CHAPTER

1 Overview

Learning Objectives

After reading this chapter, you should be able to:

- Describe an operating system, what it does, and why it is important.
- Describe the key concepts and terminologies pertaining to operating systems.
- Explain the two different platforms, namely hardware and software, and the interactions between them.
- Differentiate between application, utility, library, and operating system.
- Explain the semantics of operation, operation execution, program, and program execution.
- Describe the inner-workings of typical operating systems—processing of system call, interrupt, and exception.
- Identify the various types of operating systems.

1.1 Introduction

This chapter introduces operating systems and their functionalities, with an intuitive but overall understanding of the subject matter of this book. It explains the purpose of incorporating operating systems in computers, discusses what they actually do for computer users, and illustrates how users interact with them. Overall, it is a short tour of an operating system.

This chapter touches upon many important features of typical general-purpose operating systems. The intent is to introduce readers to some fundamental concepts pertaining to operating systems, their implied meanings, and terminologies. One should not worry too much even if

she[1] does not clearly understand everything discussed in this chapter straight away. If some concepts are difficult to comprehend, just ignore them, and move on. They will be readdressed in later chapters.

A computer system traditionally has two parts: hardware and software. The hardware is a physical medium or physical device and, in the computing context, is designed to manipulate, transmit, or store information. The processor, the memory, the monitor, the keyboard, the mouse, disks, printers, and network interface cards are examples of hardware. Some are essential and some are optional. Software, in a broader sense, refers to something that can be stored electronically. In a restricted way, software refers to programs that can be executed or transformed to be executed in a computer. We prefer this restricted definition of software in this book.

Software is loosely classified into system software and application software. System software aims to help the users of the hardware to perform their intended tasks comfortably and more efficiently. The operating system is the core of system software; and the remaining part of system software is generally referred to as utility programs. Logically, the operating system resides right on top of the computer hardware, and applications and utilities reside on top of the operating system (see Fig. 1.1). The operating system interacts with the hardware and transforms itself into an abstract system for convenient use by applications and utilities. Users interact with a computer system by executing utilities and applications.

In simple terms, the computer is a powerful information processing machine. To perform a task, the computer executes a (software) program corresponding to that task written in a very low-level language, called the *machine language*. Writing programs in machine language is generally difficult, error prone, and time consuming. Ordinary program writers are comfortable only with higher-level languages, and using a computer by means of writing programs in the machine language is a challenging task for them. The operating system comes into play at this point. On the top of "difficult-to-use" bare hardware, the operating system creates an abstract machine that offers an "easy-to-use" interface and provides users a suitable software platform to develop and execute programs. The abstraction hides the complexity that lies beneath its surface and reveals only a user-friendly interface.

Several advancements have been made in the last fifty years since computers have become common in use. The increased sharing of computers has led to individual computers being connected to become part of a global cyberspace. Box 1.1 presents the evolution of operating systems. Also, more resources such as processing units, memory units, and peripheral devices have been added to expand the abilities of a computer. Such advancements have increased the complexity of computer usage and have created a number of issues related to the resource management and security of computers. Again, it is the operating system that comes to the rescue by acting as a resource manager and security guard for the computer system and its users.

• *A concept* is an abstraction or generalization that we derive from our perceptions or understanding of some aspects of one or more physical- or logical entities. A concept is a definable, distinctive feature that we can identify out of those entities.

• Each abstract concept is known by a name that is called a *terminology*. Terminologies provide us mental frameworks to deal with concepts, and to understand the related disciplines better.

≫ Computer hardware includes the computer (essential part) and the peripheral devices (optional part). The boundary between the essential- and the optional parts changes as the requirements and the expectations of a computer system change with time. For example, the disk is no more considered as an optional device as it has become an essential part of every computer system.

[1] In this book, *she* stands for *he* or *she*; *her* for *his* or *her*.

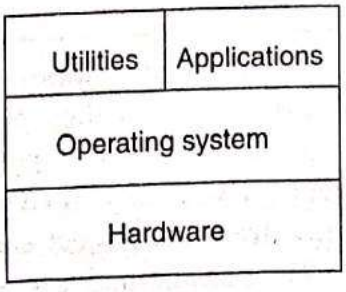

Figure 1.1: *An abstract view of a computer system.*

The operating system plays the central role in the computing world. The theme of this book is to study how the operating system accomplishes its tasks.

Our main goal in this chapter is to introduce readers to core concepts and terminologies with as few forward references as possible. Consequently, the chapter may appear to be a particle in a state of Brownian motion in space; however, it is definitely not. The following three sections (1.2–1.4) primarily deal with hardware. The rest of the chapter deals with software concepts. We conclude the chapter with a brief discussion of various design and development challenges.

1.1 Evolution of Operating Systems

The earliest computers did not use an operating system at all. Gradually, based on specific requirements, system programs were developed to facilitate the use of computers. Such programs later evolved into various operating systems. Since many operating systems evolved in various places in an uncoordinated fashion, obtaining a coherent picture of their evolution is not easy. However, the general trend indicates that the evolution has been driven mainly by new expectations, applications, and technologies. The evolution may be viewed from many different angles. This book presents one such view of evolution in seven major phases described by Brinch Hansen, and we add real-time and embedded operating systems as another phase.

Open Shop: This was the first phase in the evolution of computers. No apparent operating system was supported in this phase. Each user was allotted a fixed amount of time to use the computer. However, a major part of the time went into setting up the computing environment instead of doing the computational work. These computers could execute only one program at a time. For example, IBM 701, in 1954, was used as open shop and about US$ 146,000 per month was the cost of wasted computer time due to manual setup.

Closed Shop or Batch Processing: The cost of wasted computer time in open shop systems clearly showed that removing the general users from the computer room and closing the shop would enable better utilization of resources. Professional computer operators in closed shops handled the computers. The users were asked to prepare their application programs and data in punched cards and submit them as a "job" to the computer room for execution. The jobs were then batched by similar resource requirements, these batches were executed sequentially, and the jobs in each batch were executed one at a time, in first-in–first-out order. In this context, batch processing means that jobs are loaded and executed, all automatically instead manually. It was still a uniprogrammed system. Fast-tap stations, and small, dedicated computers for I/O were added to speed up the process. The IBM 709 was an early batch system. Though it represented an improvement over open-shop systems, yet due to the slow

(*Continued*)

Box 1.1 (*Continued*)

speed of I/O devices, the processor remained idle most of the time.

Multiprogramming: The key objective of multiprogramming was to utilize the processor to its maximum capacity and keep it as busy as possible. The initial idea was to keep many programs ready in the main memory so that if one program waits for an I/O operation, another ready program can use the processor without delay. The hardware interrupt system enabled the processor to switch between programs. As hardware technology improved, large secondary storage devices such as the drum or the disk appeared to replace the earlier storage device, called the tape. Along with multiprogramming, this change allowed the input units, the processor, and the output units to perform simultaneously. Such an arrangement was called *simultaneous peripheral operation on-line* (or spooling). Concepts such as job scheduling, demand paging, virtual memory, and supervisor calls were introduced in these systems. The Burroughs B5000 Master Control Program is an example of a multiprogramming system. Although processor utilization was improved in multiprogramming systems, substantial time was still required to complete the execution of a program. It was still a batch system, and hence, application programs were executed behind the scenes, and there was no way that users could observe, correct, or react to the results of the execution interactively.

Time Sharing: To allow users to observe, correct, or react to the results of the program execution, each user was given a remote terminal. The processor would attend to other users while one user was reacting to some I/O request. Thus the processor could serve many users in a short span of time as its speed was considerably faster than that of the users. This technique is called *time sharing* (of processor) and it creates the illusion of a dedicated processor for each user. It was a major breakthrough in operating systems. The Compatible Time-Sharing System (CTSS) was designed at the Massachusetts Institute of Technology (MIT) with such goals in mind. Later, MIT in collaboration with AT&T Bell Laboratories started a much larger time-sharing system called MULTICS (MULTiplexed Information and Computing Service) in the mid 1960s. Eventually, AT&T Bell Laboratories withdrew from the project and designed the UNIX, in 1971, as an alternative to MULTICS. Concepts such as file system, file protection, password, password protection, etc., were introduced as part of time-sharing systems. At that stage, due to concurrency, operating systems had reached a level of maturity in functionalities and hence complexity, beyond human comprehension to continue to design them in ad hoc fashion.

Concurrent Programming: Concurrent programming may be considered as the next logical step in the evolution of computer systems. It provided the conceptual basis for the development of complex software systems. Concurrent programming refers to the ability of the computer to execute multiple tasks simultaneously. This feature was demonstrated in terms of some model operating systems such as "THE" operating system of Edger Dijkstra, and "RC 4000" operating system of Brinch Hansen. (THE stands for Technische Hogeschool Eindhoven in Dutch—in English, Technical School of Eindhoven, the Netherlands.) These operating systems focused mainly on structuring operating system functions and offering effective synchronization primitives to facilitate concurrent programming. Synchronization primitives such as semaphore, monitor, etc., were introduced in these systems.

Personal Computing: Advancements in microprocessors and semiconductor memories, during the 1970s put computers within the reach of individuals. Such personal computers required simple single-user operating systems. In such systems, user convenience is more important than resource utilization. Thus, the operating systems functionalities in personal computers were simplified to suit single users. Since the system was aimed at the single-user environment, the features required for competitive scheduling and protection were considered irrelevant, and therefore, were not supported in the earlier systems of personal computing. Other features such as graphical user interface,

Box 1.1 (*Continued*)

interaction through mouse clicks were also added. The MS-DOS, Windows 95, and Macintosh systems are examples of single-user operating systems.

Distributed Systems: All the above-mentioned phases in the development of operating systems were meant only for self-contained, stand-alone computers. There was no communication with other computers and no distribution of large computations among multiple computers. Gradually, the networking component became part of operating systems to enable communication among different computers. Due to such communications becoming possible many computers could be grouped to work together, often assuming specific responsibilities. Servers and resource sharing became common in these systems. Distributed programming, message passing, remote procedure call are some dominant concepts in these systems.

Real-time and Embedded Systems: As computers became part of many real-life systems such as cruise control, the response offered by time-sharing systems was not sufficient for many systems requiring responses in real time. These systems also had other limitations such as memory, energy, size, etc. To satisfy the needs of these systems, operating systems were specially designed. Response time, fault tolerance, and low energy consumption are some typical requirements of these systems. V x Works is an example of a real-time operating system; Symbian is an embedded operating system.

1.2 Basic Terminology

The preceding section presented a very brief introduction to operating systems: why we need them and what they do. That section is perhaps a little harder on novice readers, because several terminologies (including "operating system") were introduced there without little or no explanation. Every discipline comes with its own set of specialized terminologies, and the discipline of operating systems is no different. Understanding some of these terminologies is the first step to understanding the discipline. In this section, we explain some widely-used terminologies. Some of these will be recalled in the latter part of the book. Experienced readers may skip or skim through this section.

≫ The most important aspect of learning any science or technology is to focus on core concepts. A concept, however, does not exist in isolation; usually there are collections of related concepts. We also need to understand those relationships to become experts in that field. Operating systems are no different.

1.2.1 Hardware

We start with a very basic question: What is a computer? The simple answer is that a *computer* is a powerful information manipulator—a data-processing engine that is energized by an electric power supply unit. Furthermore, a computer is composed of various independent physical units called *hardware components* or resources, often called *hardware devices*. Hardware components in a computer minimally include a processor, a main memory, and a few input-output (I/O) devices. Examples of I/O devices include the keyboard, the mouse, the monitor, disks, floppies, CDs, tapes, printers, network interface cards, etc. Each hardware component performs some specific task such as storing, transmitting, or manipulating information. For example, a processor manipulates information, a memory unit stores information, a network card transmits information. These components are connected to one another by electrically conducting wires. Different wires are normally used to transfer different types of information. Many wires, carrying the same kind of related information, are collectively called a *bus*.

≫ An *abstraction* represents a general description of a real-world problem and is a means to study a set of related properties of the problem. It helps us in modelling those properties of a problem in which we are interested. It describes certain specific features of the problem, which can be identified as separate and distinct properties.

≫ To us a piece of data is a representation of some physical or logical entity or object. The data represents a form of the object, and it is stored on a physical medium. At any time, the physical signal stored on the medium represents a state of the object, and is called a value of the object. The meanings we assign to different values of data are called *information* that is encapsulated in the data. However, we use object, data, and information interchangeably in this book. Note that the same data value may represent instances of different objects, and hence, its meaning is interpreted according to the context of the objects. An object also specifies the set of operations that are to be solely used for data manipulations or information processing purpose. The operations are the only information processing rules for the object. One execution of an operation on the object is called an *access* or *reference* to the object in a particular way.

Hardware components exchange information among themselves to accomplish their individual tasks, and for effective and smooth functioning of the computer. They communicate among themselves by changing signals (i.e., voltage levels) on the communication wires, where different voltage levels represent different pieces of information. Hardware components work together to achieve the purpose of a computer.

Figure 1.2 presents a simplified model of a typical computer. It depicts one processor, one bank of the main memory, and some peripheral devices—often referred to as I/O devices—connected to a single bus for inter-component communications. In practice, however, many different kinds of busses are used in computers. Different busses are used for different purposes, for example, for the transmission of control, data, and address information.

Each hardware resource plays a different role in a computer. The most important one among them is the processor. It is the *information manipulating engine* in the computer, and is the overall incharge of activities of all the other resources. The next important resource is the main memory; it stores information that is needed by the processor and the I/O devices during information manipulation. The processor uses I/O devices to interact with the external environment as well as to store information for future retrieval. For example, a monitor is used to display information in readable form, a printer to print information on papers, a disk to store information that would survive power disconnections, a network interface card to exchange information with other computers, and so forth.

1.2.2 Primitive Data and Operation

The processor is the brain of the computer, and it controls and coordinates activities of the other devices. It continuously carries out various "operations" for smooth functioning of the computer. Operation is a generic term and is extensively used in this book. Let us now digress briefly and define what an operation is.

An *operation* maps a non-empty set called the *input set* into another non-empty set called the *output set*, as shown in Fig. 1.3. In the figure, the arrows represent the map from the input set to the output set. The elements of these sets are also called values or data values or simply data. The sets may be infinite but the mapping associated with an operation is usually described by a finite collection of information processing rules. Given an input data value, the rules are applied on the input to produce the output data value. An entity that

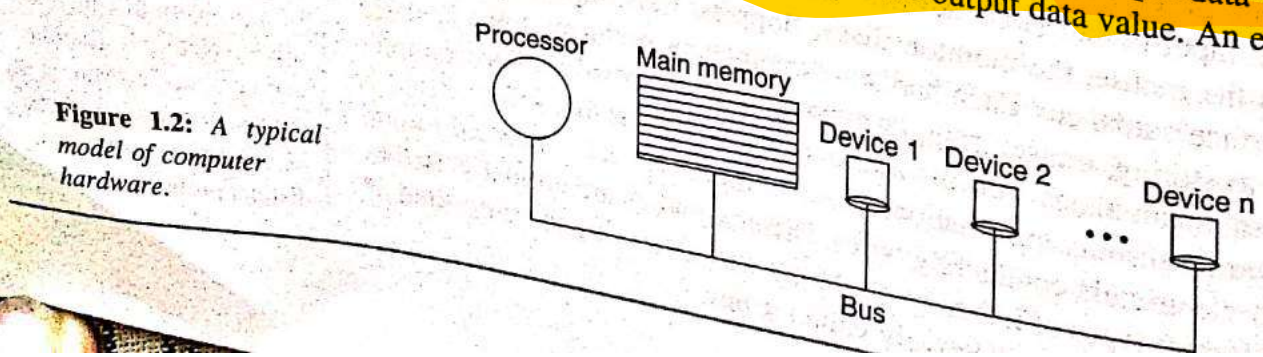

Figure 1.2: *A typical model of computer hardware.*

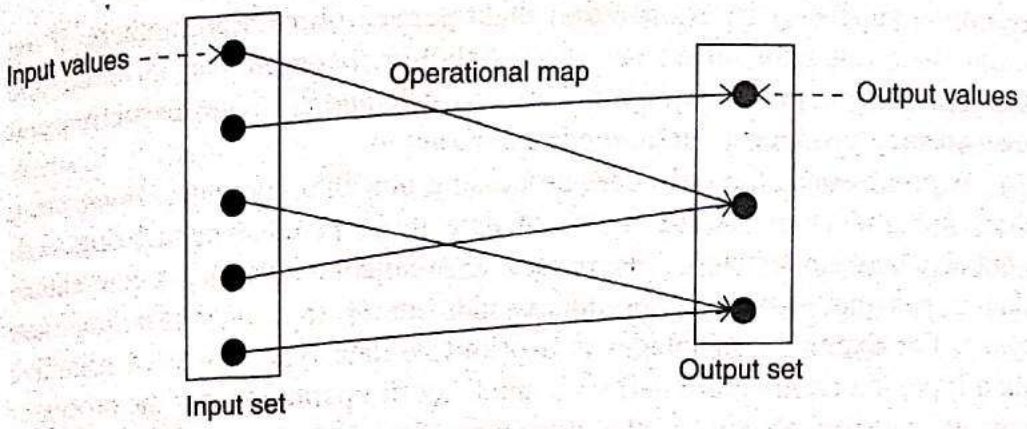

Figure 1.3: *An abstract representation of an operation.*

carries out the information processing rules is called an *execution unit* for the operation. This definition of operation is mathematically precise. However, we may also use the term operation a little loosely, and we may have multiple input- and output sets. For example, the addition of two integers is an operation that has two input sets and one output set, and the three sets are the set of integer numbers.

At one level, we assume some operations as *primitive* in the sense that we know what these operations do, but we may not know the rules of the operations. For example, at the elementary education level, we assume a table is available for adding two single-decimal digits and are not concerned how the table is constructed. Adding two single digits (say, 5 and 7) is a primitive operation for elementary students. When they multiply two multi-digit numbers, they apply various operational rules and primitive addition operations to complete the multiplication.

≫ We differentiate between operation and operation execution. For example, depositing money in a bank is an operation. Depositing $100 in a particular account numbered 12357 at 11:13 AM on 17 March 1923 at branch 2931 is an operation execution.

The *execution* of an operation is an application or realization of the operational rules on an input data value to produce the corresponding output data value. For example, given two integer numbers, adding them together is an execution of the addition operation. A specific instance of an execution of an operation is referred to as an *operation execution*. (In this context, we often refer to input value(s) as *operand*(*s*) or *parameter*(*s*) of the operation for its execution.) It is assumed that an execution unit will complete each operation execution in a finite time, though of unknown duration. For example, adding 5 and 7 is an operation execution; adding 17 and 19 is another operation execution. In the former case, 5 and 7 are operands of the execution of the addition operation, and so are 17 and 19 for the latter case.

Each operation is assumed to be *deterministic*, that is, it produces the same output data value whenever it is applied on the same input data value (regardless of the time required by an execution unit to carry out the rules of the operation). Executions of the same operation on the same input data value in different situations may need different lengths of time, but they all produce the same output. Specifically, we also assume that the input data value does not change during the execution of the operation, or at the least whenever the operation execution unit reads the input data. If the input data value changes at any time during an operation execution, its completion may not be guaranteed, and even if it completes, the output it produces may be incorrect. For

example, suppose f is an operation that operates on natural numbers. If the input data value for an execution of f flickers between, say, 5 and 7, the function output may not flicker between $f(5)$ and $f(7)$; the execution unit instead may produce some unpredictable output.

≫ The architecture of a system is the design of the system: design of its subsystems and connections among them.

A processor is the only data-processing unit in a computer. However, it does not understand all the types of data that we, the human beings, can conceive and understand. The processor architecture defines a few simple data types that the processor understands; these are called *primitive data types*. For example, an integer is a primitive data type. For each primitive data type, the architecture defines a small set of operations that the processor can execute on objects of the data type. These operations are known as *machine instructions* or *primitive instructions* (or simply, *instructions*). Each instruction is deterministic, and it encodes a simple information processing work. Briefly, an *instruction* is a primitive operation that may be applied on a piece of primitive data to deterministically transform the information (value) encoded in the data. A specific instance of an execution of an instruction is referred to as an *instruction execution*. The data specified in an instruction for manipulation during its execution are the operands of that instruction. There are also non-information processing instructions (such as jump) that are used for purposes of control.

The processor executes machine instructions sequentially, that is, one after another, and does nothing else.[2] In other words, the processor is a hardware device that, once energized by a power supply, after a very short initialization time perpetually executes machine instructions one after another until its power supply is disconnected. By executing instructions, the processor manipulates information stored in the main memory and the I/O devices, and thereby, controls activities of all the resources in the computer.

1.2.3 RAM

≫ To us, the users of computers, machine instructions are primitive because, as operating system experts, we may not really need to know how the processor carries out instruction executions.

The random access memory (RAM), popularly called the main memory that stores both data values and instructions, is used by the processor and the I/O devices. The memory is a passive device in the sense that it neither interprets nor modifies the stored content on its own. The processor alone interprets the content as primitive instructions or primitive data, but both the processor and the I/O devices have access to the content. Each primitive instruction or data occupies a tiny amount of fixed space in the main memory, and this space is called a *memory location*.[3] The main memory is viewed as a finite collection (linear array) of memory locations and each location is identified by an address called a *memory address* or *physical address*.

[2] In this book, we do not address the issues related to simultaneous executions of multiple instructions by the processor.

[3] Without loss of generality, for ease in exposing the subject material of this book, we assume that memory locations are primitive storage components of the main memory, and each memory location stores a finite amount of information (either an instruction or an operand). In practice, memory locations may be different; for example, an instruction could span many primitive

It is assumed that the memory device implements at least two operations, *read* and *write*, to access memory locations individually. The content of a memory location is read or written by invoking the read or write operation, respectively, against the memory address. To access a memory location, the corresponding memory address is given to the memory device. The memory device accepts requests for operations from the processor and the I/O devices, and carries out the operations against the memory addresses specified. Each memory access (read or write) takes a finite amount of time.

