CHAPTER 1

OVERVIEW

1.1 INTRODUCTION

The use of a bare hardware machine is cumbersome and inefficient because a large number of chores must be manually performed, such as entering programs and data at appropriate locations in the main memory, addressing and activating appropriate input-output devices, etc. When a machine is used by several users simultaneously, numerous other issues arise, such as the protection of user data and the time- and space-multiplexing of shared resources among them. An operating system relieves users of these cumbersome chores and increases efficiency by managing the system's resources.

1.2 FUNCTIONS OF AN OPERATING SYSTEM

An operating system is a layer of software on a bare hardware machine that performs two basic functions:

1 **Resource management.** A user program accesses several hardware and software resources during its execution. Examples of resources are the CPU, main memory, input-output devices, and various types of software (compiler, linker-loader, files, etc.). It is the operating system that manages the resources and allocates them to users in an efficient and fair manner. Resource management encompasses the following functions:

TSPA

- Time management (CPU and disk scheduling).
- Space management (main and secondary storages).

idiosyncrasies

- Process synchronization and deadlock handling.
- Accounting and status information.

(2) **User friendliness.** An operating system hides the unpleasant, low-level details and idiosyncrasies of a bare hardware machine and provides users with a much friendlier interface to the machine. To load, manipulate, print, and execute programs, high-level commands can be used without the inconvenience of worrying about low-level details. The layer of operating system transforms a bare hardware machine into a virtual or abstract machine with added functionality (such as automatic resource management). Moreover, users of the virtual machine have the illusion that each one of them is the only user of the machine, even though the machine may be operating in a multiuser environment.

User friendliness issues encompass the following tasks:

EEPF

- Execution environment (process management—creation, control, and termination—file manipulation, interrupt handling, support for I/O operations, language support).
- Error detection and handling.
- Protection and security.
- Fault tolerance and failure recovery.

1.3 DESIGN APPROACHES

An operating system could be designed as a huge, jumbled collection of processes without any structure. Any process could call any other process to request a service from it. The execution of a user command would usually involve the activation of a series of processes. While an implementation of this kind could be acceptable for small operating systems, it would not be suitable for large operating systems as the lack of a proper structure would make it extremely hard to specify, code, test, and debug a large operating system.

The design of general purpose operating systems has matured over the last two and a half decades and today's operating systems are generally enormous and complex. A typical operating system that supports a multiprogramming environment can easily be tens of megabytes in length and its design, implementation, and testing amounts to the undertaking of a huge software project. In this section, we discuss design approaches intended to handle the complexities of today's large operating systems. However, before we discuss these approaches, we first need to make the distinction between *what* should be done and *how* it should be done, in the context of operating system design.

Separation of Policies and Mechanisms

Policies refer to *what* should be done and mechanisms refer to *how* it should be done. For example, in CPU scheduling, mechanisms provide the means to implement various scheduling disciplines, and policy decides which CPU scheduling discipline (such as FCFS, SJTF, priority, etc.) will be used [14, 24].

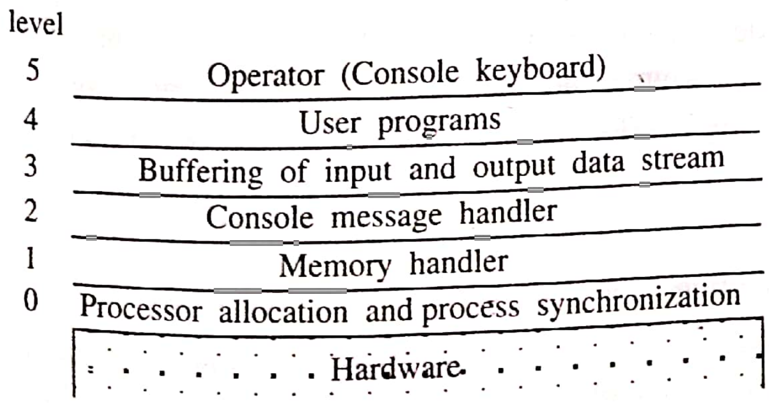
A good operating system design must separate policies from mechanisms. Since policies make use of underlying mechanisms, the separation of policies from mechanisms greatly contributes to flexibility, as policy decisions can be made at a higher level. Note that policies are likely to change with time, application, and users. If mechanisms are separated from policies, then a change in policies will not require changes in the mechanisms, and vice-versa. Otherwise, a change in policies may require a complete redesign.

