually scheduled in the system. A set of tasks is said to be *schedulable* if enough CPU time is available to execute all these tasks before their deadlines. For this reason, scheduling in real-time systems is not the same as scheduling in traditional operating systems.

Each real-time task is assigned a *priority* and a *deadline*. Also, most real-time tasks are executed *periodically*. Sometimes, execution priorities are derived from deadlines and/or periods. Most real-time scheduling algorithms are "priority-based preemptive scheduling". This scheme allows only the highest priority task among the ready tasks to run at any moment. When a task with priority higher than currently running task becomes ready, then the current task is preempted and the new higher priority task is allowed to run immediately. The crux of these classes of algorithms is how these task priorities are determined and when. Accordingly, real-time scheduling algorithms can be classified into two categories: *fixed (static) priority algorithms* and *dynamic priority algorithms*.

» Ignoring context switch cost in task scheduling is not appropriate for most modern systems. Real-time scheduling algorithms are generally preemptive, and preemption introduces the context switch. In addition to the context switch, preemption also involves activities such as processing interrupts, manipulating task queues, etc. This cost is significantly high if the system uses caches in single- or multi-levels—and cache memory is used in almost all systems today.

Earlier studies on real-time scheduling assumed a simple but powerful model in which tasks are assumed to be periodic. For example, a task is designed to read a temperature sensor value in a plant every 50 seconds or scan a security area every 10 seconds. These tasks are activated periodically to complete their missions. If a task's relative activation time (period) is not known then it is a non-periodic task. If a non-periodic task is either soft or has no deadline then it is called *aperiodic task*. A non-periodic task with a hard deadline is called *sporadic task*. We start with scheduling of periodic tasks.

15.6.1 Scheduling Periodic Tasks

The periodic task model is the simplest but sufficient for many applications. The following assumptions are made for this system:

1. All tasks run periodically on a single CPU. Deadlines are at the end of their period. That is, for a period p, the deadlines are at p, $2p$, $3p$, ...
2. The tasks are independent.
3. The execution time for each task is fixed.
4. The context switch time is ignored.

» RM and EDF algorithms are widely studied and extensively analyzed. Surprisingly, almost all other algorithms proposed in literature later are only variations of these two basic algorithms.

Liu and Layland introduced two real-time scheduling algorithms, called rate monotonic (RM) and earliest deadline first (EDF), in 1973. RM is a fixed priority scheduling algorithm which assigns higher priorities to tasks with shorter periods. EDF is a dynamic priority scheduling algorithm which assigns higher priorities to tasks with the current earliest deadline. It is easy to see that RM and EDF are simple, and they are proved to be optimal in their respective classes.

RM is used for most practical applications. The reasons for favouring RM over EDF are based on the beliefs that RM is easier to implement, introduces lesser runtime overhead, is easier to analyze, is more predictable in overloaded conditions, and has lesser jitter in task execution; the variation in task execution delays is called *jitter*.

» If jitter occurs, many undesirable consequences may result in the system. So, reducing jitter is one objective of real-time scheduling.

Priority Scheduling Algorithm

The objective of priority scheduling algorithms is simply that:

- at any time, the scheduler selects the highest priority ready task; and
- the selected task runs until either it completes its execution for that period or another task with priority higher than it becomes ready for execution.

An implementation scheme for a priority scheduler is described as follows. The scheduler maintains essentially two queues: *ready queue* and *wait queue*. The ready queue contains tasks which are ready to run and the wait queue contains tasks that have already run and are waiting for their next period to start again. The ready queue is ordered by task priority and the wait queue is ordered by the earliest start time. When the scheduler is invoked, it examines tasks in the wait queue to see if any task should be moved to the ready queue at that point of time. Then it compares the head task at the ready queue to currently running task. If the priority of the head task is higher than that of the running task, then the scheduler invokes a context switch. The scheduler is invoked by an interrupt from either an external event or a timer. The start time of a task might trigger a timer interrupt. Other efficient implementations are possible.