≫ When a memory operation is applied to a memory location, the entire information content at the location is accessed as an indivisible unit. The content of a memory location is assumed to be the unit of information transfer between the main memory and other hardware components.

1.2.4 ROM

The main memory (aka, RAM) is a *volatile* device, that is, it loses the stored content when the power supply is disconnected. Consequently, when the power supply is restored, the content of the memory is random, inconsistent, and may be unsafe to use. It is, therefore, necessary to initialize the memory content (and some I/O devices too) with proper values every time the computer is powered on or is forced to reset itself. The main memory needs to be initialized to a consistent state before it can be used effectively. The processor carries out this initialization task, and this is called *bootstrapping*.

≫ The term memory location may refer to a location in either the RAM or the ROM.

≫ The processor can only fetch instructions from the main memory and not from I/O devices even though it can access operands from both devices.

When a computer is powered on or is reset, the computer hardware circuit first resets and initializes the processor, which then starts executing instructions from a known memory location. (This location information is fabricated in the processor hardware.) The few memory locations the processor accesses right at the very beginning of bootstrapping must contain reliable information (both instructions and their operands). The processor must not refer to uninitialized volatile memory to fetch instructions or to read operands because the content of the entire memory is suspect.

Each computer includes a small amount of persistent, that is, non-volatile memory commonly called *read-only memory* (ROM). It stores the instructions and data required for the preliminary bootstrapping task. The processor cannot modify the content of the ROM. It starts executing instructions from a particular location in the boot ROM every time the processor is reset. The task of the instructions and data in a boot ROM is very simple—to initialize the processor internal hardware components, parts of the main memory, and the I/O device control circuits.

≫ Processor's basic execution cycle:

```
while (power-on) {
    Fetch next instruction
    from the main memory
    Decode the instruction
    If needed, get operands
    from the main memory
    Execute the instruc-
    tion; If needed, store
    result in the main
    memory;
}
```

The execution cycle of these three (up to five) sequential actions is an automated mechanism implemented in the processor hardware. This is the working principle of the von Neumann architecture.

1.2.5 Execution Flow

The task of the processor is very simple. It cyclically fetches, decodes, and executes instructions from memory locations. Some instruction executions involve accessing operand(s) from the main memory and/or the I/O devices. The location of the next instruction is determined by the outcome of the execution of the current instruction. In short, the processor keeps executing one instruction after another, and their *flow* is determined by the outcome of the instruction executions themselves. However, on some occasions, certain events in the computer may alter the normal execution flow abruptly. These

events are called *interrupts* and *exceptions*. When an interrupt/exception occurs, the processor breaks the current execution flow, and starts a new execution flow by executing instructions from some other memory locations. This is referred to as *interrupt/exception handling* by the processor. On the termination of the interrupt/exception handling, the processor resumes the original execution flow from the place where the interrupt/exception occurred.

≫ Interrupt events originate primarily from the I/O devices. Exception events occur in the processor due to unwanted conditions such as division by zero, execution of illegal instruction, etc.

1.2.6 Program, Programming Language, and Computation

Users utilize computers to solve problems. Designing a *solution* to a problem is constructing an operation that can correctly transform the input set of the problem into the output set. A formal description of a solution to a problem using a collection of "simpler" operations is called an *algorithm*. Thus, an algorithm employs lower-level operations to construct a higher-level operation, and encodes information-processing rules that solve the problem, that is, to carry input values to the corresponding output values in finite time. A *program* is a finite step-by-step description of an algorithm as a sequence of operations expressed in a well defined language, and the language in which the program is so expressed is called a *programming language*.

≫ A *problem* is a task that abstractly specifies a mapping from its input set into its output set. The sets may be infinite, and consequently, we may need infinite space to represent the mapping if we blindly attempt to store the mapping information as an input-output relation.

A programming language supports a set of named identifiers and operators; the identifiers are used to hold data values as well as operator names. The language defines syntax rules to compose identifiers and operators. A program is a well-formed composition of operators and identifiers, and describes how the solution for an instance of a problem is carried out. Only a language manipulator that understands the language can carry out an execution of a program written in that language.

A specific instance of an execution of a program is referred to as a *program execution*. A program execution involves applying a sequence of operators to the values of identifiers, and the sequence is called a (sequential) *computation*. A solution to a problem may then be described by a set of computations, where each computation solves one instance of the problem, that is, it systematically transforms one input value to the corresponding output value. In the book, we use the terms program execution and computation to mean the same.

≫ An algorithm may be viewed as an abstract representation of a solution to a problem using lower-level simpler operations. A program is a concrete representation of the solution in a particular programming language.

It is stated in Section 1.2.2 on page 6 that the processor architecture implements a small set of primitive instructions and defines a small set of primitive data types. The architecture actually defines a programming language called the *machine language* of the processor. The processor is the only device in a computer capable of executing machine language programs.

A machine language program, also called primitive program or machine code, is composed of machine instructions and primitive data values. It is a finite description of a solution to a problem, which the processor understands. A machine language program instructs the processor precisely what it is to do at each point in the program execution. That is, when the processor executes a primitive program, the program itself directs the execution flow of the program execution in order to achieve its intended task. Thus, at the hardware

≫ In machine languages, identifiers are contents of memory locations or their addresses and operators are machine instructions.

level, each program execution (i.e., a computation) is merely a sequence of machine instruction executions and nothing else.

Processor architectures implement different machine languages, and these can be intricate. Users prefer to solve problems using different programming languages rather than in the machine language. Those are called *higher-level programming languages*. Compared to machine languages, human beings find it easier to handle higher-level languages. A higher-level programming language is a "machine-independent" language, and supports a set of machine-independent operations and a set of (higher-level) data structures. The programming language shields programmers from the need to knowing details of the machine. It is quite natural that users spend lesser effort and time to develop programs using higher-level programming languages than in using machine languages. Higher-level programs are easily portable to different computers with different types of processors.

≫ The introduction of FORTRAN in 1956 marked the beginning of the use of higher-level languages in writing programs. Examples of other higher-level languages are Ada, Cobol, Pascal, Algol, C, C++, Java. There are many others.

1.2.7 Application Programs

Higher-level user programs are called *application programs* (or simply, *applications*). An application is a standalone program that describes step-by-step procedures (in a higher-level language) to solve a problem. The processor in a computer may not understand application programs in their higher-level forms. An application is first translated into an equivalent machine language program so that the processor can execute the translated program. A translated program is known as a *binary image* or *executable code*.[4] An application program execution begins when the processor commences executing the very first instruction from the translated binary image. The program execution continues until the processor executes a termination or halt instruction, and we say that the program execution is complete. During an application program execution, an instruction execution may involve accessing the program's own data or data from outside such as those from the I/O devices. If the program reads a data item from an I/O device, it is called an input data item, and if it writes to an I/O device, it is called an output data item produced by the program execution.

≫ The spreadsheet, the flight simulator, and computer games are examples of application programs.

1.2.8 Operating System

Modern day hardware resources are very powerful but complex devices. They mostly provide complicated "difficult-to-use" intricate interfaces. Manoeuvring hardware resources by (naive) application developers is a time-consuming, daunting task if not an impossible one. To alleviate their hardship, we need sophisticated special-purpose programs to manoeuvre hardware resources. We quote Rosin verbatim here: "given that computer hardware systems are not the most convenient devices to use, the development of

≫ A program has two parts: the code and the data. The code is a sequence of instructions that manipulates the data to obtain the desired final result. Each instruction has two parts: operation code (opcode) and zero or more addresses of the data to be manipulated by the operation.

[4] We use the terms application, program, executable, binary, and code interchangeably to mean the same.

≫ The objective of system software is to provide a friendly environment in which application software can be developed and executed with relative ease.

software systems is an attempt to simulate (on existing hardware) a variety of idealized systems convenient, flexible, powerful, as well as efficient in terms of the total cost of a computer installation." Expert developers who have extensive knowledge of hardware resources can write such special programs. Application developers would find it convenient to have these special programs available in the computer itself. Such special programs would shield application developers from the complexities of hardware resources, and simplify the use of the resources by applications. These individual hardware manipulating special program pieces, that control all hardware resources in a computer, are put together and form a part of a larger software called operating system.

≫ A service is a set of functionalities provided by a system,. A service has two important characteristics: interface and implementation. The interface is the specification of the public functions of the service. Applications avail the service via the interface. A service program implements the interface and provides the functionalities described in the interface. It is important to note that an implementation of the service may be replaced by another one without affecting its interface and, therefore, the application design, development, and maintenance work remain unaffected.

An *operating system* is a software entity that controls the operation of all the hardware resources, and that effectively provides users with a new (virtual) software machine that is more convenient to use than the base hardware resources. Concisely, an operating system is a well-organized collection of special programs, and it provides an execution environment (a.k.a., software platform) in which users can develop and run their applications with relative ease. Two primary goals of an operating system are (1) user convenience in utilizing computers and (2) effective and efficient utilization of hardware resources. As shown in Fig. 1.1 on page 3, the operating system sits in between applications and hardware resources, and provides many services (e.g., reading and writing data from and to disk) that the applications can avail at runtime. The operating system coordinates the executions of these service programs by the applications.

Users utilize a computer to solve their problems by executing application programs. The processor, however, alternately executes application programs and operating system programs. It executes primarily applications, and it executes operating system programs when applications require services from the operating system. Whenever the processor executes the latter, we say that the operating system has "gained" or taken control of the computer. The theme of this book is the study of operating systems, and more specifically how they help users to manoeuvre computers and arbitrate resources as conveniently and efficiently as possible. Let us first define what we mean by a complete computer system, in the next section.

1.3 The Computer System

In a computer, the hardware resources do the real work. However, they are difficult to handle by (naive) application developers and general users. An operating system acts as a mediator between them and the hardware resources. It creates a friendly environment in which users (naive or expert) feel comfortable in using the computer and its resources. User convenience in utilizing computers is the prime concern of the designers of operating systems. Although hardware resources actually solve the problems of users, an operating system helps them do so with relative ease. This is *the* primary objective of an operating system. It transforms the computer into an "easy-to-use"

≫ Readers might be a bit confused about the term application. The operating system is an application relative to hardware. The utility and user programs are applications relative to the operating system. Unless stated otherwise, in this book, we reserve the term application to mean the latter.

software platform (known as *virtual machine*) on which users can conveniently develop applications and run them.

The operating system is an essential part of all computers. It is the sole governing system and; it not only oversees the use of all the resources in the computer but also manages their efficient utilization. It is a manager of all resources. To this end, it allocates resources to applications when needed, and reclaims them when they are no longer required. The system, therefore, promotes the automatic and effective sharing of resources among users leading to enhanced resource utilization. In addition, it resolves access conflicts among competing users and enables amicable sharing of resources.

To achieve these two objectives, the operating system controls all the resources, and provides users a software interface that is more convenient to use than what the bare hardware resources may provide. The operating system is a collection of diverse- but related software entities (program pieces), which lies between the applications and the hardware resources. The applications see the operating system as a black box that knows how to handle hardware resources. They interact directly with the operating system, and never with hardware resources.

As the operating system manages all the hardware resources, ordinary application developers are not burdened with the technological details and complexities of hardware. Instead, they can focus on the development of applications. An operating system actually provides a set of services to applications, and relies on the underlying hardware resources to implement those services. (Different operating systems may provide different services, but all provide a common set of services.) Each operating system defines a set of *service access points*, and applications connect to the appropriate access points to obtain the desired services from the operating system. A connection call to a service access point is known as a *system call, a monitor call, a supervisor call*, or an *operating system call*. They are the sole means of normal communications between the applications and the operating system. System calls are similar to ordinary function invocations in the operating-system space. System calls are the *programming interface* to the operating system.

Operating systems vary and have different mechanisms to make system calls. These calls may have different parameters. The ways of passing parameters to system calls vary from one operating system to another. Thus, the use of system calls may appear a little difficult to ordinary application developers. Owing to such variations, applications developed for use with one operating system may not work with another. To make application development even easier and ensure their portability across different operating systems, IEEE POSIX[5] standards define a set of *application programming interfaces* (APIs). The POSIX APIs help application developers to write "system-independent" applications. Applications developed using these APIs can be run, without little or no alteration, on any operating system that conforms to POSIX standards.

[5]POSIX stands for Portable Operating System based on UNIX.

≫ An *interface* is a contract between a system and its environment, that is to say, the system users. The interface specifies how the system interacts with the users. It is a named collection of functions and constant declarations. It defines the protocol (rules and conventions) for communications between the users and the system, and the behaviour of these functions. It describes the input assumptions the system makes about the environment and the output guarantees it provides. The implementation of an interface constructs all the functions declared in the interface specification. The purpose of an interface is to minimize dependencies between applications, services, and service providers. Some authors use the terminology *call-level interface*.

≫ APIs are a large collection of "ready-made" programs that provide many useful capabilities that applications can use to indirectly obtain operating system services.

System developers implement POSIX APIs, and make these available to application developers in the form of libraries. On two different systems, the interface specification remains the same, but their implementation may vary. A system library defines a set of standard functions through which applications normally interact with the operating system. When an application program wants to make use of an operating system service, it invokes a specific API in a system library that, in turn, makes appropriate system calls to obtain the required service. To application developers, APIs appear to be higher-level operations that extend the original machine instruction set. Application developers are not concerned about the actual steps involved in making system calls. The POSIX APIs take care of those steps. Therefore, to applications, system calls appear as if they are higher-level function calls to system libraries. (In spite of that, invoking a POSIX API is often *naively* termed as making a system call. We also do so in this book.)

≫ A *library* is simply an object containing the machine code (and data) that is later incorporated into application programs. A library is used to group related functions together into a single object. A set of services contained in a library object can either be statically linked or dynamically loaded into application codes. Services from one library may invoke other library services that are external to the former library. Libraries help in making application programs modular, compact, and easier to maintain.

An operating system is a well-organized collection of a basic set of programs. The most important program in the collection is called *kernel* of the operating system. (The kernel is also called *supervisor*, *monitor*, or *nucleus*.) The kernel is the minimal operating system program that is loaded in the main memory when a computer is initialized during bootstrapping. It resides permanently in the main memory until the system is shutdown (see Fig. 1.4). It typically resides in the low-memory address part of the main memory. The term *operating system* is defined as the notion of a kernel (the core), the supporting hardware management software, the various system libraries, and the specialized system applications (called utilities).

≫ *Utilities* are ready-made special purpose standalone application programs (distinct from the operating system programs) that help users to manoeuvre the computer as conveniently as possible. Some utilities perform specialized management tasks. Users (human beings or machines) always interact with a computer by executing utilities and applications.

The kernel is the central controlling authority, and provides core operating system functionalities. It typically manages hardware components, and exports services such as memory management, processor management, and I/O device management. It provides a generic interface for the rest of the operating system programs. The non-kernel parts of the operating system are loaded into and unloaded from the main memory as the situation demands during runtime. (In some systems, these other parts are supported through system applications.) The kernel contains all the critical functionalities of an operating system, which are needed for the effective and smooth functioning of a computer. However, in modern monolithic systems, the demarcation between the operating system and the kernel is blurred. Unless stated otherwise, we will use the terms kernel and operating system to mean the same in this book.

By a *computer system* we mean a computer that is fitted with an operating system. That is, it consists of the bare hardware resources and the interface operating system software. It, as a whole, provides a set of services for computer users to develop and run their applications. Figure 1.5 presents a typical hierarchical interface to the computer hardware. From the figure, it is easy to see the central role the operating system plays in the computing domain. Shown in the figure is a set of utilities. Utilities are called system programs, and are developed by experienced system

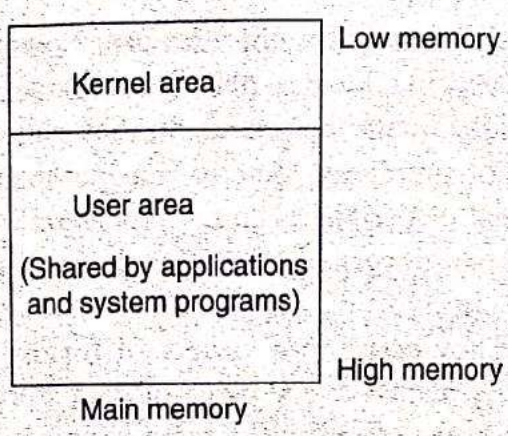

Figure 1.4: *Kernel location in the main memory.*

programmers. Compilers, assemblers, linkers, debuggers, database management software, file transfer software, shells, text editors, common I/O operations, etc., are examples of utilities. Utilities collectively define the *user interface* to the computer system. Utilities provide a friendly- and convenient environment for computer users. In practice, and as mentioned above, a computer system also supplies a set of system libraries for use by the applications and utilities. The aforementioned POSIX APIs are available in those libraries.

1.4 I/O Device Operation

In previous sections, we have mentioned that the processor architecture implements a set of primitive instructions and only the processor has the ability to execute those instructions. We also have discussed memory interface, and the manner in which the processor accesses memory locations. In this section, we study how the processor interacts with input-output (I/O) devices.

The processor uses the I/O devices to interact with the external world. Some devices transfer transient data, and some store durable data for future use. These latter devices can retain data across power disconnections, and are called *storage devices*. Examples of former devices include printers and network interface cards, and examples of the latter devices are disks, tapes, CDs, and flash memories. Some devices such as the keyboard are input-only devices, some such as the monitor are output-only devices, and some others

≫ I/O devices may be loosely classified into two types, communication devices (the printer, the keyboard, the mouse, the monitor, network interface cards, etc.) and storage devices (tapes, disks, CDs, flash memory, etc.). Communication devices act as the interface between a computer and its external world, and storage devices act as auxiliary storage units.

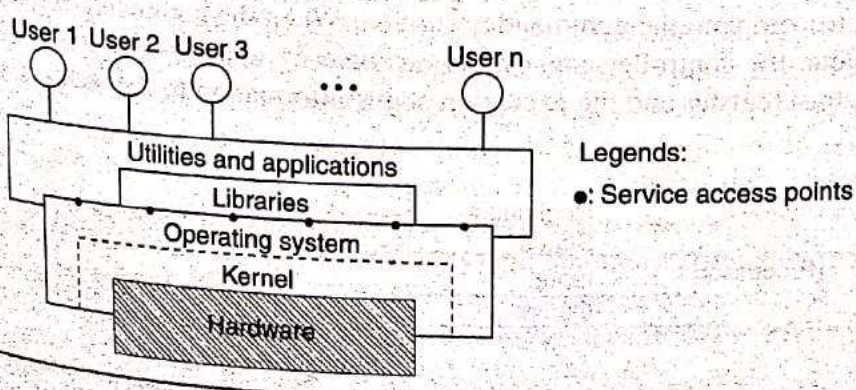

Figure 1.5: *A typical hierarchical interface to computer hardware.*

such as disks are both input and output devices. Nevertheless, in this book they all are referred to as I/O devices. (There are devices such as *timer clocks* that do not exchange data with the processor; they generate interrupt signals at regular intervals. For simplicity, we also consider them I/O devices. Timer devices can be internal or peripheral to the processor.)

As far as usage is concerned, I/O devices are the most complicated hardware resources. In addition, they are slower than the processor and the main memory. They come in a variety of forms. The processor may not be able to directly access every piece of data stored in I/O devices in the same way it accesses data from the main memory. It accesses the data indirectly.

≫ Peripheral devices are those that reside on the periphery of the computer system. All I/O devices are peripheral devices but some peripheral devices are not I/O devices.

I/O devices are connected to the host computer system via peripheral devices called *I/O controllers*. (Some I/O devices also come with built-in I/O controllers.) Controllers know how to operate the devices. Each I/O controller implements a small set of access locations called *device registers* or *operating registers*. These registers are the sole interface to the controller. The processor architecture implements some special machine instructions (called *I/O instructions*) to read and write the registers of these devices. Some processor architectures support memory-mapped I/O in which normal memory read and write instructions are also used to access device registers.

Device operating registers are classified into four broad categories: (1) command, (2) status, (3) input, and (4) output. The processor writes to the command and input registers, and reads from the status and output registers (see Fig. 1.6). There are other device registers such as configuration registers that are used to configure a device at the time of its initialization. Sometimes, a single register is used for multiple purposes.

I/O controllers carry out I/O operations on the I/O devices on behalf of the processor. Modern I/O controllers are mostly autonomous devices, in the sense that they can carry out I/O operations independent of interventions from the processor, even though the latter controls all their activities. Each I/O controller cyclically accepts commands and input data from the processor, executes the commands, and returns output data and status information to the processor. The processor directs the I/O controller by writing an I/O command to the controller command register and input data to the controller input register. After receiving a command from the processor, the controller starts executing the command in its own way and own speed, and it may take a while to complete the command execution. When the command execution is complete, the controller conveys its outcome by writing the output value to the output register and the execution status information to the status register.

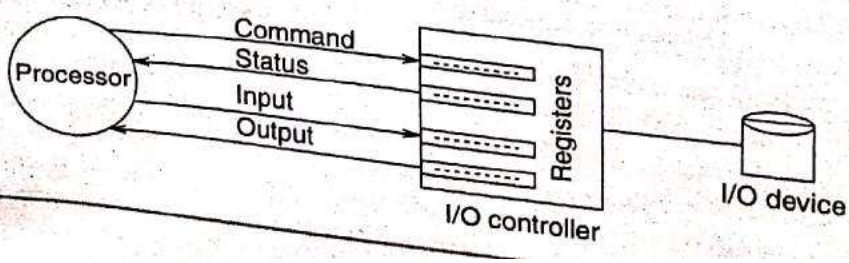

Figure 1.6: *Processor and I/O controller interactions.*

After initiating an I/O command execution at an I/O controller, the processor needs to confirm when the command execution is indeed complete. The processor has the following two options until the controller completes the command execution.

1. The processor continuously (or intermittently) tests the controller state by reading the controller status register. It does so until the command execution by the controller is complete. When the command execution is indeed complete, the processor reads the output data from the controller output register. This option is called *programmed I/O*.
2. The processor, after delivering the command to the controller, moves on to do something else. (The controller and the processor concurrently and independently perform different tasks.) When the controller finishes the command execution, it interrupts the processor to draw its attention. On receiving the interrupt, the processor suspends its current program execution, and starts executing a different program called an *interrupt service routine* where it reads the controller status and output registers. This option is called *interrupt-driven I/O*.