1.3.1 Layered Approach

Dijkstra advocated the layered approach to lessen the design and implementation complexities of an operating system. The layered approach divides the operating system into several layers. The functions of an operating system are then vertically apportioned into these layers. Each layer has well-defined functionality and input-output interfaces with the two adjacent layers. Typically, the bottom layer interfaces with machine hardware and the top layer interfaces with users (or operators). The idea behind the layered approach is the same as in the seven-layer architecture of the Open System Interconnection (OSI) model of the International Standards Organization (ISO).

The layered approach has all the advantages of modular design. (In modular design, the system is divided into several modules and each module is designed independently.) Thus, each layer can be designed, coded, and tested independently. Consequently, the layered approach considerably simplifies the design, specification, and implementation (the coding and testing) of an operating system. However, a drawback of the layered approach is that operating system functions must be carefully assigned to various layers because a layer can make use only of the functionality provided by the layers beneath it.

A classic example of the layered approach is the THE operating system [8], which consists of six layers. Figure 1.1 shows these layers with their associated functions. Another classic example of this approach is the MULTICS system [19], which is structured as several concentric layers (rings). This ring structure in MULTICS not only simplifies design and verification, but it also serves as an aid in designing and implementing protection. In MULTICS, privilege decreases from the inner ring to the successive outer rings. The ring structure nicely defines and implements the protection in MULTICS.


level
5 Operator (Console keyboard)
4 User programs
3 Buffering of input and output data stream
2 Console message handler
1 Memory handler
0 Processor allocation and process synchronization
Hardware

HPMCBUO
HPMCBUO
HPMCBUO
HPMCBUO

FIGURE 1.1
Structure of the THE operating system.

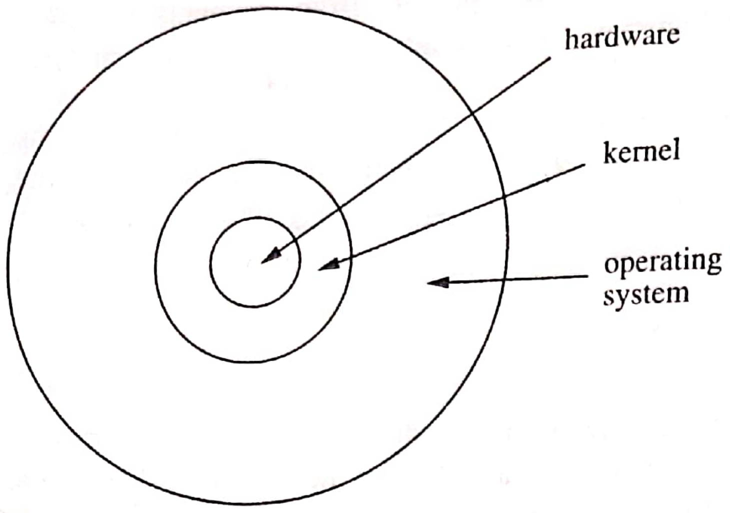


FIGURE 1.2
Structure of a kernel-based operating system.

1.3.2 The Kernel Based Approach

The kernel-based design and structure of operating systems was suggested by Brinch Hansen [12]. The *kernel* (more appropriately called the *nucleus*) is a collection of primitive facilities over which the rest of the operating system is built, using the functions provided by the kernel (see Fig. 1.2). Thus, a kernel provides an environment to build operating systems in which the designer has considerable flexibility because policy and optimization decisions are not made at the kernel level. It follows that a kernel should support only mechanisms and that all policy decisions should be left to the outer layer. An operating system is an orderly growth of software over the kernel where all decisions regarding process scheduling, resource allocation, execution environment, file system, and resource protection, etc. are made.