» In fixed priority scheduling, task priorities are assumed to be fixed throughout the execution. Dynamic priority scheduling computes the priorities during runtime.

Rate Monotonic Scheduling Algorithm

The crux of the RM scheduling algorithm lies in the way the priorities are computed. RM computes priorities based on task periods. A task with a shorter period has higher priority. That is, the task with the higher rate of occurrences has higher priority, hence the name *rate monotonic*. Since the period of each task is fixed, once their priorities are computed and assigned in advance, the priorities stay static. For the following discussion, we use the notation $T(e,p)$ to denote a task T with execution requirement of e time units and period p time units.

Consider three periodic tasks T1(1,4), T2(2,5), and T3(3,10), as shown in Fig. 15.1. The down arrow indicates both the ending of the previous period and starting of the new period. Tasks are activated at every starting period. According to RM, T1 has the highest rate of 1/4 and, therefore, has the highest priority; T2 has the next highest rate of 1/5 and, therefore, has the next higher priority; and T3 has the lowest rate of 1/10 and hence has the lowest priority. All three tasks [illegible] execution at time 0.

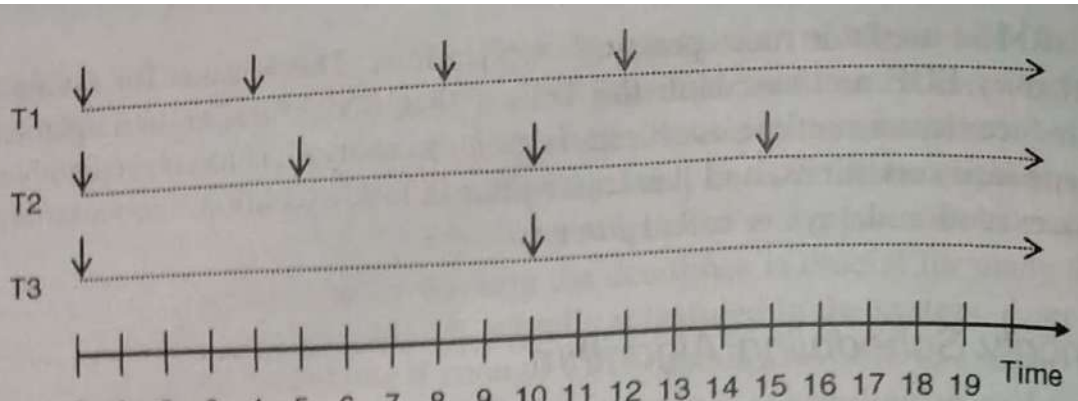


Figure 15.1: *Example Task Set 1.*

At time 0, since T1 has the highest priority, it starts its execution and completes at time 1. At time 1, T2, the currently highest priority ready task, can start its execution and completes it at time 3. Now, at time 3, T3 can start its execution, but at time 4 the highest priority task T1 will be again ready and that will preempt T3, and run until time 5. Since T2 arrives at time 5, T3 has to wait until T2 completes its execution at time 7. Now, T3 can have 1 unit of execution at time 7 before T1 arrives at time 8. Then after T1 finishes at time 9, T3 can complete its execution for the first period at time 10. In this example, T3 is preempted twice in its first period, but eventually completes its execution on time. Consider another example.

Three periodic tasks are T1(2,4), T2(2,6), and T3(3,12), as shown in Fig. 15.2. According to RM, T1 has the highest priority; T2 has the next highest priority; and the task T3 has the lowest priority. All three tasks are ready for execution at time 0.

Now, in the time interval 0 to 12, T1 must execute 3 times, T2 must execute 2 times, and T3 must execute 1 time. So we need $6 + 4 + 3 = 13$ units of execution in 12 units of time, which is not possible. Another related metric is utilization that is computed as the sum of execution ratios of the tasks. That is, in the above example, utilization is $2/4 + 2/6 + 3/12 = 13/12$. If the utilization is above 100 per cent, the tasks cannot be scheduled to meet deadlines. Therefore, these three tasks are not schedulable and some task, naturally the lowest priority task, cannot meet its deadline. Thus, the utilization has to be less than or equal to 100 per cent. However, having less than 100 per cent utilization does not imply that the task set is schedulable. This is a necessary condition in theory, but not a sufficient condition. It has to be verified practically.