If the I/O controller responds to commands very quickly, the programmed I/O option is the best. Unfortunately, most devices are slow compared to the processor speed, and the programmed I/O would degrade the performance of the system. In modern computers, an interrupt driven I/O scheme is implemented; an interrupt mechanism is the primary means of coordination between the processor and I/O controllers. Controllers raise signals on interrupt communication wires whenever they need the processor's attention. The processor continuously monitors all the interrupting wires. It literally examines the presence of interrupt signals after completing each instruction execution (but not in the midst of an instruction execution). As noted in Section 1.2.5 on page 9, if an interrupt signal is present, the processor suspends the execution of the current program, and starts executing the corresponding interrupt service routine. When the execution of this service routine is complete, the processor resumes the execution of the suspended program or starts executing another program depending on the requirements. The interrupt-driven I/O scheme makes it possible for the processor and the I/O controllers concurrently perform different tasks.

≫ Though not shown in Fig. 1.6, there is an interrupt wire connecting the I/O controller to the processor via an interrupt controller device.

In general, interrupts from different I/O controllers are handled by different interrupt service routines. Each interrupt service routine is a part of a larger piece of software called the *device driver*. A device driver is a piece of software that understands and exploits various device capabilities to operate the device in the most efficient way. A device driver interfaces each device with the operating system, and may control many similar devices (see Fig. 1.7). These drivers are a part of the operating system, and they provide a uniform interface for the rest of the operating system. The operating system invokes the (uniform) driver interface routines to obtain services from the devices and to control their activities.

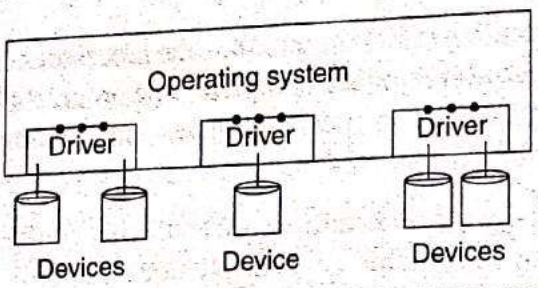

Figure 1.7: *Device driver interface to I/O devices.*

1.5 Process Abstraction

A computer system is composed of three distinct parts (see Fig. 1.5 on page 15): (1) hardware resources, (2) operating system software, and (3) utility programs. Users manoeuvre the computer by running system utilities and their own application programs. They run these programs to solve their problems. The processor, on behalf of the users, may execute these programs sequentially, that is, one after another, or concurrently in an interleaved manner. In the latter case, the processor executes parts of a program, parts of another program, parts of yet another program, and so on and so forth until all program executions are complete.

≫ The concept of a process is fundamental in the domain of operating systems. The operating system keeps track of many concurrent program executions, all in the abstraction of different processes.

Truly speaking, the processor executes only machine instructions sequentially, one instruction after another, and does nothing else. It does not differentiate between the instructions of different programs. The processor executes instructions from the programs depending on the manner they are fed into the processor instruction execution stream. The operating system decides which instructions from which programs the processor would execute next. Although the processor executes one instruction after another of a certain program, the operating system may interleave executions of instructions from other programs. Then, a definite need arises to supervise the various (concurrent) program executions to keep track of which program executions are using which resources, which of their instructions would be executed next by the processor, which program executions are initiated by which users, etc. Each program execution builds an "execution context" as the execution evolves. Program executions would behave correctly only under the respective execution contexts.

The operating system tracks down program executions in the abstraction of processes. A *process* is an entity or object that represents a single computation, and that models the execution context of the computation. That is, a process represents a distinct instance of program execution.

Processes and program executions are in one-to-one correspondence. We often use both terminologies interchangeably to mean the same. The operating system starts an execution of an application program by creating a process, and destroys the process when that execution is complete. That specific instance of the program execution is synonymous with that process. The process is allocated some part of the main memory to hold its program and data. When a process is started, the operating system brings the required program and data into the main memory, and builds up the initial execution context of the process. The operating system allocates

various resources to the process as the program execution evolves. The execution context of the process consists of this resource allocation information, the current state of the program execution, the instruction that is to be executed next, and other information related to the program execution. In short, the operating system uses a process as a "handle" to manage one program execution.

Note that an operating system (or kernel) is not a process, but a collection of programs that the processes may execute. However, there might be a few dedicated processes to carry out various operating-system-specific tasks. These processes execute only operating system programs, and neither the utilities nor the user applications. They are referred to as *operating system processes* or *kernel processes*. The rest of the processes in the system are referred to as *user processes* or *application processes*. User processes execute utilities and application programs. They also execute operating system programs from time to time, but only when they require services from the operating system.

Although we sometimes say a process executes a program, it is the processor that executes machine instructions on behalf of the process under its execution context. We sometimes casually say that the program is being executed by the operating system in the abstraction of the process. At the very beginning of system bootstrapping, for a limited period of time, there is no operating system nor a process in the system. The processor executes a bootstrap program from the boot ROM, first brings a portion of operating system into the main memory, and then executes operating system programs. In this initial phase of bootstrapping, the processor executes operating system programs, though not in the context of any process. The operating system initializes various data structures and creates a few kernel- and user processes. In the midst or at the end of the bootstrapping, the execution control is transferred to a process. From then onwards, the processor always executes machine instructions on behalf of a single (user or kernel) process.

≫ A process may be viewed as the eventual avatar of a solution to a problem. A solution is first abstracted as an algorithm in the software design phase, then transformed into an application program in the programming phase, and finally created as a process to solve the intended problem by executing the corresponding program under the premise of an operating system. Every process has a program as its component and, at any time, is in a state of readiness to execute the program.

1.5.1 Resource Allocation

Most operating systems allow more than one process to be simultaneously present in the computer system. During its lifetime, a process uses many resources to accomplish its tasks on hand. For example, it uses the main memory to hold programs and data, the processor to execute the programs, the printer to print data on papers, the keyboard to read input data, and so forth.

Processes need the processor to make progress in their computations. As there is more than one process in the system, many of them may be ready to use the processor at the same time. The processor is an exclusive or dedicated resource. (It is exclusive in the sense that it cannot be used by more than one process at a time; and can only be used mutually exclusively by processes. This form of resource sharing is called *resource multiplexing* or *time division multiplexing of resource*, and the operating system decides which processes get which resources when and for how long.) Consequently, a process that is ready to run may need to wait until the operating system assigns the processor

≫ There are two ways to share or multiplex a resource: (1) time-multiplexing, where users take turns in using the resource (e.g., processor), and (2) space-multiplexing, where users occupy different parts of the resource simultaneously (e.g., main memory).

to the process. A process may encounter such situations when it has to wait as it tries to use other exclusive resources such as printer.

When an exclusive resource is allocated to a process, the resource is said to be *busy* or *occupied*. Any other process that needs the resource must wait until the resource becomes *free* or *available*. The operating system must keep track of which processes use which resources so that it can manage both resources and processes effectively and fairly. We use the term *allocation* to denote a reservation of a resource by a process. Every resource (exclusive or otherwise) in the system is managed by a program piece called an *allocator*. (Depending on the context, an allocator is also referred to as a *resource scheduler* or *resource manager*; for example, the processor scheduler, the memory manager.) Allocators of resources are part of the operating system. A process interacts with an allocator to acquire (that is, reserve) and release (or free) the resource managed by the allocator.

Resource sharing (or resource multiplexing) is a vital activity in a computer system to improve resource utilization as well as system performance. However, the sharing leads to competition among processes for the resources. The operating system has to ensure that the sharing proceeds smoothly and amicably. When there is more than one process waiting for a busy resource and the resource becomes free, the resource allocator is required to make a decision in finite time of the order the waiting processes will get the resource. To allocate the resource to these competing processes, the allocator follows a set of decision rules called the *allocation algorithm* or *scheduling algorithm*. When executed, a scheduling algorithm may take into account various process-dependent parameters to select a process for the allocation of the resource. The allocator is expected to provide service that is acceptable overall to the processes. For example, the processor scheduler may allocate the processor to processes based on their priorities or in the order of their creation.

An important objective of any operating system is to allocate resources to processes in a fair and effective manner. The system should follow a policy towards the orderly grant of resources to contending processes. It must specify how simultaneous conflicting requests from processes are resolved. For example, a policy statement may suggest that requests be granted in the first-come first-serve order. Resource allocators must enforce these policies when allocating resources to processes. If a request for a resource cannot be granted to a process immediately, the process is put to wait in a queue associated with the resource. Later when the resource becomes free, it is allocated to the most deserving process from the queue. The selected process is removed from the queue, awakened, and allowed to use the resource. We say the process "owns" the resource. The operating system reclaims a resource when the owner process no longer requires it.

≫ A policy specifies what should be done. A mechanism to implement a policy describes how it is done.

1.5.2 Process Address Space

In a single-process system, at the most one process is present in the system. The operating system does not start another process until the execution of the current

one is complete. The (current) application process alternately executes instructions from its own application programs and the programs of the operating system. It executes the latter programs to obtain services from the operating system and to use various system resources. If a process accidentally or intentionally corrupts the operating system programs or data, the functioning of the entire system will be in jeopardy. Application processes must not be allowed to modify operating system programs and data. Consequently, operating system programs and data need protection at all costs from erroneous and malicious application programs. This duty lies upon the operating system and hardware.

In a multiprocess system, many application processes can be simultaneously present in the system. These processes may execute the same or different application programs, but all of them execute the same operating system programs. The operating system, in addition to protecting itself from application processes, has to ensure that one process does not (intentionally or accidentally) modify programs and data belonging to other processes. It is expected that the operating system ensures a protected working environment for each process.

≫ The address space of a process is an abstraction of the storage area allocated to a process where it can keep all its information such as code and data during execution. The physical locations of this space may be scattered in different places such as the main memory, the disk, and others.

It is clear from the above two paragraphs that an operating system has at least two minimal protection tasks at hand: (1) to protect application processes from one another, and (2) to protect itself from them. A part of these tasks of protection is solved by implementing a concept called address space. Loosely, an *address space* consists of many different entities, and these entities have distinct individual addresses in the address space. An *address* here is a name or a number that uniquely identifies an entity in the address space. An entity in the address space is always referenced by using its address relative to the address space. For example, a collection of user programs (and their data) that are allowed to be executed (and accessed) by a process form an address space; it is called the *logical* or *private address space* of the process, and individual addresses in that space are called *logical addresses* (see Fig. 1.8). The process references entities in the address space using their logical addresses. Analogously, the operating system programs and data constitute an address space that we call the *kernel space*. Private address spaces of all processes collectively form what we call the *user-* or *application space*.

≫ We would like to raise a warning flag here for readers. The word logical is a very generic term. As we will see later, it may not always bind to process address space.

Though each application process has its own private address space, all application processes share the common kernel space. At any juncture, a process operates in a single address space—either its private address space or the kernel space. The process is allowed to access entities only from the current address space, and not from elsewhere. That is, when a process executes application programs, it references entities from its private address space; but when it executes operating system programs, it references entities from the common kernel space. It is the duty of the operating system to make certain that processes are confined to their respective current address spaces. The operating system detects any violation of references caused by a process, and, if needed, forcefully terminates the process. It takes the help of the

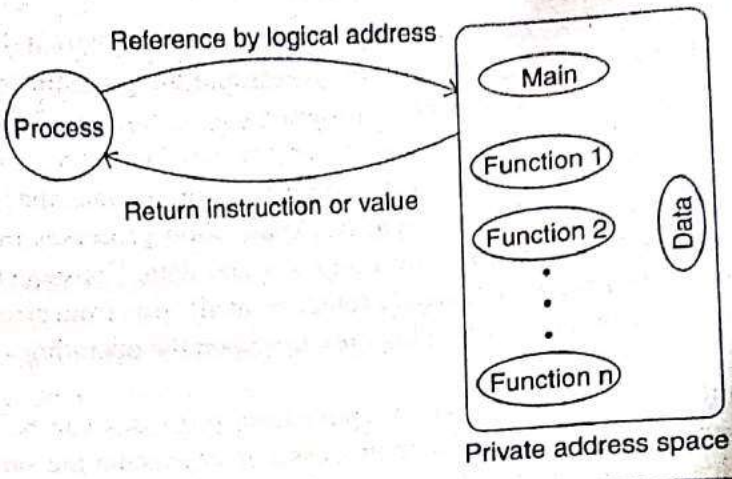

Figure 1.8: *Process address space abstraction.*

processor hardware in detecting reference violations. A slight technical issue arises here from this clean way of defining the two address spaces. As we will see shortly in Section 1.7 on page 24, during a system call, a process can access both the kernel space and its private space. Thus, in theory, there is another type of space that combines both the kernel- and a private address spaces. However, like other authors we call this space also the kernel space. Readers will be able to determine from the context whether we are referring to the pure kernel space or to a combined kernel space (see Fig. 1.9).

A process alternately operates in its private address space and the kernel space; such switching between the two is called *address space switching*. An application process primarily executes application programs in the user space. When it executes operating system programs, it is in the kernel space, and it is considered to have "effectively" become a kernel process. By contrast, the original kernel processes always execute in the (pure) kernel space, and never in the user space; they always execute operating system programs and access operating system data. Kernel processes never perform address space switching.

≫ You may note a difference between application and application process in the context of execution. When we say an application executes an instruction or accesses a resource, the action happens only in the user space. Of course, the action is carried out by an application process. When we say an application process does something, the action happens in ether the user space or the kernel space.

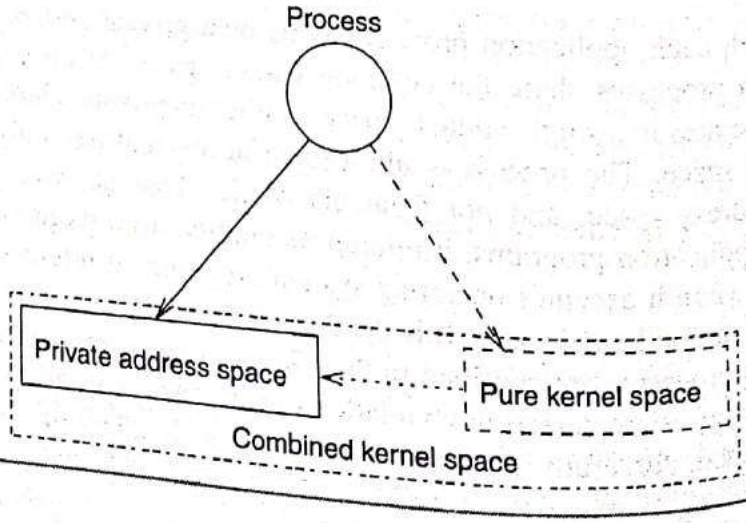

Figure 1.9: *Relationship between pure and combined kernel spaces.*

1.6 Processor Operating Mode

One vital responsibility of operating systems is to ensure protection of various kinds is present in computers. It is stated in Section 1.5.2 on page 20 that application processes are not allowed to access system resources directly from their private address spaces, because direct use of the resources by them might jeopardize the smooth functioning of the computer. If an application process puts a resource into an inconsistent state, it could jeopardize the behaviour of other processes that use the same resource. Consequently, system resources must be prohibited from direct use by application processes from the user space. These resources are a part of the kernel space. Consequently, application programs, operating in the user space, must not be allowed to access entities from the kernel space. Requests for resources from the applications must be channelled through the operating system so that it can coordinate accesses to these resources.

The processor manipulates hardware resources by executing special machine instructions. For example, IN/OUT instructions are executed to read/write data from/to I/O controllers' operating registers. Consequently, computer systems may need to prevent application processes from executing instructions of this nature in the user space. Then, the set of machine instructions available to application programs is a strict subset of the original machine instructions supported by the processor architecture. The instructions that are not available to application programs are called *privileged instructions*. I/O instructions are privileged in modern processors, and consequently, application processes are not allowed to execute I/O instructions when they operate in the user space. The processes can however execute I/O instructions when they operate in the kernel space. Consequently, application programs must not be allowed to make references to operating system programs and data from the user space, because otherwise application processes may take control of hardware resources by manipulating operating system programs. We need a protection scheme to prevent application processes directly executing privileged instructions or operating system programs, and accessing operating system data. What we mean is that application processes cannot be allowed to enter operating system programs (i.e., the kernel space) by executing instructions that permit them to jump from one memory location to another.

Processor architecture enables operating systems to implement the above mentioned protection task. The architecture supports at least two processor-operating modes called (1) the *user mode* and (2) the *kernel mode*. (The kernel mode is also called the *supervisor*, *system*, *monitor*, or *privileged mode*.) The processor runs either in the user mode or in the kernel mode, but never in both. When the processor operates in the user mode, it makes certain resources and instructions (such as IN/OUT) unavailable to processes. Applications are prohibited from executing all privileged instructions, and thereby, prohibited to manipulate hardware resources. If, due to programming errors or due to malicious program design, the processor attempts to execute a privileged instruction in the user operating mode, an exception signal (a kind of interrupt, see Section 1.7 on page 24) is raised by

≫ Nothing that is shared by different applications is allowed to be directly manipulated by the applications themselves from their private address spaces. Operating system programs are supposedly correct and trustworthy. Processes in the kernal space can execute the operating system programs without many restrictions. Protection checks may not be needed for kernel processes.

≫ Some operating systems such as MS-DOS allow application processes to work directly on hardware resources. However, this is considered unusual and very risky. A process can then modify the program and data of another process, causing the latter to behave unpredictably.

≫ The processor hardware implements a flag whose value indicates the current operating mode of the processor. We need at least two values. Some modern processors such as Intel 80386 have more values.

the internal hardware circuits of the processor. The hardware also checks all memory references made in the user mode. Any address violation also leads to an exception. Such exceptions are directed to the operating system. On receiving the exception, the operating system immediately gains control of the computer, takes appropriate actions and may terminate the defective process.

The processor executes application programs in the user-operating mode, and operating system programs in the kernel-operating mode. The rule is never to execute application programs in the kernel-operating mode. Thus, the processor operating modes and the checking of the address by the hardware help the operating system to protect the kernel space from application processes as they will not be able to make direct references to entities in the kernel space from their private address spaces. When an application process needs service from the operating system, it needs to switch the processor to the kernel mode before it starts executing operating system programs. The processor architecture implements special machine instructions to switch the processor from the user mode to the kernel mode and vice versa. Processes execute mode-switching instructions to raise the privilege level of the processor to become eligible to execute privileged instructions (see Fig. 1.10). When an application process executes a mode switching instruction in the user space, the processor hardware ensures that, upon completing the instruction execution, the application program loses the control of the processor and the operating system takes over the control. The operating system executes another mode-switching instruction to return the control back to the application program in the user operating mode.

≫ Address space switching is always carried out by special machine instructions, and not by mere jump instructions.

≫ Whenever the operating system gains control of the computer system the processor is in the kernel mode. When the control returns to an application, the processor is in the user mode. This way the safety of privileged resources is ensured.

1.7 System Call, Interrupt, Exception, and Kernel Path

Application processes execute application programs when the processor is in the user-operating mode. When an application process needs to access system resources or kernel data, it has to execute various operating system service

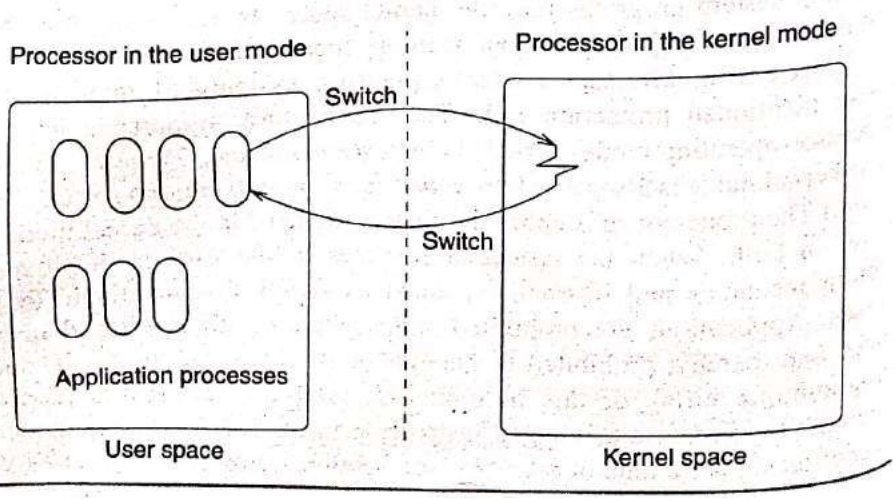

Figure 1.10: *Address space switching by special machine in[illegible]ns.*

programs. However, as noted in Section 1.6 on page 23, a process cannot execute operating system programs when the processor is in the user-operating mode. The operating system defines a set of entry points called *service access points*. The entry points collectively define the programming interface to the operating system. Application processes connect themselves to entry points to obtain services from the operating system. A connection call to an entry point is called a *system call;* the application is calling the system seeking some service from the system.

≫ The supervisor mode of operation takes over when the application program makes a system call. The supervisor mode overrides almost all protection mechanisms in the system, as the operating system is supposed to be defect-free.

A system call is very much like an ordinary function call, but is implemented in a very special way. A process executes a special machine instruction (in the user space) to make a system call. The instruction execution performs two actions at one stroke. First, the instruction execution flips the processor from the user-operating mode to the kernel-operating mode. Second, it alters the current program execution flow, and the processor (in the same process context) starts executing instructions from an operating system routine called *system call handler*. That is, the (system call) instruction execution teleports the process from its private address space to the kernel space, and the process becomes "effectively" a kernel process for the duration of the system call, and we say the kernel is being executed on behalf of the process. Depending on the type of the call, the system call handler invokes the appropriate operating system service routines. When the execution of the system call handler is complete, the kernel executes another special instruction and the execution control returns to the original program from where the system call originated, and the processor automatically reverts to the user-operating mode. The system call makes an address space switch, but as pointed out in Section 1.5.2 on page 20, this is not a switch to the pure kernel space. The process, while in the kernel space, can reference entities from its private address space; for example, to read input values of the system call parameters.