According to Hansen, a kernel is a fundamental set of primitives that allows the dynamic creation and control of processes, as well as communication among them. Thus, the kernel as advocated by Hansen only supports the notion of a process and does not include the concept of a resource. However, as operating systems have matured in functionality and complexity, more functionality has been relegated to the kernel. A kernel should contain a minimal set of functionality that is adequate to build an operating system with a given set of objectives. Including too much functionality in a kernel results in low flexibility at a higher level, whereas including too little functionality in a kernel results in low functional support at a higher level.

An outstanding example of a kernel is the *Hydra*, the kernel of an operating system for C.mmp, a multiprocessor system developed at Carnegie-Mellon University [28]. The Hydra kernel supports the notion of a resource and process, and provides mechanisms for the creation and representation of new types of resources and protected access to resources.

1.3.3 The Virtual Machine Approach

In the virtual machine approach, a *virtual machine software* layer on the bare hardware of the machine gives the illusion that all machine hardware (i.e., the processor, main memory, secondary storage, etc.) is at the sole disposal of each user. A user can execute the entire instruction set, including the privileged instructions. The virtual machine software creates this illusion by appropriately time-multiplexing the system resources among all the users of the machine.

A user can also run a single-user operating system on this virtual machine. The design of such a single-user operating system can be very simple and efficient because it does not have to deal with the complications that arise due to multiprogramming and protection. The virtual machine concept provides higher flexibility in that it allows different operating systems to run on different virtual machines. Uniprogrammed operating systems can mix with multiprogrammed operating systems. The virtual machine concept provides a useful test-bed to experiment with new operating systems without interfering with other users of the machine. The efficient implementation of virtual machine software (e.g., VM/370), however, is a very difficult problem because virtual machine software is huge and complex.

A classical example of this system is the IBM 370 system [21] wherein the virtual machine software, VM/370, provides a virtual machine to each user. When a user logs on, VM/370 creates a new virtual machine (i.e., a copy of the bare hardware of the IBM 370 system) for the user. In the IBM 370 system, users traditionally run the CMS (Conversational Monitor System) operating system, which is a single-user, interactive operating system.

1.4 WHY ADVANCED OPERATING SYSTEMS

In the 1960s and 1970s, most efforts in operating system design were largely focused on the so-called traditional operating systems, which ran on stand-alone computers with single processors. Considerable advances in integrated circuit and computer communication technologies over the last two decades have spurred unprecedented interest in multicomputer systems and have resulted in the proliferation of a variety of computer architectures, viz., shared memory multiprocessors to distributed memory distributed systems. These multicomputer systems were prompted by the need for high-speed computing that conventional single processor systems were unable to provide [1].

Multiprocessor systems and distributed systems have many idiosyncrasies not present in traditional single-processor systems. These idiosyncrasies render the design of operating systems for these multicomputer systems extremely difficult and require that nontrivial design issues be addressed. Due to their relative newness and enormous design complexity, operating systems for these multicomputers are referred to as *advanced* or *modern* operating systems. An advanced operating system not only harnesses the power of a multicomputer system; it also provides a high-level coherent view of the system; a user views a multicomputer system as a single monolithic powerful machine. A study of advanced operating systems entails a study of these nontrivial design techniques.

Due to the high demand and popularity of multicomputer systems, advanced operating systems have gained substantial importance and a considerable amount of research has been done on them over the last two decades. This book presents a study of advanced operating systems with a special emphasis on the concepts underlying the design techniques.

1.5 TYPES OF ADVANCED OPERATING SYSTEMS

Figure 1.3 gives a classification of advanced operating systems. The impetus for advanced operating systems has come from two directions. First, it has come from advances in the architecture of multicomputer systems and is now driven by a wide variety of high-speed architectures. Hardware design of extremely fast parallel and distributed systems is fairly well understood. These architectures offer great potential for speed up but they also present a substantial challenge to operating system designers. Operating system designs for two types of multicomputer systems, namely, multiprocessor systems and distributed computing systems, have been well-studied.

A second class of advanced operating systems is driven by applications. There are several important applications that require special operating system support, as a requirement as well as for efficiency. General purpose operating systems are too broad in nature and inefficient and fail to provide adequate support for such applications. Two specific applications, namely, database systems and real-time systems, have received considerable attention in the past and the operating system issues for these systems have been extensively examined. Other applications include graphics systems, surveillance, and process control.