For a given task set, the least common multiplier, aka, LCM, of all the task periods is called the *hyperperiod* of the task set. For the task set shown in Fig. 15.2, the hyperperiod is 12. Since all tasks will have their start times simultaneously at the beginning of the next hyperperiod, the schedule just repeats itself in each hyperperiod. Therefore, it is enough to verify task schedulability for one hyperperiod. Consider the following example:

Three periodic tasks are T1(1,3), T2(1,4), and T3(2,5). According to RM, T1 has the highest priority; T2 has the next highest priority; and T3 has the lowest priority. All three tasks are ready for execution at time 0.

The hyperperiod for Example Task Set 3, illustrated in Fig. 15.3, is 60, and task utilization $1/3 + 1/4 + 1/5 = 0.9833$. Although the utilization is

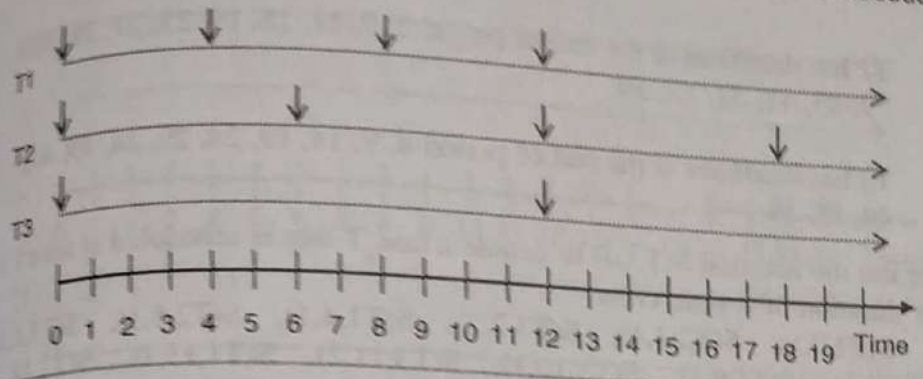


Figure 15.2: *Example Task Set 2.*

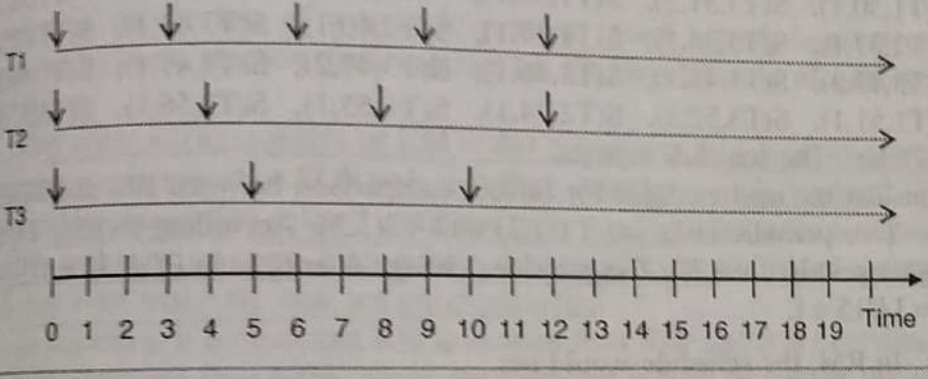


Figure 15.3: *Example Task Set 3.*

98.33 per cent, which is less than 100 per cent, this task set is not schedulable under RM. Tasks T1 and T2 take time intervals 0–1 and 1–2, respectively, and task T3 the time interval 2–3, then T1 and T2 take again take time intervals 3–4 and 4–5, respectively. Now, T3, having executed for only 1 unit of time, misses its deadline (it has to complete execution for 2 units of time before time 5) for the first period.