≫ Teleportation refers to the instantaneous transportation of a particle or element from one space into another without transiting through the two spaces via their boundaries.

In its lifetime, an application process alternately executes application programs in its private address space and operating system programs in the kernel space. Figure 1.11 shows a schematic of an application process that switches between the two spaces. (Note that when a system call is made, the processor executes operating system programs on behalf of the calling process.) It is stated in Section 1.5.2 on page 20 that kernel processes always run in the (pure) kernel space. Consequently, the processor always executes the programs in the

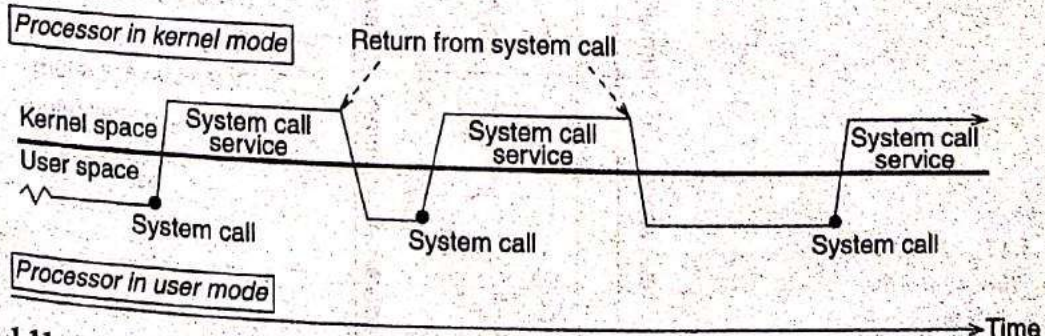

Figure 1.11: *Process switching between the user space and the kernel space by system calls.*

kernel-operating mode, and never in the user-operating mode. Kernel processes do not need to make system calls to execute operating system service routines. They can directly execute any operating system routine, or access operating system data.

Different operating systems have different mechanisms for making system calls. Ways of passing parameters to system calls also vary. Thus, invocations of system calls may appear a little difficult to naive application developers. To ease application development, system designers implement a set of APIs through which the applications interact with the operating system. These APIs, in turn, will make system calls to obtain appropriate operating system services for applications. The applications, in general, do not directly make system calls. Instead, they invoke these APIs like ordinary function calls. These APIs are installed in various system libraries that are accessible to application developers. For example, in UNIX systems, the libc library contains implementations of APIs (read, write, sleep, etc.) for C language-based applications. Figure 1.12 shows how application processes make system calls through these libraries. Note that the libraries reside in the user space.

There are two other situations when the processor is flipped to the kernel operating mode: (1) the interrupt and (2) the exception. It is stated in Sections 1.2.5 and 1.4 that interrupts are signals that are sent to the processor by the peripheral (and some internal) devices. The devices raise interrupt signals to draw the attention of the processor to them. On receiving an interrupt signal, the processor changes its operating mode to the kernel mode. The processor simultaneously suspends the execution of the current program, and commences executing an interrupt service program that resides in the kernel space. When the execution of the interrupt service program is complete, the processor reverts to the original operating mode right when it was interrupted. An interrupt may come to the processor at any point of time, and hence, it is called an *asynchronous* event. The processor, however, services the interrupt only after the completion of the execution of the ongoing instruction. The interrupt service program is always executed in the context of the interrupted process even though the service is normally for some other process. Figure 1.13 shows how an application process reacts to interruptions.

≫ We say that interrupts are serviced in an "arbitrary" process context. (Some authors call it the system context.) In contrast, system calls are serviced in particular process contexts. During the interrupt service, the "arbitrary" process is in the "pure" kernel space, and it cannot access data from the process's private address space.

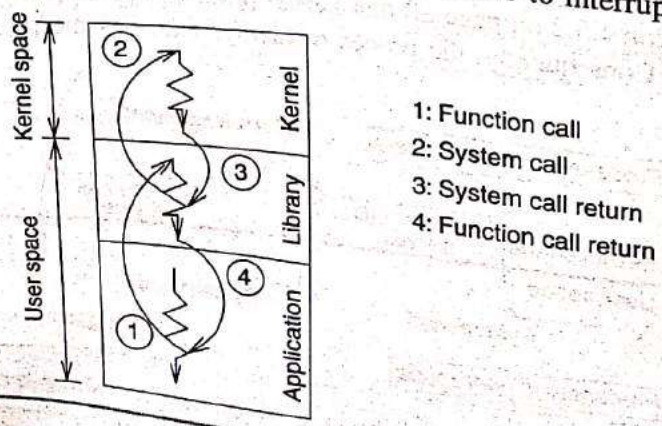

Figure 1.12: *Process execution flows into the operating system through a library.*

Unlike interrupts, exceptions are *synchronous* events raised by the internal processor hardware circuits. An exception is an internal event that occurs during an instruction execution when the execution has envisaged an unusual condition. Exceptions may arise because of several reasons such as execution of an illegal instruction, division by zero, arithmetic overflow, address space violation, protection violation, and so forth. Some of these are due to program defects, and (as we will see later) some by operating system design choices. Unlike interrupts, hardware circuits raise exceptions only at the end of instruction executions, and never in the midst of one. Like interrupts, exceptions disrupt the normal flow of instruction executions. The operating system needs to resolve these exceptions and to take appropriate actions to rectify the exceptional situation or to terminate the process. The operating system in general handles exceptions in the same ways it services interrupts.

≫ Exceptions are synchronous because the hardware raises them only after aborting the current instruction execution.

For each of these three services (system call, interrupt, and exception), the processor executes a sequence of instructions from operating system programs. In each case, we say the operating system has gained control of the computer, and the system is in the kernel operating mode, and we often say the kernel or operating system is executing itself. This sequence of instruction executions is called a *kernel execution*, and is referred to as a *kernel path* in this book to distinguish it from an application program execution. Obtaining an operating system service implies executing a specific kernel path by the processor. Generally, kernel paths are executed sequentially (one after another, as shown in Figs. 1.11 and 1.13), in a mutually exclusive manner. Several kernel paths, however, may be interleaved by the interrupts and the exceptions or if the current kernel path suspends itself for some reason. Figure 1.14 displays an execution scenario where two kernel paths interleave due to an interrupt while a process has been executing a system call. Note that, in the example, when the kernel path due to the interrupt service returns, the processor remains in the kernel-operating mode and resumes executing the suspended/interrupted kernel path. The processor always switches back to the user-operating mode only when execution of an application program resumes.

≫ In a multiprocessor system, where many processors are present in the computer system, the processors may execute different kernel paths simultaneously.

We specifically note one point here. An interrupt or exception handler is not a process; but a kernel path that runs in the context of the interrupted process. As mentioned previously, the handler is said to be running in "arbitrary" process context. It is a kind of context switch, though not a

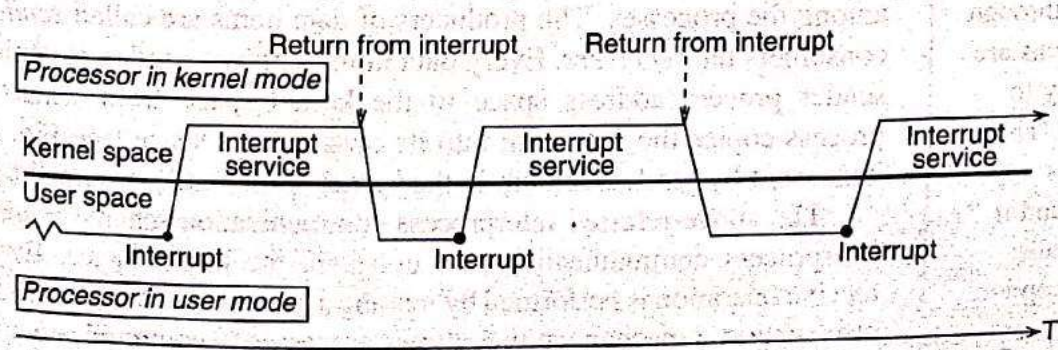

Figure 1.13: *Process switching between the user space and the kernel space by interrupts.*

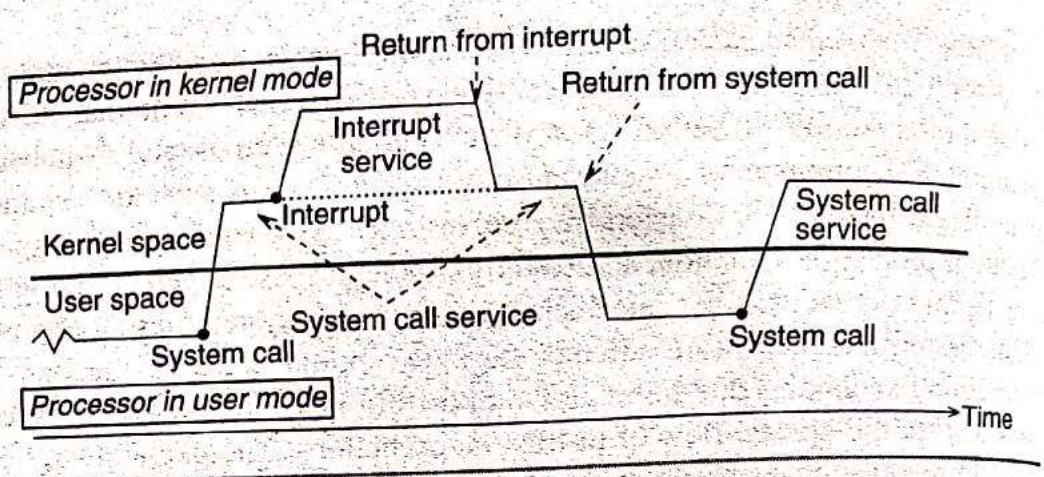

Figure 1.14: *Interleaved execution of two kernel paths.*

process switch. Depending on the scenario, it may cause an address space switch too. All interleaved kernel paths run in the context of the same process until the process suspends itself or returns to the user space.

1.8 Interprocess Communication and Process Synchronization

There are applications that involve each executing multiple processes concurrently. Processes work together to perform application-specific tasks. They are referred to as *cooperating processes*. Cooperating processes are "loosely" connected in the sense that they have independent private address spaces and run at different speeds. The relative speeds of the processes are not normally known. From time to time, they interact among themselves by exchanging information. An exchange of information among processes is called an *interprocess communication*.

≫ System commands (or simply commands), system calls, and interrupts are the ways to interact with the computer system. Users interact with the system through commands (to a command interpreter), processes through system calls, and devices through interrupts. Exceptions are generated internally to indicate violations. The operating system is a reactive program and it reacts to system calls, interrupts, and exceptions.

It is stated in Section 1.5.2 on page 20 that one process cannot access elements from the private address space of another. Such access violations cause address exceptions, and the operating system traps them. This implies that processes need assistance from the operating system to set up communication facilities among themselves. Most operating systems implement a few different communication schemes to facilitate interprocess communications. For each scheme, they support a few communication primitives (interface operations). Processes execute these primitives to exchange information among themselves.

Ultimately, all information exchanges involve exchanging data items among the processes. The producers of data items are called *senders*, and the consumers the *receivers*. Every data item, sent by a sender, is copied from the sender process address space to the kernel space from where a receiver process copies the data item into its own address space (see Fig. 1.15). Data items sent by senders remain in the kernel space until consumed by receivers.

The above-referred interprocess communication scheme is indirect as all interprocess communications are done via the kernel space. Every send or receive operation is performed by making a system call. Most operating systems also support a mechanism that allows processes to communicate among themselves directly without copying data via the kernel space. To do this a "shared memory region" is set up among the cooperating processes (see Fig. 1.16). The

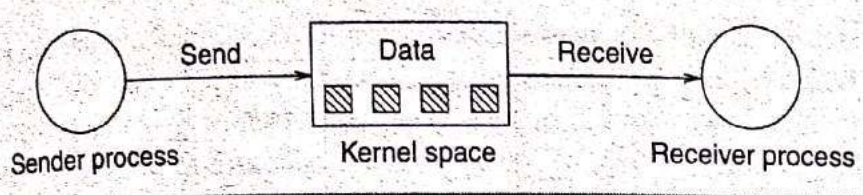

Figure 1.15: *Interprocess communications via the kernel space.*

operating system maps parts of the process address spaces to the same physical memory locations so that all processes having access to the shared memory region immediately observe changes that may be made in that region. Note that shared memory regions reside in the user space, and not in the kernel space.

In indirect communication schemes, processes make system calls for each interprocess communication (send or receive of a data item). By contrast, in shared memory-based direct communication schemes, processes directly communicate by reading and modifying values in the common physical memory locations. As these memory locations are mapped to the user space, the processes do not need to make system calls to access contents of these locations. Nonetheless, they create, attach, detach, and destroy shared memory regions through system calls.

Every interprocess communication scheme (direct or indirect) can be modelled as schematics shown in Fig. 1.17. Ultimately, all communications involve storing and retrieving information in some shared medium. The senders store information in the medium, and the receivers retrieve the information from it. For easier manipulation, the information is structured into various data objects called *shared data* or *shared variables*. Shared variables are units of information referenced by processes. Each shared variable has a unique name (or address) and a *type*. The type defines a finite domain of values, interface operations, and consistency semantics. The variable can store any value from the domain. Interface operations are the only means to access the shared variable. The semantics of each operation describe the permitted behaviour of executions of the operation. Processes communicate among themselves by manipulating values of shared variables by executing

≫ Unlike space- or time-division multiplexing, shared regions represent true sharing of the main memory locations.

≫ Communication between processes may be explicit (by specifically addressing the data or message to the receiver process(es)) or implicit (by writing on shared variables).

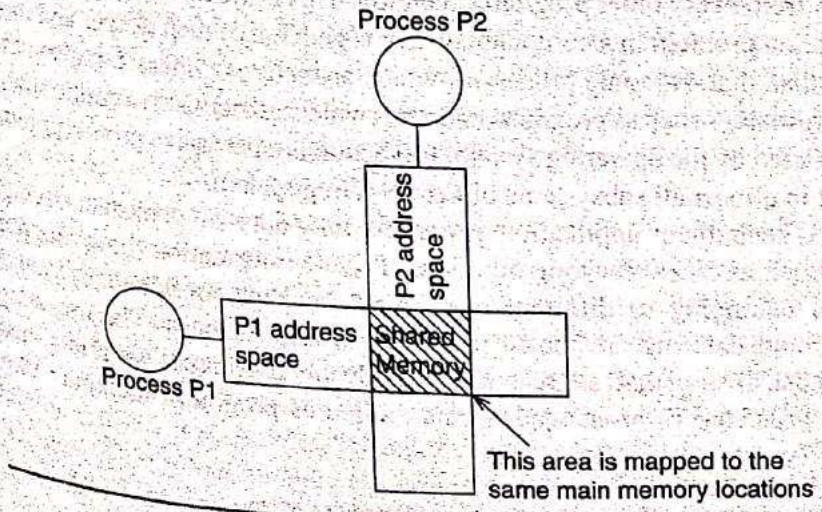

Figure 1.16: *Interprocess communications through address space sharing.*

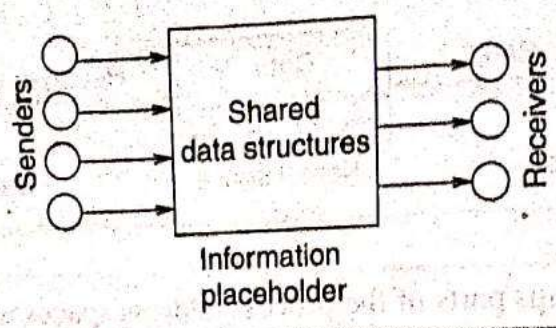

Figure 1.17: *A typical interprocess communication facility.*

operations supported by these variables. Interprocess communication schemes vary and use shared variables of different types.

Each process executes its own operations on shared variables sequentially, as specified by its own program. Nevertheless, different processes may execute their operations on the same shared variable concurrently.[6] That is, operation executions of different processes may overlap, and may affect one another. Note that each operation, when executed indivisibly on a shared variable, transforms the variable from one consistent value to another. However, when the operations are executed concurrently on a shared variable (without any access control), the consistency of its values may not be guaranteed. The behaviour of operation executions on shared variables must be predictable for effective interprocess communications. Thus, operation executions on shared variables may need to be coordinated to ensure their consistency semantics. Coordination of accesses to shared variables (or shared space) is called *synchronization*. Most operating systems implement a few different synchronization schemes for process coordination purposes. Each scheme supports a set of primitives. The primitives are used when it is absolutely necessary to order executions of operations on shared variables in a particular manner.

1.9 Protection and Security

In Sections 1.5.2 and 1.6, we talked about the necessity of protecting the kernel space from application processes, and private address space of one process from other processes. It is the task of the operating system to protect every program execution from every other program execution. This is the tip of the iceberg, and the protection problem in a computer system is generally much deeper and more involved than that. Not only processes (which are active entities), but also other software resources such as programs and data (which are non-active entities) need to be protected by the operating system. Protection is a pervasive property, and it is present in almost all subsystems of the operating system.

Users keep their application programs and data in persistent storage devices (such as disks) for long durations. Private information (both program and data) belonging to different users must be kept confidential. System programs and data also reside in persistent storage devices. The operating system must ensure that all resources (hardware resources, programs, and data) are protected from erroneous- and malicious program executions. Not

[6] By concurrency we mean either simultaneous or interleaved executions.

all users are allowed unhindered access to every resource. That is, each user is "authorized" to access only a subset of resources, that too in a restricted way. Unauthorized accesses to resources must be trapped by the operating system. By doing so, it ensures protection in the system. In short, the privacy of user data, programs, and activities needs to be ensured by the operating system.

Operating systems store programs and data in the abstraction of files. A *file* is a sequence of primitive data values (that are bytes in most systems), and it stores related information. Files are logical units of information storage. Some user owns each file, and the owner may expect that access to it is limited to *authorized users*, because the file may contain valuable confidential programs and/or data. The operating system must ensure the privacy of the file contents by concealing the file from unauthorized users.

Some files store application programs and some system utilities. Not all users may be authorized to execute all these programs. The operating system must ensure that users cannot execute unauthorized programs. Even when a user executes an authorized program, the corresponding application process must not access unauthorized data files. Any intentional (or erroneous) unauthorized accesses to files must be denied by the operating system.

Operating systems must provide some level of security for computer systems. Security deals with protecting a computer system from the external environment. A computer system can be used only by a known set of users, called *legal users*, and cannot be used by illegal users. Security is ensured by various protection mechanisms employed in the system itself. Every legal user in a computer system is identified to the system by a user identifier (login name, number, or access card) and a secret password or token. When a user tries to begin a session with the system, she provides her login name and password to the system. The operating system *authenticates* the validity of the login name and password pair, and she successfully starts a new login session. For the purpose of authentication, the operating system keeps the login name and the password pair information in persistent storage devices. The password information must not be available (readable or inferable) to any user except a few authorized entities (system administrators). Once a legal user logs into a system, she will have restricted access rights to various resources. She will have rights to those resources that are available to her login name. Each user in the system is associated with a set of capabilities that specify what resources she can use in what manner, for example what files she can read, write, or execute. The operating system ensures that she does not cross her capability barrier either intentionally or accidentally.

≫ Broadly, protection deals with internal authorization and security deals with external authorization. In most systems, security is effective at login time. Once a user logs in, the protection mechanisms become effective for her.

Ensuring information security in a computer system is a considerable challenge. Security related problems become a formidable challenge to handle for system designers when many computers are connected together to form a network of computers. Each individual computer system not only needs to take care of local users, but also those who access its resources through remote computers. In such systems, in addition to protecting internal resources, computers need to protect information that is exchanged among themselves through the network. This is achieved by using various authentication and encryption/decryption schemes.

1.10 Execution Semantics

We introduced some core hardware- and software concepts in previous sections. The purpose of introducing these concepts upfront is to avoid distractions that may occur due to forward references in later chapters. In this section, we draw the reader's attention to the possible meanings of the term "execution". This word is very important to us. Readers might be a little bit perplexed over the use of the term "execution" so far. The term is indeed used in various contexts to mean different, but very closely related things. Execution is an "overloaded" term in this book, and it is used at various levels of abstractions. The following are a few examples that show how the term is used in different contexts. We strongly believe that readers will be able to grasp the right meanings of execution from the contexts of its usage.

≫ User is also a generic term and is used for many purposes. The term may not necessarily mean a human being. For example, the users of an operating system are applications.

1. Instruction execution: Processor, and no other hardware component, executes machine instructions. That is the reality.
2. A processor executes a program: We mean that the processor executes instructions from the program and does nothing else.
3. A processor executes a process: We mean that the processor executes instructions from programs in the current address space of the process.
4. Operating system (or kernel) executes a program: We mean that the operating system (or the kernel) has selected the program for execution on a processor.
5. Operating system (or kernel) executes a process: We mean that the operating system (or the kernel) has scheduled the process for execution on a processor.
6. A process executes an instruction: We mean that a processor executes the instruction in the context of the process.
7. A process executes a program: We mean that a processor executes the program in the context of the process.
8. A process executes the operating system (or kernel): We mean that a processor executes the operating system (or kernel) programs in the context of the process.
9. A user executes (or runs) an application program: We mean that a processor executes instructions from the application program in the context of a process on behalf of the user. The user is said to be the *owner* of the process.

1.11 Classification of Operating Systems

Many operating systems have been designed and developed in the past several decades. They may be classified into different categories depending on their features: (1) multiprocessor, (2) multiuser, (3) multiprogram, (3) multiprocess, (5) multithread, (6) preemptive, (7) reentrant, (8) microkernel, and so

forth. These features, and the challenges involved in implementing them, are discussed very briefly in the following subsections.

1.11.1 Multiprocessor Systems

A *multiprocessor system* is one that has more than one processor on-board in the computer. They execute independent streams of instructions simultaneously. They share system busses, the system clock, and the main memory, and may share peripheral devices too. Such systems are also referred to as *tightly coupled* multiprocessor systems as opposed to network of computers (called distributed systems).