A brief introduction of four advanced operating systems follows.

Distributed Operating Systems

Distributed operating systems are operating systems for a network of autonomous computers connected by a communication network. A distributed operating system controls and manages the hardware and software resources of a distributed system such that its users view the entire system as a powerful monolithic computer system. When a program is executed in a distributed system, the user is not aware of where the program is executed or of the location of the resources accessed.

The basic issues in the design of a distributed operating system are the same as in a traditional operating system, viz., process synchronization, deadlocks, scheduling, file systems, interprocess communication, memory and buffer management, failure recovery, etc. However, several idiosyncrasies of a distributed system, namely, the lack of both

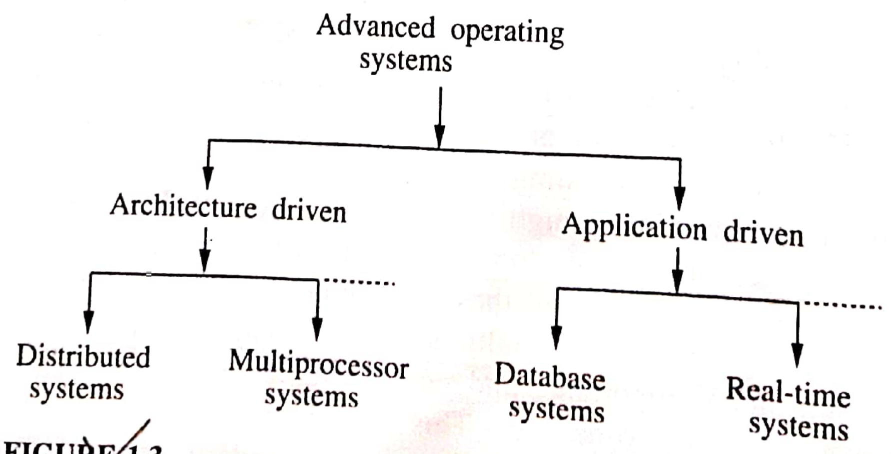


FIGURE 1.3
A classification of advanced operating systems.

shared memory and a physical global clock, and unpredictable communication delays, make the design of distributed operating systems much more difficult.

Multiprocessor Operating Systems

A typical multiprocessor system consists of a set of processors that share a set of physical memory blocks over an interconnection network. Thus, a multiprocessor system is a tightly coupled system where processors share an address space. A multiprocessor operating system controls and manages the hardware and software resources such that users view the entire system as a powerful uniprocessor system; a user is not aware of the presence of multiple processors and the interconnection network.

The basic issues in the design of a multiprocessor operating system are the same as in a traditional operating system. However, the issues of process synchronization, task scheduling, memory management, and protection and security, become more complex because the main memory is shared by many physical processors.

Database Operating Systems

Database systems place special requirements on operating systems. These requirements have their roots in the specific environment that database systems support. A database system must support: the concept of a transaction; operations to store, retrieve, and manipulate a large volume of data efficiently; primitives for concurrency control, and system failure recovery. To store temporary data and data retrieved from secondary storage, it must have a buffer management scheme.

In this book, we primarily focus on concurrency control aspects of database operating systems. Concurrency control, one of the most challenging problems in the design of database operating systems, has been actively studied over the last one and a half decades. An elegant theory of concurrency control exists and a rich set of algorithms to solve the problem have been developed. Recovery and fault tolerance are covered in Chaps. 12 and 13.

Real-time Operating Systems

Real-time systems also place special requirements on operating systems, which have their roots in the specific application that the real-time system is supporting. A distinct feature of real-time systems is that jobs have completion deadlines. A job should be completed before its deadline to be of use (in *soft* real-time systems) or to avert a disaster (in *hard* real-time systems). The major issue in the design of real-time operating systems is the scheduling of jobs in such a way that a maximum number of jobs satisfy their deadlines. Other issues include designing languages and primitives to effectively prepare and execute a job schedule.

1.6 AN OVERVIEW OF THE BOOK

In this book, we study three types of advanced operating systems, namely, distributed operating systems, multiprocessor operating systems, and database operating systems. Based on these topics, the book is divided into seven parts.