Earliest-deadline-first Scheduling Algorithm

The purpose of EDF is to assign priorities to tasks dynamically, based on the current order of their deadlines. The highest-priority task is the one whose deadline is earliest. Clearly, the priorities must be recalculated at every scheduling point. Similar to RM, at any moment, the highest priority task is executed, but the priority of the task changes over time. We have shown that Example Task Set 3 is not schedulable under RM. This task set is however schedulable under EDF as shown below.

At any point of time, EDF schedules the task with the earliest deadline. When two or more tasks have the same deadline, one task is chosen randomly for scheduling. The hyperperiod (i.e., the LCM of 3, 4, and 5) for this example is 60.

T1 has deadlines at the end of period 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59

» Processor utilization (U) for any feasible schedule under RM generally decreases as the number of tasks (n) increases ($U = n(2^{1/n} - 1)$). When the number of tasks increases, the CPU utilization reaches to about 69 per cent. The theoretical limit of 69 per cent indicates that RM could waste as much as about 31 per cent of CPU time. The next algorithm could bring down the CPU wastage time close to 0 per cent.

T2 has deadlines at the end of period 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59

T3 has deadlines at the end of period 4, 9, 14, 19, 24, 29, 34, 39, 44, 49, 55, 59

We use the notation S(T,t,d) to denote a task T that is scheduled at time t for the duration of *d* time units.

S(T1,0,1), S(T2,1,1), S(T3,2,2), S(T1,4,1), S(T2,5,1), S(T1,6,1), S(T3,7,2), S(T1,9,1), S(T2,10,1), S(T3,11,2), S(T1,13,1), S(T2,14,1), S(T1,15,1), S(T2,16,1), S(T3,17,2), S(T1,19,1), S(T2,20,1), S(T1,21,1), S(T3,22,2), S(T1,24,1), S(T2,25,1), S(T3,26,2), S(T1,28,1), S(T2,29,1), S(T1,30,1), S(T3,31,2), S(T1,33,1), S(T2,34,1), S(T3,35,1), S(T1,36,1), S(T2,37,1), S(T3,38,1), S(T1,39,1), S(T2,40,1), S(T3,41,1), S(T1,42,1), S(T3,43,2), S(T1,45,1), S(T2,46,1), S(T3,47,2), S(T1,49,1), S(T2,50,1), S(T1,51,1), S(T3,52,2), S(T2,54,1), S(T1,55,1), S(T2,56,1), S(T3,57,2), (59 idle). The schedule repeats.

Consider the next example for further comparison between RM and EDF.

Two periodic tasks are T1(1,3) and T2(3,5). According to RM, T1 has higher priority than T2. The priorities change over time in EDF. $U = 1/3 + 3/5 = 14/15 < 1$.

In RM, the schedule would be:

S(T1,0,1), S(T2,1,2), S(T1,3,1), S(T2,4,1), S(T2,5,1), S(T1,6,1), S(T2,7,2), S(T1,9,1), S(T2,10,2), S(T1,12,1), S(T2,13,1), ...

In EDF, at time 0, the priority of T1 is higher than that of T2. The scheduler runs T1 in interval 0-1, runs T2 in interval 1-3. At this point T1 comes back. Since task T1 at this time has a deadline (at time 6) that is later than the deadline of task T2 (at time 5), it would be more appropriate to continue T2. In addition, this would reduce one preemption. So the schedule for EDF would be:

S(T1,0,1), S(T2,1,3), S(T1,4,1), S(T2,5,1), S(T1,6,1), S(T2,7,2), S(T1,9,1), S(T2,10,3), S(T1,13,1), ...

The schedules are depicted in Fig. 15.4, and we can see that the time interval 14-15 is free and the schedule will repeat itself starting from time 15, the beginning of the next hyperperiod.