A uniprocessor system can execute only one process at any point of real time, though there might be many processes ready to be executed. By contrast, a multiprocessor system can execute many different processes simultaneously at the same real time. However, the number of processors in the system restricts the degree of simultaneous process executions.

There are two primary models of multiprocessor operating systems: symmetric and asymmetric. In a *symmetric multiprocessor* system, each processor executes the same copy of the resident operating system, takes its own decisions, and cooperates with other processors for smooth functioning of the entire system. In an *asymmetric multiprocessor* system, each processor is assigned a specific task, and there is a designated master processor that controls activities of the other subordinate processors. The master processor assigns works to subordinate processors.

In multiprocessor systems, many processors can execute operating system programs simultaneously. Consequently, kernel path synchronization is a major challenge in designing multiprocessor operating systems. We need a highly concurrent kernel to achieve real gains in system performance. Synchronization has a much stronger impact on performance in multiprocessor systems than on uniprocessor systems. Many known uniprocessor synchronization techniques are ineffective in multiprocessor systems. Multiprocessor systems need very sophisticated, specialized synchronization schemes. Another challenge in symmetric multiprocessor systems is to balance the workload among processors rationally. Multiprocessor operating systems are expected to be fault-tolerant, that is, failures of a few processors should not halt the entire system, a concept called graceful degradation of the system.

1.11.2 Multiuser Systems

A *multiuser system* is one that can be used by more than one user. The system provides an environment in which many users can use the system at the same time or exclusively at different times. Each user can execute her applications without any concern about what other users are doing in the system. When many users run their applications at the same time, they compete and contend for system resources. The operating system allocates them the resources in an orderly manner.

Security is a major design issue in multiuser operating systems. Each user has a private space in the system where she maintains her programs and data, and the operating system must ensure that this space is visible only to her and authorized ones, and is protected from unauthorized and malicious users. The system needs to arbitrate resource sharing among active users so that nobody is starved of system resources. Multiuser systems may need an accounting mechanism to keep track of statistics of resource usage by individual users.

1.11.3 Multiprogram Systems

A *multiprogram system* is one where many application programs can reside in the main memory at the same time (see Fig. 1.18). (By contrast, in uniprogram systems, at the most one application program can reside in the main memory.) Applications definitely need to share the main memory, and they may also need to share other system resources among themselves.

Memory management is a major design challenge in multiprogram operating systems. Multiplexing of the main memory is essential to hold multiple applications in it. Different standalone applications should be able to share common subprograms (procedures) and data. Processor scheduling (long-term) is another design issue in such systems as the operating system needs to decide the best applications to bring in the main memory. Protection of programs from their own executions is another issue in designing such systems.

1.11.4 Multiprocess Systems

A *multiprocess system* (also known as multitasking system) is one that executes many processes concurrently (simultaneously or in an interleaved fashion). In a uniprocess system, when the lone process executes a wait operation, the processor would sit idle and waste its time until the process comes out of the wait state. The objective of multiprocessing is to have a process running on the processor at all times, doing purposeful work. Many processes are executed concurrently to improve the performance of the system, and to improve the

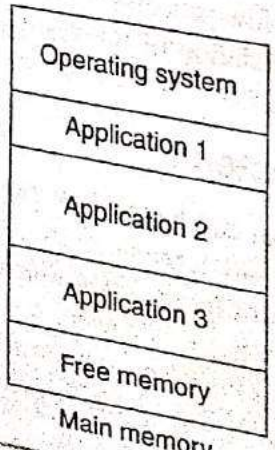

Figure 1.18: *Multiplexing of the main memory by applications.*

utilization of system resources such as the processor, the main memory, disks, printers, network interface cards, etc. Processes may execute the same program (in uniprogram systems) or different programs (in multiprogram systems). They share the processor among themselves in addition to sharing the main memory and I/O devices. The operating system executes processes by switching the processor among them. This switching is called *context switching*, *process switching*, or *task switching*.

Short-term process scheduling is a major design issue in multiprocess systems. Multiprocess systems need to have schemes for interprocess communications and process synchronizations. Protection of one process from another is mandatory in multiprocess systems. These systems, of course, need to provide operations for process creation, maintenance, suspension, resumption, and destruction.

1.11.5 Time-sharing Systems

In an interactive system, many users directly interact with the computer from terminals connected to the computer system. They submit small execution requests to the computer and expect results back immediately, after a short enough delay to satisfy their temperament. We need a computer system that supports both multiprograms and multiprocesses. The processes appear to be executing simultaneously, each at its own speed. Apparent simultaneous execution of processes is achieved by frequently switching the processor from one process to another in a short span of time. These systems are often called *time-sharing systems*. It is essentially a rapid time division multiplexing of the processor time among several processes. The switching is so frequent, it almost seems each process has its own processor. A time-sharing system is indeed a multiprocess system, but it switches the processor among processes more frequently. Thus, one additional goal of time-sharing is to help users to do effective interaction with the system while they run their applications.

≫ A time-sharing system provides each process an illusion that it has the whole computer at its disposal.

1.11.6 Multithread Systems

A *thread* is an independent strand that executes a program concurrently with other threads within the context of the same process. A thread is a single sequential flow of control within a program execution. Each thread has a beginning, a sequence of instruction executions, and an end. At any given point of time, there is one single point of execution in each thread. A thread is not a process by itself. It cannot run on its own; it always runs within a process. Thus, a multithreaded process may have multiple execution flows, different ones belonging to different threads (see Fig. 1.19). These threads share the same private address space of the process, and they share all the resources acquired by the process. They run in the same process execution context, and therefore, one thread may influence other threads in the process.

Different systems implement the thread concept differently. Some systems have user-level library routines to manage threads in a process. An application process can be multithreaded, but the operating system sees only

≫ A conventional process is a single threaded program execution.

the process and not the contained threads. When any thread makes a system call and is blocked, the entire process is blocked too, and no other threads in the process can make any progress until the former thread returns from the system call. No change in the operating system is required for thread handling. (We often say the operating system is single threaded, but applications are multithreaded.) In some other systems, every thread has a kind of process entity called *lightweight process* (LWP) in the operating system. The LWPs in a process are truly independent strands. If one LWP is blocked in the operating system, other sibling LWPs in the process can make progress in their executions. These systems are truly multithreaded as the threads are visible to the operating system. These systems need to provide support for LWP creation, maintenance, scheduling, and synchronization.

≫ You may note a difference between preemption and interrupt. An interrupt breaks the current program execution and starts a new program execution in the same process context. Preemption also breaks the current program execution, but starts a new program execution in a different process context.

1.11.7 Preemptive Systems

Here, preemption means taking away of the processor from a process and allocating it to another process. Preemption can take place any time, whether the process is in the user space or in the kernel space. Most modern systems allow arbitrary preemption in the user space. (Without such preemption, time-sharing of the processor is not possible.) They may or may not have arbitrary preemptions of kernel executions. In the rest of this section, by a preemptive system we mean a preemptive kernel.

≫ It is substantially easier to design and develop non-preemptive systems compared to preemptive ones.

Designers of an operating system must decide whether the system may or may not arbitrarily interleave executions of different processes while the processes execute operating system programs. In a *non-preemptive system*, when the processor executes the operating system on behalf of a process, the processor cannot be arbitrarily preempted (that is, taken away) from the process and allocated to another process. The processor scheduler is called upon only when the current process, running in the kernel space, voluntarily relinquishes the processor or is about to return to the user mode. In a *preemptive system*, the operating system, in addition, may preempt the processor from a process (in the kernel space) at any point in time and allocate the processor to another process. Many general-purpose operating systems are non-preemptive, but they implement a few selective preemption points where a process can be forced to relinquish the processor while it is in the kernel space. There are operating systems that claim to be fully preemptive. They

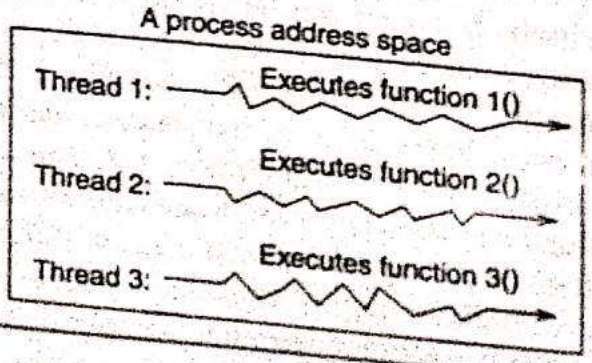

Figure 1.19: *Multiple threads of execution flows in a process.*

are actually partly preemptive because there are some regions in the kernel where kernel executions are non-preemptive.

1.11.8 Reentrant Kernels

A *reentrant* program is one that does not modify itself and any global data. Multiple processes or threads can execute reentrant programs concurrently without interfering one another. They can share reentrant programs, but have their own private data. A *reentrant kernel* is one where many processes/threads can execute the same kernel programs concurrently without affecting one another. In non-reentrant kernels, a process does not modify kernel programs, but can modify global kernel data. Consequently, if a process is executing operating system programs, no other processes may be allowed to execute the programs, nor may the system start another kernel path execution (due to an interrupt or exception) when the kernel accesses the global data. Consequently, interrupts from I/O devices may not be handled immediately by the operating system.

Generally, an operating system is composed of both reentrant and non-reentrant functions. The reentrant functions may modify local (on-stack) data, but they do not modify any global data. Concurrent executions of reentrant function(s) do not affect the behaviour of one another. The operating system needs to make sure that non-reentrant functions are executed mutually exclusively by kernel paths so that the function executions do not modify global data at the same time.

≫ In multiprocessor systems, many processes may execute the kernel simultaneously. In uniprocessor systems, concurrency is only achieved in the form of execution interleavings; only one process can make progress in the kernel mode, while others are blocked in the kernel waiting for processor allocation or some events to occur.

1.11.9 Monolithic Kernels and Microkernel Systems

A *monolithic* kernel is a large single piece of code, composed of several logically different program pieces. It is one single large program where all the functional components of the operating system have access to all the data and routines. (All those components reside in the kernel space.) Such a program, with the passage of time, grows more and more complex, and becomes difficult to maintain. To avoid these problems, modern monolithic kernels are structured in strictly functional units. One unit cannot directly access data and routines belonging to other units. The units follow strict communication protocols to avail services from one another. Nevertheless, whatever be the internal structure, every part of the operating system runs in the kernel mode on behalf of the running process.

A *microkernel system* is one that provides only the bare minimum functionalities in the kernel, and hence, is quite small and compact. The aim is a kernel that provides the most basic functionalities to construct the minimal operating system services: a few interprocess communication and synchronization primitives, a processor scheduler, and an interrupt dispatcher. The remaining functionalities are implemented through autonomous processes called operating system service processes. (These are also called *servers*; in contrast, application processes are called *clients*.) The server programs reside

≫ Linux and Windows are monolithic kernel systems. We will study these two systems later in this book.

outside the kernel in different address spaces (see Fig. 1.20). Thus, servers are not *truly* kernel processes. They are just like any other utility process. They run in the user space, and not in the kernel space. The user-level server processes provide services that have been traditionally parts of the kernel executed by client processes when they operate on the kernel space. They include services such as memory manager, device driver, system call handler, and others. The services offered by the servers are called from the kernel via upcalls. A client does not interact directly with a server. Clients in fact are not aware of servers, and they interact with the operating system via regular system calls. This approach helps in making the kernel smaller. It is easier to develop and test new services or to modify existing services in such systems.

≫ A monolithic system is a "self service" system in the sense that each application process executes the operating system programs to serve itself.

The operating system development code base is relatively easy to maintain for microkernel-based systems. However, it leads to slower execution of operating system services as there are high overheads involved in frequent communications among application processes and service processes via the kernel space. To obtain services from the operating system, application processes communicate with the kernel that channels the service requests to appropriate service processes. The replies to the service requests follow a similar path in the reverse direction. This extra level of indirect interprocess communications slows down the speed of application execution considerably whenever they need frequent operating system services. Each service request leads to at least two process switches: one from the client to the server and the other from the server back to the client. Monolithic kernel is bulky, but ensures relatively fast responses to service requests compared to microkernel-based operating systems. In this book, we restrict ourselves to monolithic kernels.

≫ Mach, from Carnegie Mellon University, is an example of a microkernel operating system. It implements a minimal kernel to handle thread scheduling, message passing, virtual memory, and device drivers. Everything else, including process manager, memory manager, I/O manager, file management systems, networking, and various APIs, runs in the user mode as separate processes in their own private address spaces. Symbian, used in cellphones, is another example of microkernel operating system.

1.12 Designing Operating Systems

We discussed many types of operating systems and their design and development challenges in the previous section. In this section, we present general guidelines of the various challenges encountered in the construction of new operating systems. Designing a new operating system is no doubt a challenging task, but it is not radically different from any large software product design. The key to success of any new software product is to aim at realistic and well-defined goals. Designers need to have a clear idea about what features (services) the operating system will have for its users. Goals of the new operating system must be stated clearly, because a clear understanding of

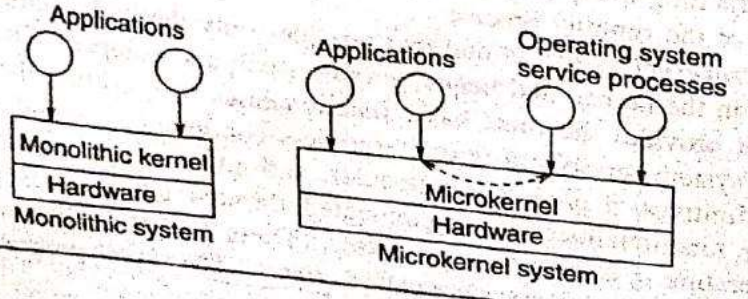

Figure 1.20: *Layout of monolithic kernel vs. microkernel operating systems.*

are actually partly preemptive because there are some regions in the kernel where kernel executions are non-preemptive.

1.11.8 Reentrant Kernels

A *reentrant* program is one that does not modify itself and any global data. Multiple processes or threads can execute reentrant programs concurrently without interfering one another. They can share reentrant programs, but have their own private data. A *reentrant kernel* is one where many processes/threads can execute the same kernel programs concurrently without affecting one another. In non-reentrant kernels, a process does not modify kernel programs, but can modify global kernel data. Consequently, if a process is executing operating system programs, no other processes may be allowed to execute the programs, nor may the system start another kernel path execution (due to an interrupt or exception) when the kernel accesses the global data. Consequently, interrupts from I/O devices may not be handled immediately by the operating system.

Generally, an operating system is composed of both reentrant and non-reentrant functions. The reentrant functions may modify local (on-stack) data, but they do not modify any global data. Concurrent executions of reentrant function(s) do not affect the behaviour of one another. The operating system needs to make sure that non-reentrant functions are executed mutually exclusively by kernel paths so that the function executions do not modify global data at the same time.

≫ In multiprocessor systems, many processes may execute the kernel simultaneously. In uniprocessor systems, concurrency is only achieved in the form of execution interleavings; only one process can make progress in the kernel mode, while others are blocked in the kernel waiting for processor allocation or some events to occur.

1.11.9 Monolithic Kernels and Microkernel Systems

A *monolithic* kernel is a large single piece of code, composed of several logically different program pieces. It is one single large program where all the functional components of the operating system have access to all the data and routines. (All those components reside in the kernel space.) Such a program, with the passage of time, grows more and more complex, and becomes difficult to maintain. To avoid these problems, modern monolithic kernels are structured in strictly functional units. One unit cannot directly access data and routines belonging to other units. The units follow strict communication protocols to avail services from one another. Nevertheless, whatever be the internal structure, every part of the operating system runs in the kernel mode on behalf of the running process.

A *microkernel system* is one that provides only the bare minimum functionalities in the kernel, and hence, is quite small and compact. The aim is a kernel that provides the most basic functionalities to construct the minimal operating system services: a few interprocess communication and synchronization primitives, a processor scheduler, and an interrupt dispatcher. The remaining functionalities are implemented through autonomous processes called operating system service processes. (These are also called *servers*; in contrast, application processes are called *clients*.) The server programs reside

≫ Linux and Windows are monolithic kernel systems. We will study these two systems later in this book.

goals provides us visualizing the challenges lying ahead for the development task. Goals must be possible to realize with available hardware support and development tools at hand. There is no single right solution to the design and development of operating systems. Different operating system development groups have experimented with various approaches. It is better if designers have some experience in software engineering practices.

Hardware resources support many features that enable operating systems to function in a safe and efficient manner. There are operating system features that need direct support from hardware. Thus, the very first step in designing an operating system is to understand the individual hardware resources thoroughly for the target computers. The development of an efficient operating system is possible only if the designers and developers have an in-depth knowledge of all available features (and limitations) of the individual hardware resources to take advantage of the features.

Note that an operating system is merely a manager of all system resources. Designers should state various policies adopted to allocate and reclaim resources. They must have ways to keep resources as effectively occupied as possible. The promotion of resource sharing is one objective of every operating system. At the same time, the operating system must satisfy rigorous protection and security requirements, if there are any.

Any large- and complex software is developed by partitioning it into a number of components. The advantage is twofold: (1) The complexity of any large system is more manageable when it is decomposed into relatively smaller and appropriate components; (2) Decomposition facilitates code reuse. Likewise, an operating system is partitioned into a number of well-defined subsystems. The subsystems interact with one another to achieve their goals. The decomposition is then reiterated over subsystems too such that it results in a hierarchically structured system. A well-structured system is easier to understand, develop, and maintain. Performance is another design consideration. Nevertheless, one should keep the design as simple as possible even if it may not produce a super-efficient operating system. Complex designs often lead to unreliable, defective systems.

It is better that each subsystem is designed for a single specific purpose. One may have a subsystem for each type of hardware resource. These subsystems directly manage hardware resources. Based on these lower-level hardware-tuned subsystems, one may develop higher-level subsystems for process management, interprocess communication, process synchronization, memory management, file management, etc. The interface specification of each subsystem is crucial for interactions between subsystems. Interactions between subsystems need to be clearly defined and closely controlled. Each subsystem should be simple, efficient, and reliable. The most notable subsystem is the application programming interface to the operating system, that is, the system call interface. System calls are the sole means for applications to obtain services from the operating system. The application interface must be precisely defined, and should not change in future upgrades of the operating system. While designing subsystems, the designers need to keep two

≫ Designing interfaces is the fundamental design activity in any software project. Otherwise, we will be quickly flooded with backward compatibility problems.

broad goals in mind, namely user convenience in utilizing computers and efficient utilization of hardware resources.

The operating system resides directly on the hardware platform. Consequently, its design depends on the underlying hardware platform components and the features they provide. However, an operating system should have only a small fraction of the code that is directly dependent on the platform. This would help the system to be easily portable across different hardware platforms. One very important component of the hardware platform is the main memory. The designers must keep the finite size of individual memory locations in mind. Overflows and underflows of these locations may cause failures. It is sometimes difficult to diagnose root causes of these kinds of failures.

It will be an added benefit if the same development code base can be used to produce multi-modal operating systems such as embedded, real-time, general purpose. A *customizable* or *adaptable* operating system is one that allows for flexible modification of system policies. Developers can take advantage of compile time directives in achieving this objective. Compile directives and installation-time parameter values are often used to make operating systems customizable to the requirements of divergent customers, even those with conflicting needs. Customization is an important aspect of the operating system development code. For example, operating system requirements for cell phones differ from those for routers; they may have different processor scheduling algorithms. Nevertheless, it would be very convenient if we could use the same code base for both purposes.

Design and development tasks should be documented well. All design changes must be documented. Interface specifications of all subsystems and their interactions must be included in the document. Constituents of development teams change over time. Newcomers join the teams in the middle of development. They may need these documents to build up their basic knowledge of various system components.

≫ General purpose operating system are huge, for example Windows XP is close to 40 million lines of code, developed by several thousands of people over many years.

Summary

Many different operating systems for different hardware platforms were developed in the past several decades. Computer hardware technologies and operating systems have been evolving since the early 1950s. The operating system evolution centres around two primary goals, namely user convenience in using computers and efficient utilization of resources. We discussed one evolution scenario in this chapter. The evolution occurred in eight phases: open shop, closed shop or batch processing, multiprogramming, time-sharing, concurrent programming, personal computing, distributed systems, and real-time and embedded systems. The operating system reached some level of maturity during the time-sharing era. Time-sharing systems help users to interact closely with computers.

This chapter presents an introduction to general-purpose operating systems. It explains why we need operating systems in computers, and what they really do for computer users. It explains in what ways operating systems help users they manoeuvre computers with relati

A bare computer co resources such as

CHAPTER

2 Hardware Platforms

Learning Objectives

After reading this chapter, you should be able to:

- Describe the main components of a typical hardware platform, namely the processor, the main memory, the I/O devices, and the bus.
- Explain the functions of hardware components.
- Describe the ways of interconnecting hardware components to form a full-fledged computer.

2.1 Introduction

We start this chapter with a basic question: what is a platform? A *platform* is an entity that provides an environment for carrying out certain intended tasks, often with relative ease. In the context of computing, the term platform usually refers to the composition of an operating system and the underlying computer hardware. However, these are two different things, as explained below. Here, by a platform we mean either a hardware platform or a software platform, but not both. The hardware platform is intended for creating and executing application programs. The software platform is intended to enable computer users perform these two activities more effectively, securely, and conveniently. Each platform defines its own interface by means of which users can accomplish the intended tasks.

≫ A platform is either a hardware equipment or a software package that is used as a base to build something else.

The hardware platform is the real computer that does useful work, and which sets up the foundation for the software platform by providing basic services. The software platform, in turn, provides a friendly environment in which users can conveniently do their work. Although this book is all about software platforms, knowledge about hardware is essential to be an operating system expert. We discuss hardware platforms in this chapter and software platforms in the next. A hardware platform consists of individual hardware

devices. We briefly discuss here the working principles of some essential hardware devices.

2.2 The Main Hardware Components

A bare computer comprises many individual hardware components (also called physical resources). Figure 2.1 presents the model of a typical computer. It consists of a few processors, a bank of main memory, and a number of input-output (I/O) devices. Examples of I/O devices include the keyboard, the mouse, the monitor, disks, printers, networks interface cards, etc. A hardware resource is a physical entity that performs a specific task such as storing, transmitting, or manipulating information. For example the processor manipulates information, the memory unit stores information, and a network card transmits information. These resources work together to achieve the purpose of a computer.

A bunch of communication wires, called *bus*, connects the numerous hardware components in a computer. The components work independently, and communicate among themselves over the wires for the smooth functioning of the computer. The memory device is connected to the bus through a memory controller, and each I/O device through an I/O controller. The main processors and I/O controllers work concurrently, and store/retrieve information to/from the main memory (via the memory controller). The processors communicate with the I/O controllers either directly over the bus or indirectly via the main memory.

Each hardware component plays a different role in a computer. The functions of these essential hardware components, namely the bus, the memory, the processor, and I/O devices, are discussed briefly in the following subsections. We present them in the order of increasing complexity.

2.2.1 Bus

For a computer to function properly, the hardware components must be able to communicate among themselves. The components are connected to one

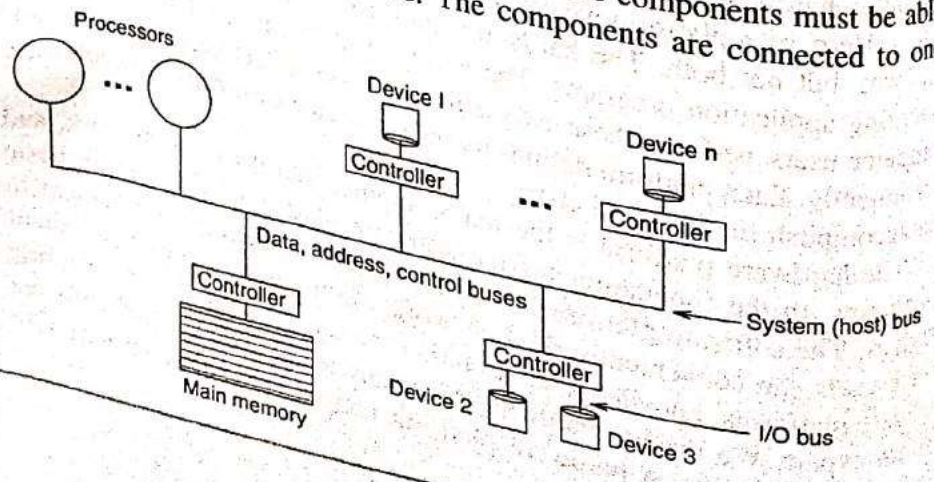

Figure 2.1: *The hardware components of a typical modern computer.*

another by electrically conducting wires. They communicate among themselves by changing electrical signals (voltage levels) on the connecting wires, where different signals represent different types of information. Different wires are used to transfer different types of information. Many wires, each carrying the same type of related information, are collectively called a *bus*.

Buses are primarily classified into three different categories based on their usage in the computer: data, address, and control buses. Sometimes, a single bus is used for multiple purposes.

1. A *data bus* comprises a group of wires, which transfers data between two hardware components connected in parallel. It holds the data being transferred between the two components. Data buses generally contain 8-, 16-, 32-, or 64-lines.
2. An *address bus* comprises a group of wires, which transmits an address in parallel. Each hardware component is identified by a distinct address (or a set of addresses), called its *bus address*. If a component A wants to exchange data with another component *B*, *A* enters the bus address of component *B* on the address bus to connect to *B*.
3. A *control bus* comprises a group of wires, which transfers control information to connected hardware components. The control bus is used to route timing- and control signals. It is used to exchange commands, directives, and status information among connected components.

These three kinds of busses are sometimes collectively referred to as the *system* or *host bus*. In contrast, the buses that connect I/O devices to I/O controllers are called *I/O buses* (see Fig. 2.1 on page 48).

Communications over a bus have to strictly follow a set of rules, collectively called a *bus protocol*. This protocol governs the behaviour of the bus. It defines a set of messages (electrical signals) that can be sent over the bus, and specifies the rules for bus requests, arbitration, connections, disconnections, data transfers, and releases. If more than one component tries to use the bus at the same time, the bus arbitration rule decides which of them will be permitted to use the bus. Once a component (referred to as the *initiator*) has been granted use of the bus, it connects the initiator to another component (referred to as the *target*) to establish a nexus for the communication. Then, the two components can exchange information over the bus wires for a limited length of time. When the data exchange is complete, the initiator (and sometimes the target) releases the bus, and it becomes free for use by another component.

Not all devices may be capable of driving the system bus to read and/or write the memory unit or device registers. A device that can drive the system bus to initiate a connection for communications (such as to read/write the main memory) is called a *bus master*; others are *bus slaves*. Devices, such as the processor and the DMA (discussed in Section 2.2.4 on page 60), have bus master capability. If more than one such device tries to utilize the bus at the

≫ A bus may be unidirectional or bidirectional. In the unidirectional bus, communication is always in one direction only. The bidirectional bus allows communication in both directions. When two-way communication is needed between two components, either a bidirectional bus or two unidirectional busses (one for each direction) are used.

≫ In memory-mapped I/O systems, the same system bus is used for both memory- and I/O devices. In separate (i.e., non memory-mapped) I/O systems, there are separate busses for them. The bus that connects the processor to the I/O devices is called the I/O bus. We try to avoid this usage of the term in this book. We instead call the bus that connects the I/O controller to the I/O devices the I/O bus.

≫ A protocol is a set of conventions and communication rules that two or more agents follow to achieve something cooperatively.

≫ The bus arbiter can be either a separate hardware resource or a part of the processor.

same time, then arbitration among these competing devices is required. A hardware device designed for bus arbitration is called a *bus arbiter*.

2.2.2 Main Memory

The *main memory* is a hardware device (also called the memory chip) that stores information used by the processors and I/O controllers. It is connected to the host bus through a memory controller. It is a passive device, in the sense that it neither interprets the stored information nor modifies the information on its own. It only guarantees that whatever information stored in it can be repeatedly retrieved without any alteration.

The main memory is considered a linear array of elementary placeholders of primitive information (data or instruction). Each elementary placeholder is called a *memory location*, more popularly a *memory cell*, or simply a *cell*. Information stored in a cell is called its *content* or *value*. Each cell holds values from a predetermined finite domain of values. As mentioned in Section 1.2.3 on page 8, memory locations are primitive storage components of the main memory, and each location holds an instruction or an operand. In practice, all cells in a memory device are of the same size, and they store a byte (8-bits) or word (fixed size in multiple bytes) of information. The cell size depends on the memory architecture. Figure 2.2 shows the structure of a typical memory cell of one byte.

≫ Each cell is a finite sequence of atomic components called flip-flops. Flip-flops can store a binary bit value of either 0 or 1, and, therefore, are popularly called bits. These bit values also can be interpreted as boolean values of false or true, respectively.

Each memory cell has a distinct address called its *memory address*, or more popularly its *physical address*. A physical address refers to all the bits of the cell collectively as a single indivisible unit. The address does not refer to the individual bits. The processor and the I/O devices access the content of a cell by passing its physical address to the memory device via the memory controller. The memory device has an address decoder embedded in it, as shown in Fig. 2.3. The decoder is connected to the host address bus. When activated by the control system, the decoder, in turn, activates a single cell that is identified by the address value on the host address bus. The addresses of all the memory cells constitute the *memory address space* of the computer system.

To access its content, each cell implements two primitive operations, namely *fetch* and *store*. An execution of the fetch operation on a cell returns

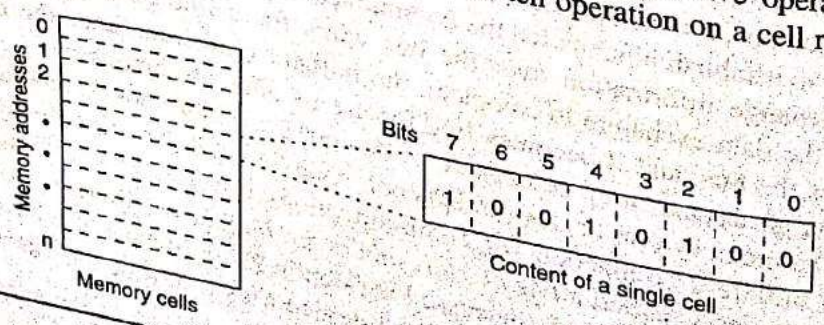

Figure 2.2: *The structure of a memory cell.*

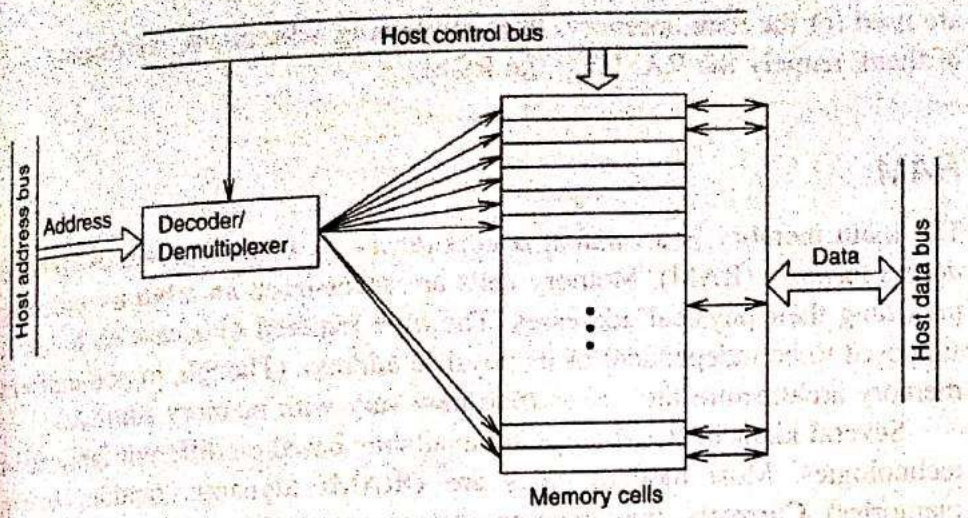

Figure 2.3: *A memory address decoder.*

the latest value written in the cell without altering the content. An execution of the store operation on a cell overwrites its previous content, replacing it with a new value. The memory controller, in turn, implements two operations, namely *read* and *write*. (Incidentally, read and write operations are often called fetch and store, respectively.) The controller translates each read (respectively, write) request to a fetch (respectively, store) operation on the cell addressed. An execution of the read operation takes a physical address, and returns a value from the cell identified by that address. An execution of the write operation takes a physical address and a value, and overwrites the current content of the corresponding cell; the operation finally returns an acknowledgment. Execution of a memory operation is called a memory *access* or memory *reference* (either a *read* or a *write*). Each memory reference consists of a physical address and a command to fetch the existing value from or to store a new value in the corresponding cell.

It is quite possible that more than one user of the memory (main processor or I/O controller) executes read and write operations on the same memory cell at the same real time. The memory controller synchronizes these simultaneous accesses by serializing them in an arbitrary order. Synchronization is needed to ensure the orderly execution of read- and write operations, otherwise chaos would ensue in the system. Most memory controllers execute one memory access request to the main memory at a time, and delay servicing the other concurrent access requests until the first one is complete. An operating system expects this minimal synchronization from the memory controller. Thus, unless specified otherwise, we will assume that read and write executions on each memory cell are *atomic*, that is, indivisible. That is, memory operation executions take place sequentially in a linear or total order. As noted previously, a read operation returns the current value from a memory cell, and a write operation updates the value of a memory cell.

The main memory is the only external (residing outside the processor chip) online storage medium from which the processors can directly reference individual memory cells. There are various types of hardware resources that

≫ There are non-atomic memory systems, where the memory controller may allow concurrent executions of some operations to reduce the response time. The memory hardware sees stream(s) of read and write requests, and executes these requests one by one or concurrently depending on the memory semantics.

≫ The term memory location henceforth will refer to a location in either a RAM or a ROM.

are used for the main memory. In the following subsections, we discuss two of them, namely the RAM and the ROM.

RAM

The main memory is a *random access device*. It is also known as *random access memory* (RAM). Memory cells are accessed in an arbitrary order by providing their physical addresses. The time required to access each cell is supposed to be independent of its physical address. (Though, in non-uniform memory architecture the access time may vary with memory address.)

Several kinds of RAM chips are available, based on different fabrication technologies. Most modern chips are DRAMs (dynamic random access memories). Currently, they have an average access time of about 50–70 ns.

ROM

≫ In personal computers, the ROM stores BIOS (basic I/O system) of about 64–128 KB. Embedded devices may have more ROM space where the entire operating system and some static constant data reside.

The RAM is a volatile storage device, and it loses the stored content across power disconnections. The content is random, unpredictable, and becomes suspect upon power reconnection. Processors need to have access to some trustworthy content upon power reconnection. This trustworthy content is stored in the read-only memory (ROM). It does not lose the content across power disconnections. It is generally used to hold a tiny part of the boot-strapping program. A ROM has all the properties of a RAM, except that processors or I/O controllers cannot overwrite its content.

2.2.3 Processor

≫ In practice, the term processor is loosely used to refer to everything, from the CPU to the entire motherboard. In Intel families, the interrupt controller initially was a separate chip, but now is a part of the chipset on modern ×86 motherboards. However, for PCI-based systems, it is an external component.

The processor is the most vital hardware resource in a computer. Its main task is to execute machine instructions. By so doing, it manipulates data stored in the main memory and controls the operations of the peripheral devices. A block diagram of a typical hypothetical processor with various interface lines is shown in Fig. 2.4(a), and internal components in Fig. 2.4(b). The hypothetical processor consists of a CPU, a memory management unit (MMU), an instruction/data cache, a cache management unit (CMU), an interrupt controller, an internal timer device, and various protection units. These components are discussed briefly in the following subsections.

Central Processing Unit

The CPU is the most important component in a processor. It is the brain of any computer, and it performs the tasks in the system by executing machine instructions.[1] The task (of executing instructions) is divided between a

[1] We assume that the CPU executes instructions sequentially, that is one after another. In this book, we do not address the issue of parallel executions of instructions.

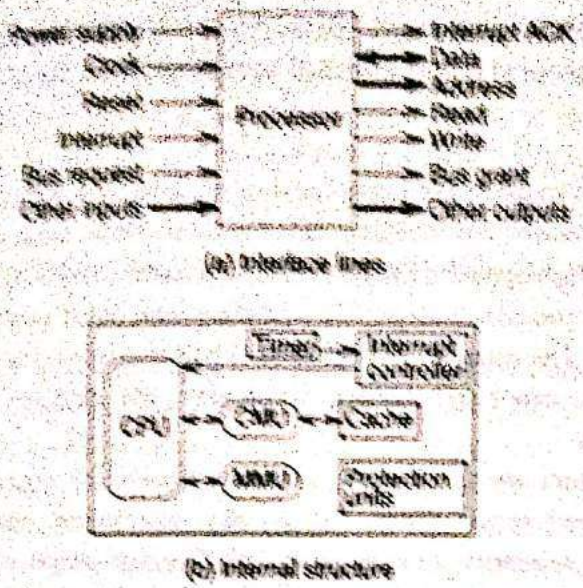

Figure 2.4: *Processor interface and processor composition.*

control unit that directs execution flow, and one or more execution units that execute the instructions. Figure 2.5 presents the model of a typical CPU organization. The CPU has a few registers that are used to store data and instructions; it has arithmetic, logic, and control units to execute instructions. A fixed frequency input clock drives the CPU hardware circuit. This clock, known as the *system clock*, generates and sends clock pulses to the CPU (and other components of the processor) at fixed intervals. The speed of a processor is also described in terms of the pulse rate of the system clock. At each pulse of the clock, the CPU performs some useful work.

The processor architecture implements a set of machine instructions, and supports a set of primitive data types. The CPU cyclically fetches instructions from the main memory, decodes them, and executes them utilizing appropriate execution units. The execution of an instruction may involve accessing additional data called *operand(s)*. Operands may be located in the CPU registers or outside it. The CPU automatically reads (writes) external

≫ Most authors of books on operating system do not differentiate between the CPU and the processor. We follow the same convention, and in this book, unless stated otherwise, we often use the terms processor and CPU interchangeably to mean the same component.

≫ Any instruction to be executed must be present in the main memory before the CPU can start executing it.

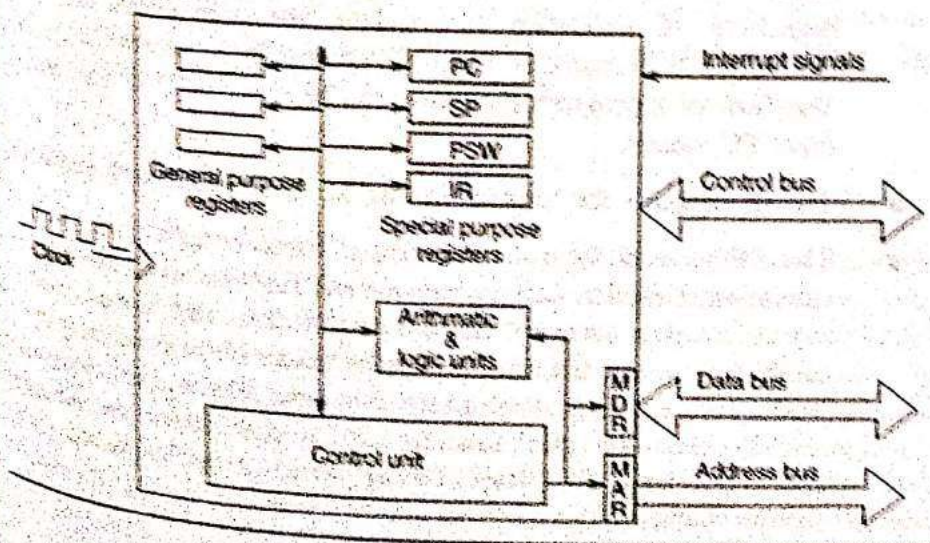

Figure 2.5: *The organization of a typical CPU.*

operands from (into) the main memory or I/O devices. In the following subsections we present a few internal components of the CPU.

CPU REGISTER Registers are composed of a finite number of flip-flops, and they are placeholders for holding information (both data and instructions). They are very high-speed storage units, and each is accessible in one clock cycle. They are volatile storage units as they lose their contents when power is disconnected. The number of registers implemented in the CPU varies from one type of processor to another. Typically, the number is in the neighbourhood of ten.

Some registers are categorized as *general-purpose registers*. Application programs can reference them. Frequently referenced data are stored in general-purpose registers for speedy access. General-purpose registers include accumulators, data registers, base registers, index registers, etc. Apart from general-purpose registers, there is the class of *special-purpose registers*. They have a very special use in the CPU in that they have specific control functions. Some of them are privileged resources, and are not available to the application programs. Operating system programs that are executed when the processor is in the kernel operating mode can, however, reference some of these privileged special-purpose registers. There are some special-purpose registers that the operating system may read but not modify; the CPU and other processor internal units use these registers for controlling the system. Most CPUs have the following special-purpose registers: memory data register (MDR), memory address register (MAR), program counter (PC), stack pointer (SP), instruction register (IR), processor status word (PSW) register, process identification register (PIR), and memory management registers. The roles of these special-purpose registers are briefly discussed here.

≫ The PC controls the flow of execution. This flow can be redirected by simply changing the content of the PC.

- The PC is the most important register. It points the CPU to the instruction that it is to execute next. The PC is automatically incremented by one after execution of each instruction. When explicit branching of execution is required, the appropriate branching instruction is executed to make the change in the content of the PC. The flow of a program execution can be observed from the consecutive PC values.
- The IR contains the instruction that the CPU is currently executing.
- The SP is used to track the nested function calls in a program execution. Function call parameters and function local variables are stored in a data structure called the *stack*. The stack is the means of easily saving and restoring temporary values in the main memory. It works on a last-in first-out basis. Processor architectures implement special instructions that allow the CPU to push values onto the stack and to pop them off it later. The SP always points to the top of the current stack.

- The PSW contains information on the current state of the CPU. It may also store additional information on the current state of the processor. For example, most processors have at least two different modes of operation: the kernel and the user. One or two bits in the PSW indicate the current processor-operating mode.
- The PIR stores the information to identify the process currently running.
- The MAR and the MDR are used to access information from the main memory and the I/O devices. They hold the addresses and the data, respectively, during memory operations.

CONTROL UNIT The control unit is in control of all CPU activities. It executes instructions sequentially, as the CPU fetches them. After execution of each instruction, it tests for the presence of interrupt signals. If some interrupt signal is present, the CPU commences executing instructions from a different memory location. The control unit coordinates all these CPU activities.

ARITHMETIC AND LOGIC UNIT The arithmetic and logic unit (ALU) handles the final execution of the decoded instructions. It carries out all the arithmetic and logical operations required for their execution.

Memory Management Unit

It was mentioned in Section 1.5.2 on page 20 that each process operates in its own private address space or the kernel space. When the CPU executes a process, the references to the main memory it produces are (logical) addresses in the respective address space. In most modern processors, these logical addresses differ from physical memory addresses. The memory management unit (MMU) translates each logical address the CPU generates, to its corresponding physical memory address. This runtime logical to physical address translation is transparent to the CPU. The MMU uses some CPU registers and physical memory contents to carry out the task of address translation. The MMU will be dealt with in detail in Chapter 8.

Hardware Cache

Registers are internal to the CPU, and they are the fastest storage units accessible to (various units in) the CPU. The latter can access one register in one clock-cycle. In contrast, to access a memory location, the CPU needs many (of the order of 10) clock-cycles as the access is carried out by following the system bus protocol. In most cases when the CPU makes a reference to the memory, it has to wait until the reference is complete. This leads to a substantial degradation of CPU performance as the waiting effectively slows down the speed of instruction execution. This waiting on memory references by the CPU, is called *stalling*, and is deemed

necessary for those instruction executions that have to access operands from the main memory. Stalling is undesirable as the CPU performs at least one memory reference for each instruction fetch (and more than one memory reference, if the instruction execution references operands in the main memory).