RM vs EDF

Traditionally RM has been favoured over EDF. Recently, the validity of some of these acclaimed attractive properties of RM have been questioned. In addition, it is observed that most of the advantages of RM over EDF considered in literature are either very slim or incorrect when the algorithms are compared with respect to their development from scratch rather than

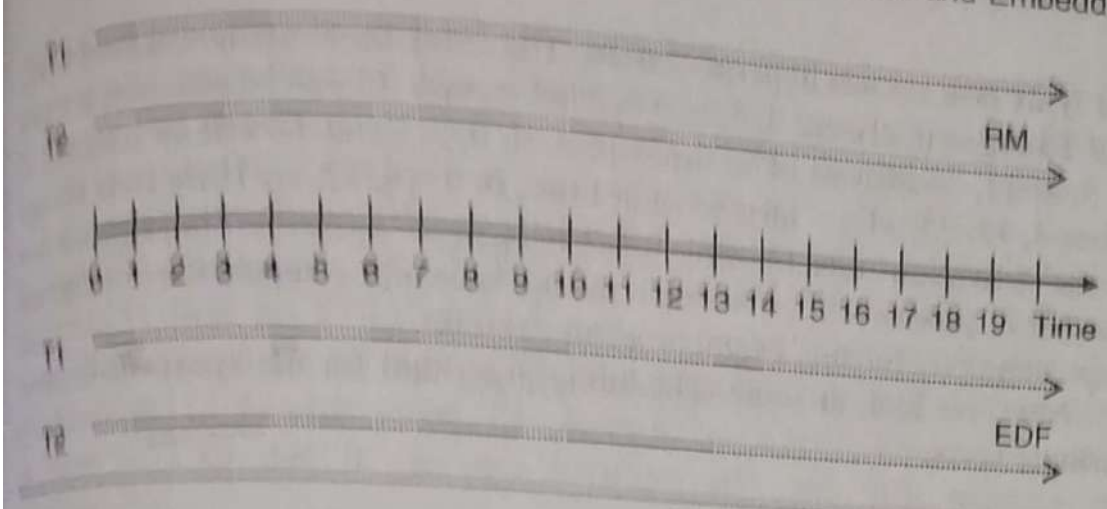


Figure 15.4: *Task schedules under RM and EDF.*

developing on the top of a generic priority based kernel. Some recent operating systems provide such support for the development of user level schedulers.

One unattractive property of RM is that it experiences a large number of preemptions compared to EDF and, therefore, introduces high overhead. The undesirable preemption-related overhead may cause higher processor overhead in real-time systems, high energy consumption in embedded systems, and may even make the task set unschedulable.

In summary, it is clear that RM is simpler than EDF and RM experiences more preemptions than EDF. Now, the question is how to reduce the preemptions without losing the simplicity of RM. The next algorithm is directed towards answering this question.

» As advanced architectures with multi-level caches and multi-level context switch (MLC) are becoming increasingly common, the continued use of the popular scheduling algorithms such as RM are likely to experience cascading effect on preemptions.

Activation Adjusted Scheduling Algorithm

Preemption of tasks occurs when a higher priority task is activated during the execution of a lower priority task. Consequently, a lower priority task would experience more preemptions as it stays longer in the ready queue. Therefore, to reduce preemptions, it is necessary to reduce the lifetime of lower priority tasks waiting in the ready queue.

One way to reduce the lifetime of lower priority tasks, if possible, is to delay the activation of higher priority tasks. This would increase the chance of lower priority tasks using the CPU as much as they can and complete their executions quicker. The activation delays can be computed offline similar to computing worst case response time and incorporated in the periods to arrive at the "adjusted-activation" times. The actual computation of delay times is beyond the scope of this book. Once the delays are computed offline and activation times are adjusted, the remaining actions of the algorithm coincide with RM. This is the objective of the activation-adjusted scheduling algorithm. We illustrate this idea using the following example. Three periodic tasks are T1(1,3), T2(3,9), and T3(2,12). According to RM, the priority order, from highest to lowest, is T1, T2, and T3.

- For CP and PC, the priority ceiling is equal to the highest priority of all tasks requiring R. The difference is in allowing- or denying access to R.
- For MB-CP, the priority ceiling is equal to the priority ceiling of the monitor, which contains the critical section of R.
- Assume that a task A holds R. In PI, whenever a higher priority task B requests R, A inherits the priority of B and B is blocked.