≫ A cache is a small "redundant" storage device, in the sense that it creates and holds copies of memory cells strictly temporarily.

To compensate for the performance penalty due to stalling of the CPU, a small amount of fast memory, called *hardware cache* or *cache memory*, is incorporated between the CPU and the main memory. It resides in the processor chip, outside the CPU. The processor architecture may use the same cache unit for both instructions and data, or provide them separate units. Caches are built using static RAM technology. They are costlier, but their typical access time is about 20 ns compared to the DRAM used as the main memory, which has a typical access time of 70 ns. Thus, caches can usually be referenced in 20 per cent to 25 per cent of time required to access the main memory. Consequently, caches substantially speed up the execution of instructions by the CPU.

≫ In some systems, caches may reside outside the processor as a separate chip on the system board. There, it "logically" resides between the CPU and the main memory.

Although caches reside outside the CPU, they are internal to the processor, and act as high-speed temporary storage devices, holding parts of the main memory. Caches help to reduce the number of references to the physical memory, and thereby, the workload on the memory. A cache unit is installed to store frequently referenced instructions and data. The processor thus can satisfy most of the CPU's memory access requests from the cache without having to access the main memory. Thereby, the stalling of the CPU considerably reduces.

Note that cache memory is not programmable, and is managed solely by a processor hardware circuit called CMU. As far as application programs (even operating system programs) are concerned, the cache is transparent to them. However, some processor architectures define instructions to enable, disable, and flush cache memory. When caching is disabled, the processor directly accesses data/instructions from the main memory. When caching is enabled, the CMU intercepts every physical address generated by the processor MMU to check if the corresponding memory cell copy is present in the cache. If a copy is present, a *cache hit* has occurred and the CMU returns the copy to the CPU on read access or overwrites the copy on write access. If the copy is absent, a *cache miss* has occurred and a copy is brought from the main memory. Achieving a very high cache hit rate speeds up the program execution substantially.

≫ There are processors in which cache operates on logical addresses that are generated by the CPU.

The cache storage space is partitioned into a number of fixed-size *cache lines*. A typical cache line holds values from many consecutive memory locations, collectively called a *memory line* (see Fig. 2.6). For example, in Intel ×86 processors, a cache line holds a few dozen bytes. When caching is enabled, the processor reads (writes) data/instructions from (to) the main memory one-memory-line at a time in a single memory transaction burst. Note that the CPU always produces references to individual memory cells. The processor translates the CPU requests to references to memory lines when the processor needs to access the main memory.

≫ Although cache memory is not programmable by software, we can structure programs in a way that they can exploit the cache in the best possible way. In short, programming becomes cache-conscious, especially for high performance applications.

Unlike the main memory, a cache is a "content-addressable" storage unit. A cache is never referenced by the address of cache lines. A cache line holds a pair consisting of a "key" and a corresponding value. The value here is a

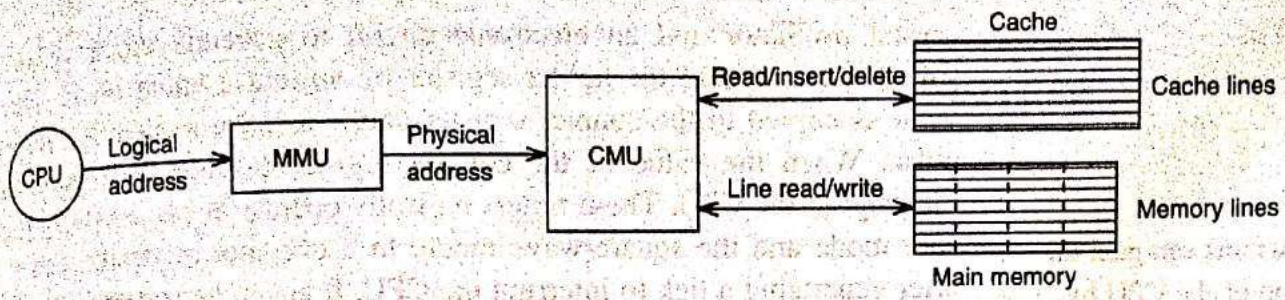

Figure 2.6: *Cache access.*

memory line, and the key is an entity that uniquely distinguishes one memory line from another. For example, a fixed prefix of memory addresses is used as the key. The key is used to access the cache. A search key value is compared against the keys in all cache lines. If a cache line contains the search key value, the memory line is obtained from the cache. Otherwise, the cache signals a negative acknowledgment, indicating a cache miss.

As stated earlier, in the case of a cache miss, the processor fetches the corresponding memory line and inserts it into the cache. It may so happen that there is no suitable free space available in the cache for the new line. In that event, the CMU replaces some cache line for the new memory line. We will study various cache replacement schemes in Chapter 14.

Timer Devices

Knowing the current time is important for many purposes. Modern processors have *internal real-time clocks* (RTC) to determine the current time. The RTC has its own battery so that it continues to run even when the computer is not powered. Another important timing device is a *programmable interrupt timer* (PIT); its architectural model is shown in Fig. 2.7. It has a clock made up of a

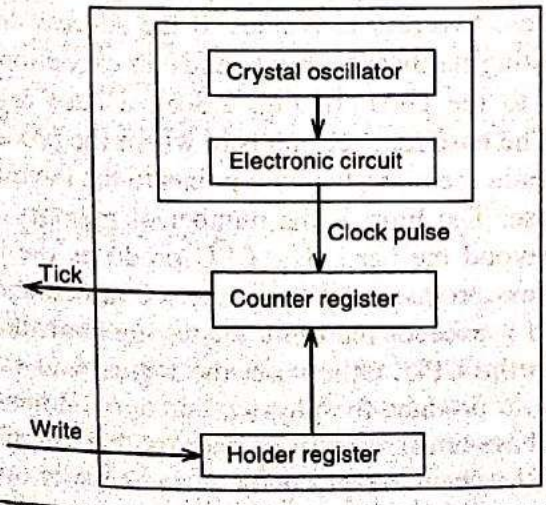

Figure 2.7: *The architecture of a programmable interrupt timer.*

crystal oscillator and an electronic circuit to generate clock pulses. The software can access the holder register by writing a value on it. Then this value is copied to the counter register and it is decremented at every clock pulse. When the value of the counter register reaches 0, it generates an interrupt to the CPU. These timers typically operate in two modes—the one-shot mode and the square-wave mode. In a one-shot mode the timer stops after generating a tick to interrupt the CPU. It has to be restarted to generate further interrupts. The square-wave mode timer repeats the process and generates periodic interrupts until the holder value changes.

≫ Programs can gain the attention of the CPU by setting a suitable timer value.

Protection Hardware

In Section 1.5.2 on page 20, we talked about the need to protect the kernel space and individual process address spaces. This need for protection is partly ensured by the operating system software and partly by the native hardware. Many kinds of protection circuits are present in modern processors. We argued, in Section 1.6 on page 23, of the necessity to have at least two processor operating modes to protect privileged resources from application processes. In this subsection, we show how one application process can be restricted to its private address space. Here, we discuss two simple hardware protection schemes. Others will be discussed in Chapter 8.

The processor MMU implements a couple of special-purpose registers. These registers are also called *bound registers*, or *base* and *limit registers*. Figure 2.8 presents two typical address space protection schemes. In the two schemes, every memory reference by the CPU is checked by an extra piece of hardware circuit before the reference is sent to the memory unit. The hardware controls precisely the part of the main memory that is to be made accessible to the CPU. In Fig. 2.8(a), the processor uses two bound registers: the lower bound register and the upper bound register. The operating system sets the contents of these registers to appropriate values whenever it schedules a process for execution. These two registers delimit the range of the physical memory addresses that a running process is allowed to access. The CPU produces physical addresses. Every memory address that the process produces is tested against these two register values. If the address does not fall within the bounds, the testing hardware circuit sends an exception signal (a kind of internal interrupt) to the CPU. In Fig. 2.8(b), a base register and a limit register are used. The base register points to where the process's program and data reside in the main memory; the base points to the lowest memory address the process may use. The limit is the number of memory locations that the process can use beyond the base. The CPU produces the logical addresses. Every logical address produced by the process is tested against the value of the limit register. If it exceeds the limit, the testing hardware circuit sends an exception signal to the CPU. Otherwise, the logical address is added to the base register value to produce the physical memory address.

The bound (or base-limit) registers are inaccessible to application programs, that is, to the CPU, when the processor is in the user-operating mode. If

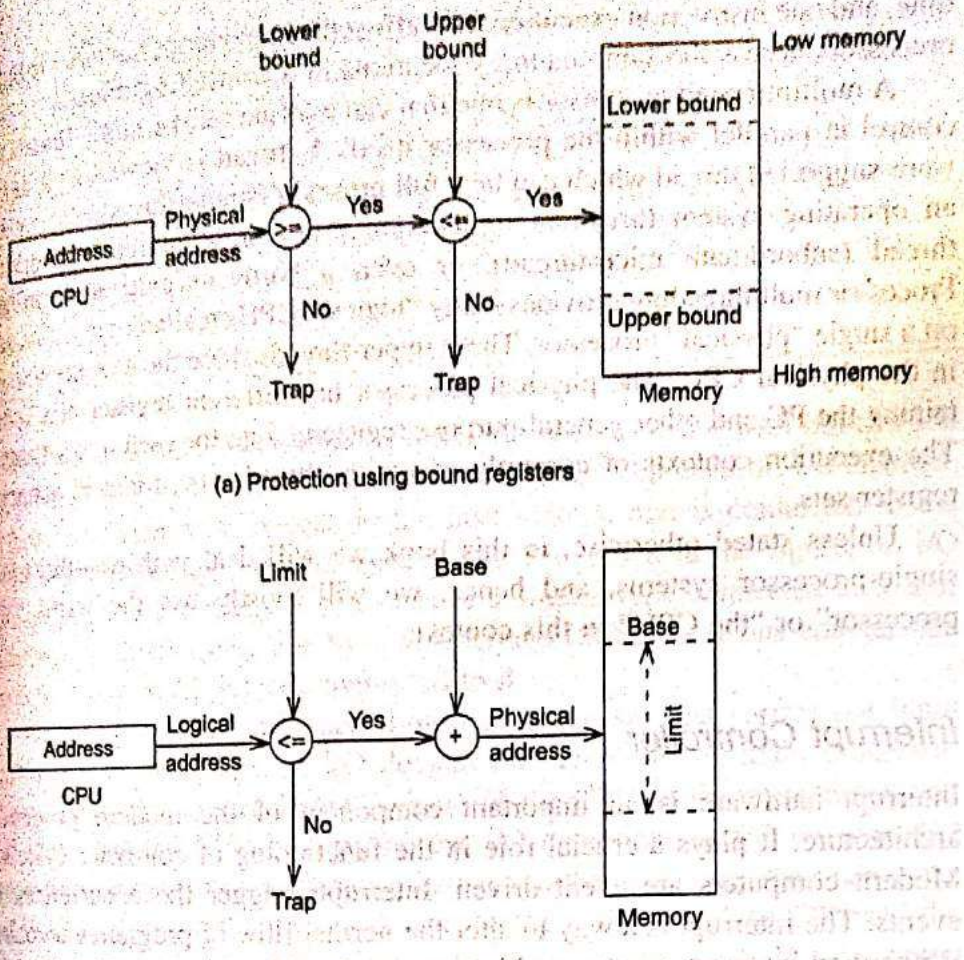

Figure 2.8: *Address space protection schemes.*

application programs can load them, then there would be no memory protection. They are treated as privileged resources, and can only be accessed in the kernel mode. This way the operating system ensures that no application process can make references to entities that reside outside its private address space.

In summary, protection hardware, in general, is an integral part of the MMU. Every reference to the main memory by the CPU is checked by the hardware to verify whether the reference violates any protection criteria. The check is performed before the actual memory is accessed. If the check fails, a protection violation exception is generated by the hardware.

Multiprocessor vs. Multithreaded Processor

A computer system may have more than one on-board processor (as shown in Fig. 2.1 on page 48). Such a computer system is called a *multiprocessor* or *tightly coupled system*. In a multiprocessor system, all processors share the system bus, the system clock, and the main memory, and may share peripheral devices. The CPUs in the processors operate concurrently, and execute different instructions simultaneously. The instruction executions overlap real

≫ Multiprocessor systems have been developed to achieve higher throughput using the same non-processor hardware resources. The increase in throughput is however not linear, as there are substantial overheads involved in tracking and synchronizing activities of all the processors and their contentions for shared resources.

≫ Intel was the first to implement hyper-threading in its Xeon processor, and it later ported hyper-threads to Pentium 4. Xeon forms two logical CPUs.

time, and one instruction execution may affect the behaviour of another if both processors access the same memory locations in a conflicting manner.

A multithreaded processor is one that can execute two or more threads of control in parallel within the processor itself. A thread is viewed as a hardware-supported thread which can be a full program (single-threaded process), an operating system thread (a lightweight process), a compiler-generated thread (subordinate microthread), or even a hardware-generated thread. Processor multithreading provides many "logical" CPUs (called *hyper-threads*) on a single "physical" processor. These hyper-threads share the execution units in the physical CPU. The physical processor has different register sets (containing the PC and other general-purpose registers), one for each hyper-thread. The execution contexts of currently executing threads are stored in separate register-sets.

Unless stated otherwise, in this book we will deal with non-threaded, single-processor systems, and hence, we will mostly use the terms "the processor" or "the CPU" in this context.

Interrupt Controller

Interrupt hardware is an important component of the modern processor architecture. It plays a crucial role in the functioning of computer systems. Modern computers are event driven. Interrupts trigger the occurrences of events. The interrupt is a way to alter the normal flow of program execution. Without an interrupt system, multiprocessing would not be possible.

≫ An interrupt is the only mechanism by which the CPU takes a note of the occurrences of various events. In the absence of interrupts, it continues the current execution flow.

An *interrupt* is a signal that peripheral devices (and some processor internal components such as a timer) raise to draw the attention of the CPU to these devices. As mentioned previously, the CPU examines the presence of interrupt signals on its interrupt lines after the completion of every instruction execution. If interrupt signals are present, the CPU handles the interrupts before executing the next instruction from the current execution flow. At each interruption, the CPU suspends its current program execution, saves the address of the interrupted instruction and processor registers, and starts executing another program called the interrupt service routine. On termination of this service routine, the CPU restarts and resumes the original program from the interrupted instruction.

Figure 2.9 presents the model of a typical interrupt circuit. A number of interrupt request (IRQ) lines are connected to the CPU through an interrupt multiplexer. The multiplexer listens to all the IRQ lines, and if an interrupt signal is present on some input line, it in turn raises an interrupt signal to the CPU. Once it raises an interrupt signal, it waits until it receives an acknowledgment from the CPU. Upon receiving the acknowledgement, it withdraws the interrupt signal.

≫ In modern systems, interrupt multiplexers are external to the processor. They are popularly called *programmable interrupt controllers* (PICs). We study more about PICs in Section 12.5.

2.2.4 I/O Devices

Figure 2.10 presents a schematic representation of how I/O devices are connected to a computer system. Each I/O device is connected to the host

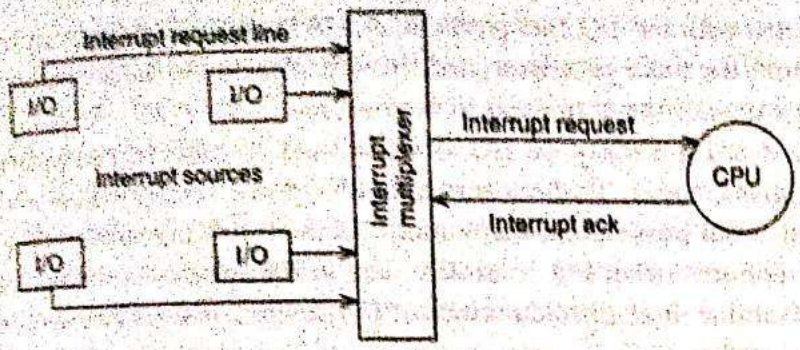

Figure 2.9: *A typical interrupt circuit.*

system through an I/O controller. (Some I/O devices come with built-in I/O controllers.) An I/O controller is also known as an I/O processor, I/O channel, or host bus adapter. It resides in the host system, and is connected to the processor through the host- or system bus. As shown in the figure, an I/O controller can have many I/O devices of a similar type connected to it. For example, a Small Computer System Interface (SCSI) host bus adapter can have up to 31 SCSI devices connected to it.

An I/O controller is a special-purpose processor that carries out input and/or output operations in I/O devices on behalf of the main processor. These controllers are intelligent helpers of the CPU in the main processor, and they help the CPU in carrying out I/O operations in I/O devices. I/O controllers are mostly autonomous devices, in the sense that they can carry out I/O operations without interventions by the main CPU. The CPU and the controllers can carry out different tasks concurrently. Each I/O operation involves transferring data between the host system and an I/O device; the corresponding I/O controller does the transfer, though the main CPU actually decides what data is to be transferred to (or from) which I/O device.

≫ An SCSI bus can have a total of 32 SCSI devices (both host bus adapters and device controllers).

The electronics that directly controls an I/O device is called a *device drive* (see Fig. 2.10). A device controller is built into the device drive. The device controller operates the device drive to carry out operations on the device. The device controller usually has a built-in data cache. Data transfers at the device drive happen between the cache and the device. The device controller is connected to an I/O controller by a bus called *I/O bus*. Data transfers between the device controller's cache and the I/O controller take place via the I/O bus. The device controller and the I/O controller communicate in

≫ Wireless devices are not physically connected to the host system by wires. They communicate over the air.

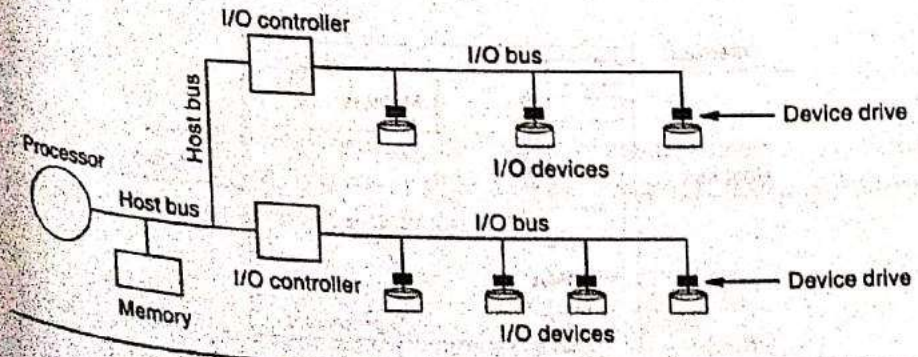

Figure 2.10: *The connection of I/O devices to the host system.*

accordance with the I/O bus protocol. An I/O controller accepts an I/O command from the main processor, and, in turn, directs the device controller that actually executes the command in microsteps.

There are a variety of I/O devices used in modern computers. They include disks, tapes, flashes, serial/parallel ports, network interface cards, universal serial busses, and many more. In the following subsection we look at some features of an I/O controller, and in the subsequent subsections, we briefly examine three different kinds of I/O devices: disks, tapes, and network interface cards.

≫ *It must be emphasised that there is a difference between a device drive and a device controller, and that they both are different from the I/O controller. However, in this book, to avoid confusion, we will mostly refer to these as device drive or simply device instead of device controller.*

I/O Controller

Each I/O controller has a set of registers, and may have an on-board memory chip (see Fig. 2.11). The on-board memory is used as a temporary storage, popularly called *buffer*, to hold the contents of the devices while these contents are in-transit between the host system and the I/O devices. The controller is solely responsible for moving data between the on-board buffer and the devices it controls.

The main CPU cannot access data directly from the buffer of any I/O controller, but through the interface resisters of the controller. There may be DMA (direct memory access) devices on the controller board. (DMA devices are usually fitted to high-speed I/O controllers.) They help the I/O controllers in transfer of data between the on-board buffers and the main memory without involving the CPU. The CPU informs an I/O controller about the base physical memory address, data transfer length, and the direction of transfer. A DMA device carries out the actual transfer of data between the buffer and the main memory. Alternatively, there can be separate DMA devices installed in the host system. If an I/O controller does not have an on-board DMA device, the main CPU engages a host DMA device to transfer data between the I/O controllers and the main memory without involving itself in the data transfer.

≫ *Like sea ports, I/O ports are places for shipping commodities. Only here, the commodities are commands, data, or messages.*

Some I/O controller registers are called *device registers* or *operating registers* or *I/O ports*. The CPU communicates with an I/O controller via the I/O ports. The ports are points of information exchange (such as commands, data, status, messages) between the CPU and the controller. The ports of all I/O controllers in the system constitute an address space called the *I/O address*

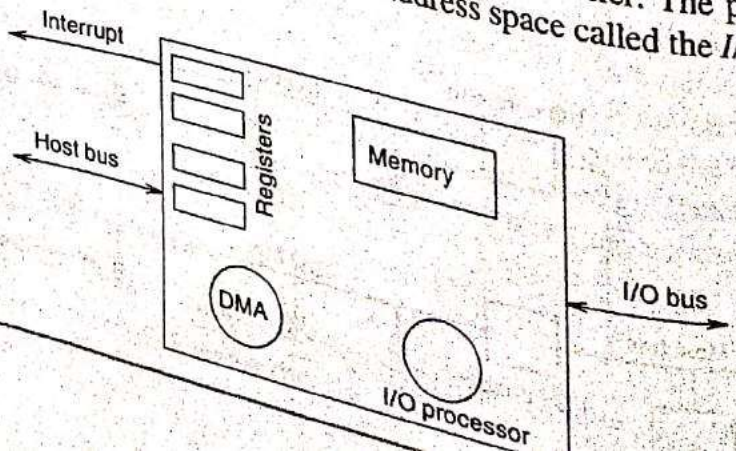

Figure 2.11: *A model of an I/O controller.*

space of the system. Each port has a unique address in the I/O address space. The processor architecture implements I/O instructions (such as IN, OUT) to access I/O ports. Executing these instructions enables the CPU to transfer data between its internal registers and the ports. The CPU controls the activities of an I/O controller by writing and reading the controller I/O ports.