15.8 Memory Management

Real-time systems require predictable memory access, and random delays must be avoided as much as possible. Since timing requirements vary for different applications, no single memory management technique suits all real-time systems. Here we look at general issues related to memory management and then discuss specific memory management strategies to use in real-time systems.

A typical task has codes, data, stacks, and working area as its components. In embedded systems, important stable codes and data are normally kept in EEPROM or flash. The rest of the codes and data are stored in the RAM. Memory management issues are related to managing the RAM. Tasks need the RAM for various purposes and the problem is how to serve requests for the RAM without causing random delays in the system. There are two approaches—*static* and *dynamic*—for memory management in real-time systems.

15.8.1 Static Memory Allocation

Static memory-allocation technique is the simplest in that it allocates the necessary memory to each task before its execution starts. Then each task holds the allocated memory until it terminates. This technique has many disadvantages. First, it is not efficient in terms of memory usage. We analyze three situations.

First, tasks rarely use all the allocated memories all the time during their lifetime; so, a large portion of allocated memory could remain unused most of the time. Sometimes tasks may request memories, intentionally or unintentionally, that they never use. This type of allocated but unused memories may be referred to as "leakage memories". Memory leakage reduces the amount of free memory available in the system. Secondly, a task might require memory larger than the size of RAM during its lifetime but at any particular interval, it may require only a small amount of memory. Such tasks cannot be served by this scheme. Lastly, released and unallocated memories may be fragmented

between allocated memories. These memories cannot be allocated when the requested memory size is larger than individual fragments.

15.8.2 Dynamic Memory Allocation

Most modern computer systems follow dynamic memory allocation. It increases the memory usage, but still suffers from fragmentation and memory leakage problems. The memory is managed as a heap. When a memory is returned to the heap, it may be broken into smaller fragments when reallocated in the future. Over a period, memory segments become smaller and smaller to a level where they are rarely useful. At this stage, even though enough free memory is available in the system, they cannot be allocated for use because they are fragmented and scattered. Fragmentation problem can be solved by *defragmention techniques*—moving the free memory to one side of the RAM and the allocated memory to the other side. This is a time-consuming complex task. It may cause random delay.

» Allocating memory to a task and then failing to reclaim it when not needed any more is called memory leakage.

Memory leakage problems are solved by a technique called *garbage collection*. Again, garbage collection may cause random delay in the system. Generally, when the available free memory reaches a critical stage, garbage collection or defragmentation is initiated to remedy the situation. As indicated earlier, such activities create uncertain delay, which is not suitable for time critical systems.

We discuss two general approaches to dynamic memory management in time-critical systems. The first approach is invoking the garbage collection or defragmentation algorithms more frequently and running them for fixed intervals in turns. Normally, they are invoked only when the available free memory reaches the critical stage and run for an unknown period for completion. The alternative is to invoke the algorithms frequently, but allow them to run for small intervals on each invocation. By putting time bounds on each run, we could greatly alleviate the random delay problem.

The second approach uses a non-fragmenting memory allocation technique instead of heaps. In this approach, the available sizes of memory blocks and the number of blocks for each size are determined initially. Same-size memory blocks are pooled together. Memory blocks are allocated from these pools based on the requirement. When freed, they are put back into their appropriate pools for future re-use at their original sizes. This alleviates the external fragmentation problem. When the requested size does not match any of the available sizes, then the next available size of memory is allocated. This, of course, wastes the extra memory allocated—called *internal fragmentation*. However, the advantage of deterministic, often, constant, timing in allocating and deallocating memory from the pools of fixed-size memory blocks outweighs the internal fragmentation problem.

In summary, the key objective of memory management in real-time systems must be to avoid, wherever possible, any strategy such as dynamic memory management that might create random delays. For time-constrained life-critical applications, it must be completely avoided. Most real-time systems do not use virtual memory.