The I/O address space is different from the memory address space. The CPU accesses entities in these two types of address spaces by executing different kinds of instructions. Many modern computer architectures, however, allow I/O controllers to be placed in the memory address space. In other words, they allow either the whole or parts of the I/O address space to be mapped into the memory address space. In such configurations, ranges of memory addresses are reserved for I/O controllers, and references to those memory addresses are automatically translated into references to I/O ports. Thus, execution of ordinary memory read/write instructions in these reserved memory addresses actually transfers data between the I/O ports and the internal registers of the CPU. This mechanism is called a *memory-mapped I/O*, and is a convenient method of accessing the I/O address space. Devices with faster response times such as video screen controllers, serial/parallel ports, and Ethernet cards are usually accessed through the memory-mapped I/O mechanism.

≫ In X386 architecture, there are 65, 535 possible I/O ports, and are accessed by a set of instructions, generically called *IN* and *OUT*.

Almost all modern I/O controllers are fitted with interrupt circuits. When a controller wants to draw the attention of the CPU, it triggers an interrupt to the CPU by raising signals on its outgoing interrupt request line that is connected to the CPU via an interrupt multiplexer (see Fig. 2.9 on page 61).

Disks

A disk is a very important device; it is used as an online persistent storage medium. It is also called a *secondary storage device*.[2] Both user applications and system programs normally reside in disks. (At the time of a program execution, the program is brought into the main memory from a disk by the operating system.) Disks are often used as both data sources and data destinations by many applications.

≫ Memory-mapped I/Os may appear a little deceptive to ordinary users. As stated in Section 2.2.2 on page 50, memory semantics says that a read on a memory location returns the latest value written there. Such is not normally the case for mapped I/O ports. You have been warned!

Logically, a disk is a linear array of disk-objects called *blocks*. A block is a sequence of bytes, and typically stores 1024, 2048, or 4096 bytes of data. Blocks are units of data transfer between the host system and the disk. The disk is a *direct addressed* device, and blocks can be accessed in arbitrary order. More about disk organization and management will be discussed in Section 10.5.

Tapes

A tape is another important storage device; it is usually used as an offline persistent storage device. It is also called a *tertiary storage device*. Compared to disks, tapes are inexpensive devices.

[2] By contrast, the main memory is called primary storage device.

Like a disk, a tape is a linear array of tape-blocks. Blocks are units of data transfer between the host system and the tape. Unlike disks, a tape is a *sequential access device*. In other words, the blocks from a tape can only be accessed in linear sequence. To read or write a particular block, the tape needs to be repositioned first by forwarding or rewinding and only then can the block be accessed. Consequently, tapes are unsuitable for online storage to store operational and frequently accessed data. They are mostly used to backup operational data periodically, and are a good choice for archival storage.[3]

Network Interface Cards

Network interface cards help a computer system to communicate with other computer systems. Unlike disks or tapes, they are not storage devices, and they transfer transient data to two or more computer systems. They are extensively used in distributed systems, where many computers are connected to form a network. Computers communicate among themselves by sending and receiving data through network cards. To send data to a remote system, a computer writes the remote system address (also called network address) and the data to the card. The card, in turn, forms packets from the data, and sends them to the remote system. A card in the remote system receives these packets, and assembles them to form the original data. We will examine one network interface card in Section 10.6.

2.3 Hardware Layout

In the previous section, we talked about some essential, commonly used hardware resources. In this section, we discuss how those resources are connected together by various busses. The peripheral component interconnect (PCI) defines one of the commonest bus architectures. The PCI is an approved standard that specifies how hardware components in a computer can be glued together in a structured fashion. The model of a typical PCI-based hardware interconnection is shown in Fig. 2.12. There are multiple processors in the system, and are connected to PCI bus 0. PCI-compatible peripheral devices are connected to PCI bus 1. Shown in the figure is a non-PCI bus containing a mouse and a keyboard. Special PCI devices, called *bridges*, connect one PCI bus to another PCI- or non-PCI bus. Bridges help in gluing the connected components together. The north bridge (connected to PCI bus 0) acts as the memory controller. There is an APIC (advanced programmable interrupt controller) device that handles interrupt request lines from devices, and delivers their interrupts to the processors.

> The PCI is an Intel standard. On PCI-based systems, no central DMA controller chip manages the PCI cards. A PCI card that does DMA is a bus master.

[3] We need to maintain archival data for long periods, but we do not expect to access such data often.

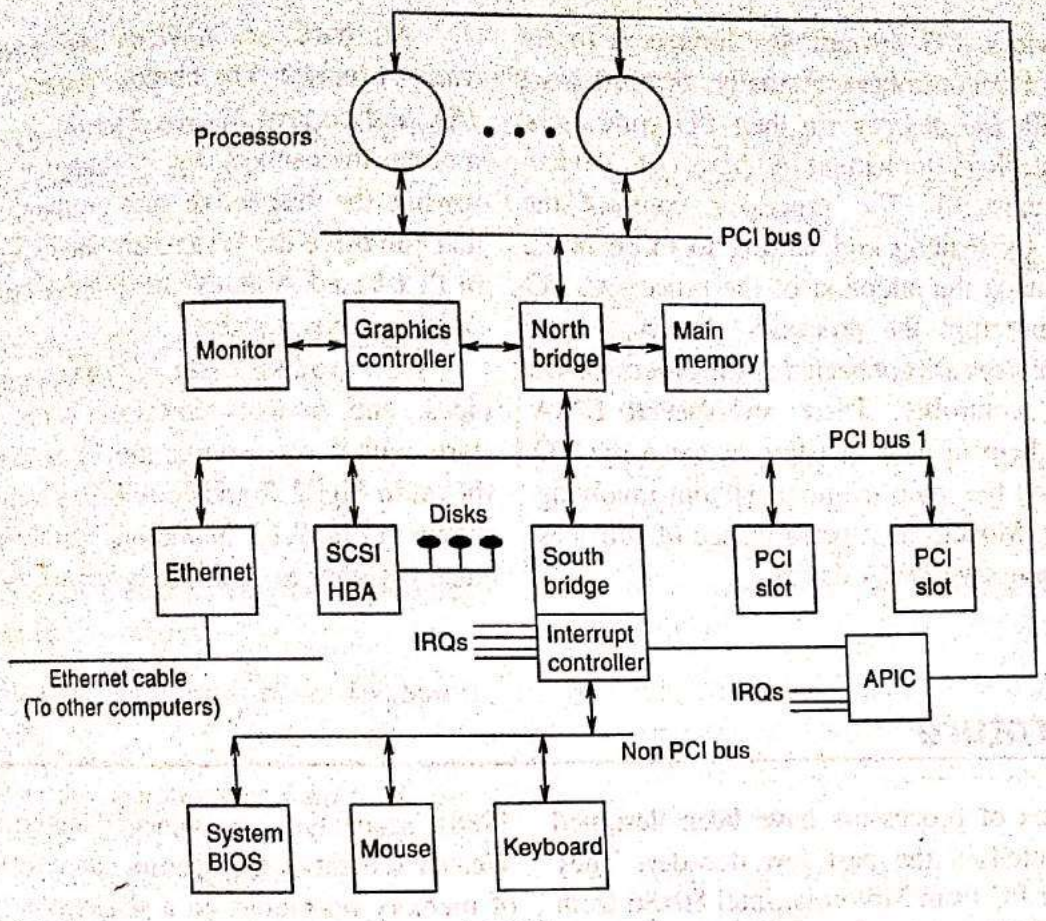

Figure 2.12: *Hardware interconnection through PCI busses.*

Summary

This chapter starts with the definition of a platform, and partitions a computer system into two platforms, namely hardware and software. The hardware resources comprise a hardware platform that carries out the real work. This chapter primarily deals with hardware platforms, and introduces the main hardware resources, namely the processor, the main memory, the I/O devices, and the bus. It briefly discusses the working principles of these resources.

The main memory stores the information required by the processor and I/O devices. There are two kinds of widely-used memory devices, namely the RAM and the ROM. The RAM is volatile, but the ROM is not. The ROM stores the bootstrap programs. They both implement read and write operations that the other devices use to access memory content. The main memory is connected to the system via a memory controller. The controller synchronizes concurrent memory operations.

The processor manipulates information by executing machine instructions. The chapter discusses some important components of the processor, namely the CPU, the memory-management unit, cache memory, the cache-management unit, and the timer device. The memory-management unit translates CPU-generated addresses to memory addresses. Protection circuits are embedded in the memory-management unit. They check for address space violations and raise address violation exceptions to draw special attention from the system. The cache memory is used to hold frequently referenced data and instructions to reduce the workload on the main memory and to improve speed of access. The cache-management unit manages the cache memory.

I/O devices are categorized in two: communication devices and storage devices. The processor interacts with the external world via communication devices and persistently stores information in various

CHAPTER

4 Processes and Threads

Learning Objectives

After reading this chapter, you should be able to:

- Describe the purpose of a process.
- Define the attributes of a process.
- Distinguish between a process and a program.
- Identify the relationship between a thread and a process.
- Describe process and thread management.

4.1 Introduction

We know from Chapter 2 that a hardware platform is composed of many different hardware resources. However, general users of a computer may have no interest in knowing about these hardware resources; their primary interest is in running utility programs of the system, the application programs they themselves develop, and the output their programs produce. When a user executes a program, the execution requires various resources to accomplish the task. For example, it needs the CPU and the main memory. We know that the operating system is the sole manager of all resources—hardware and software—in a computer system and is responsible for their allocation and deallocation. The key question is to whom does the operating system allocate resources? The short answer is, to program executions. We discuss the long answer at length below.

You may have noted that many users may run a single application program simultaneously. A single user may also run an application more than once simultaneously. In short, simultaneous executions of the same program are possible in a computer system. For example, many users may run a particular text editor program (say, vi) simultaneously. Even in such a case, the resource requirements for one editor execution differ from those for other executions of the same editor

[1] In this context, program means a standalone executable application program or utility.

program.[1] The operating system must have the means to distinguish one program execution from another; otherwise, the system may not be able to keep track of which program execution uses which resources. The system keeps track of program executions in the abstraction of processes, and allocates resources to those processes. A process represents precisely one program execution. Process is the most fundamental concept in the context of operating systems, and we first need to understand this concept to comprehend clearly the working principles of modern operating systems.

≫ In operating systems, processes are workhorses. They play a central role in the design, development, and implementation of most operating systems.

We frequently use the terminologies program execution, computation, and process interchangeably to mean the same action. Processes and program executions are in one-to-one correspondence with each other. The operating system starts an execution of a program (i.e., of a new computation) by creating a process. The process is allocated some main memory to hold its program and data. When a process is started, the operating system brings (i.e. loads) the required program and data in the allocated main memory, and builds up the initial "execution context" of the process. Note that each process starts with a predefined initial context. The operating system then allocates and deallocates various resources to and from the process as the execution of the program evolves. The process execution context stores this resource-allocation information, and the current state of the program execution and the other information related to it. In short, the operating system uses a process as a handle to manage a one-program execution.

≫ A process may be viewed as the eventual *avatar* of a solution to a problem. A solution is first abstracted as an algorithm in the software-design phase, transformed into an application program in the programming phase, and finally created as a process to solve the intended problem by execution of the corresponding program under the premise of an operating system. Every process has a program as its component and, at any given time, it is in a state of executing the program.

When a program is given to the operating system for execution, the system builds the other components of the program execution, provides necessary attributes, and eventually shapes them all into a whole structure called a process that can be conveniently, effectively, and securely managed in the operating system to accomplish the program's intended task. This brings us to a set of basic questions about processes: what exactly is a process? What are its main components and what exactly are the roles they play during the lifetime of the process? What are the states that a process may transit through and what kind of privilege modes can it adopt under various conditions during its lifetime? How are processes created, managed, and destroyed in a system?

This chapter aims to answer the above and related questions regarding processes. The next three chapters also deal with topics primarily related to processes and their management.

4.2 Process Abstraction

Performing a task using a computer essentially requires executing a suitable program. Therefore, an operating system, on receiving a specific task request, must have a suitable program available to execute the task. The entity which carries out the task described in a given program is called a *process*. (See Box 4.1 for the evolution of the process terminology.) Every process has a program as its execution component and acquires its behaviour mostly from the logic of the program. In short, program executions are abstracted as processes for the purpose

of their management by the operating system. Processes and program executions are in one-to-one correspondence with each other. A process encapsulates the program code and the program data. In addition, it has private data to be manipulated by the program, and has other information required for its management by the operating system. Depending on the conditions under which the process executes a program it may also require other information for the execution.

4.1 Evolution of Process

Historically, large software systems underwent many painful development and maintenance cycles. Many of them failed miserably. Then came the concept of the structured system. The concept of the process was one of the many structuring tools devised to master the complexity of large software systems. An operating system too is a large complex system, and the process concept has been in use as a structuring tool here as well, especially to handle its runtime complexity. Since its inception, the concept of the operating system has evolved constantly, and has undergone several revisions and advancements. The concept of the process evolved alongside the operating system. (The original terminology employed to describe it was "task", and later "job". The term "process" has superseded them.)

To understand the evolution of the concept of process, we start with the open-shop and closed-shop era. In that era, a computer system had only one CPU available, and the whole of the memory and the CPU were given over to one program until it completed the execution; no user interaction was allowed during the execution of that program. In such situation, no additional information was needed to execute the program, and therefore, in the open-shop and closed-shop (batch) systems, a process simply meant a "program or job in execution".

The situation changed in the era of multiprogramming, where more than one program was allowed to share the main memory, the CPU, and other system resources simultaneously. Apart from this, it became possible for many users to request the system to execute the same program simultaneously. Therefore, each program execution has certain fundamental requirements. These requirements are maintaining a set of ids (owner id, program id, etc.) for its identification, an exclusive and secure space for its own storage and of other related information (address space), a mechanism to record its current execution point (as it could lose the CPU any time during its execution), a stack (to hold temporary parameters, variables, return addresses), etc. The operating system specifically supplies all these, as additional information is not explicitly provided by the given program. This enables the system to effectively execute and manage a program execution even if there are concurrent executions of the same and other programs. Such additional information provided by the operating system and the given program together form a logical structure called the process of that program execution.

When a process does not have control of the CPU, it may be waiting for the CPU, for data from an I/O device, or for data from another process in the system. That is, in this context, a process can exist in various logical states during its lifetime. This state information also becomes the part of the process. Thus in the multiprogramming era, a process is to be considered a program execution, along with its associated management information, and exists in some state of action. There is no perfect universally accepted definition of a process. The modern usage of the term seems to have coined during the MULTICS days in 1960s.

The concept of a thread as an active entity in a process is the latest addition to this evolution. Threads are strands of program execution embedded in a process.

≫ A process represents an "executing" instance of a program. IA denotes an entity that carries out a task (solved by a program), requesting resources from the operating system to which it returns the acquired resources on completion of the task.

≫ A process is an active entity in the context of the operating system and contains a program as its execution component. When it has the control of the CPU to execute its program, a process uses other resources such as memory, file, I/O devices, etc., to accomplish the program's intended task. From the operational point of view, the operating system is primarily responsible for creating, managing, and deleting processes in the system.

≫ Modern day processes may be categorized into two types: single-threaded (traditional processes) and multithreaded. Resources are attached to processes, whereas execution attributes such as state, mode, context, etc., are attached to its threads. The execution of a program in a single-threaded process is sequential, whereas the execution of a program in a multithreaded process is concurrent.

The operating system executes a program using a process as a handle. The process uses various resources as the program execution proceeds. Thus, a process has two basic characteristics: a unit of resource ownership and a unit of CPU scheduling for execution. This characterization captures the essence of a process. However, the amount of information added to the process structure varies from system to system. For example, in a system with more than one CPU, the process has to maintain a record of which CPU it is using. In practical systems, a process has many other data structures for its efficient and effective management.

Although applications are typically divided into processes as logical functions during the design phase, many independent segments within its program may be coded to execute in parallel without affecting the overall goal of the process. Such a specially coded construct within a program forms an execution thread (or simply a thread) in a process when it is equipped with the information necessary for its independent execution. Thus, a process might contain one or more threads. Indeed, in many modern operating systems processes are multithreaded. Because independent activities can be abstracted as threads, they often improve both program structure and the overall performance of the application.

In summary, a process has many components such as a program, one or more threads, contexts of those thread(s) containing crucial information such as location of current instruction execution (i.e., the PC value), current stack, execution mode, state, etc., attributes for identifications (such as process id), references to resources (such as file id), and other management and accounting information. We discuss in the following sections some of these topics, such as main components of a process, overall structure of a process, and process management.

Dynamism (i.e., runtime activity) of a computer system is grossly abstracted in a collection of processes. Some processes execute only operating system programs, and are called *operating system processes* or *kernel processes*. The others are called *user processes* or *application processes*. User processes normally execute utilities and application programs, and from time to time execute operating system programs to obtain services from the system. Kernel processes neither execute utilities nor applications, nor do they interact with users. The operating system maintains a tabulated list of all living current processes recording one entry per process in the system, called the *process table*.

4.3 Programs vs. Processes

An executable program is the primary component of a process. A program is a well-defined composition of machine instructions and primitive data, which describes a step-by-step solution to a problem. Nevertheless, a program does nothing in the system until it is executed. The operating system manages one execution of the program by using its associated process as a handle.

A program has two logical components: code and data. In addition to program data, a process possesses private data required for its execution and system data required for its management. The code and data portions of a process may be scattered across various physical storage units such as CPU

registers, cache memory, main memory, and disk. It is the responsibility of the operating system to manage and protect such scattered information.

A program is a static entity that does nothing on its own; it does not change itself. A process is a dynamic entity that changes the state of the program execution it represents. Thus, a process is lot more than a program.

Many processes may concurrently execute a single program. For example, many users may run the same text editor simultaneously. As shown in Fig. 4.1, a separate process represents one execution of the same editor. In the figure, three processes are carrying out the three executions of the editor. Each process executes the same editor program, but uses different sets of data. At the other extreme, a single process may execute many programs, in which case the execution is sequential, that is execution is carried out one program after another.

≫ The operating system executes processes concurrently by multiplexing them to the CPU. In multiprocessor computers, many processes though can run simultaneously. We shall study CPU scheduling in the next chapter.

≫ In UNIX systems, a process executes an exec system call to execute a new application program. The exec call terminates the current program execution, and commences executing a new program in the same process with a new execution context.

4.4 Process Context and Process Descriptor

When a user runs an application program, the execution requires the CPU to execute a sequence of machine instructions. The program execution ends when the CPU executes a termination instruction. Although most users' interests are in the output of program executions, the operating system has to manage those executions. It needs to know what program the CPU executes, where in the main memory the program and its data reside, which instruction the CPU is executing at a given time, what resources the program execution has acquired, what the program execution does with the resources, etc. Such information pertaining to a program execution is said to represent a *state* of the program execution. The system needs the state information of all ongoing program executions, and that of the system resources to manage the program executions effectively, and run the computer system smoothly.

A process represents a collection of control information that models one program execution, that is, a computation. When the system executes a process, the value of the PC register in the CPU determines which instruction

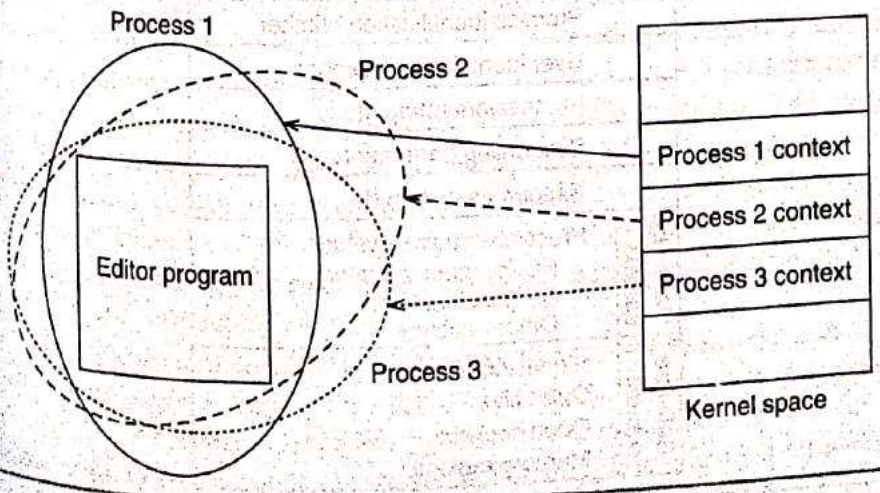

Figure 4.1: *The sharing of an application program by different processes.*

≫ Program and process are two "orthogonal" entities. A process is an active entity, whereas a program is a passive entity. A program is an executable entity that describes a transformation function to produce output data from input data. A process does actually carry out the transformation function by executing instructions from the program. The CPU executes the program on behalf of the process. For the program execution, the process is the logical environment, and the CPU is the physical environment.

≫ The context of a process typically includes values of the CPU registers (such as PC, SP, PSW, and other relevant registers), address space, process state, profiling and accounting information, and associated kernel data structures.

from the process's program the CPU will execute next. We often naively say that the process itself executes the program within the environment provided by the operating system. All control information pertaining to a process (again, one program execution) collectively defines an execution context called the *process context*. A process context represents the true state of a program execution, in which the operating system is interested.

The operating system creates, manages, and finally destroys processes. The system must know precisely what each process is doing, what resources it holds, what access rights it has on the allocated resources, what its scheduling priorities are, etc. That is, each process has to run within the context of its management information, and behave correctly only within that context. The context information is structured in the form of an object called *process descriptor* or *process control block* or *process object*. A separate process descriptor object represents each process. Note that a process does not need a descriptor for its execution. The operating system needs process descriptors for managing processes, and ensuring protection in the system. Since the descriptors are used for the protection of processes, they are privileged resources and application processes must not tamper with them.

Figure 4.2 shows the content of a hypothetical process descriptor. The content information is structured into a set of attributes. These are often called *process attributes*. The attributes noted in the figure are self-explanatory, and hence, all are not elaborately discussed here. A few attributes are explained in following sections.

A process context is divided into two parts: the hardware context and the software context. The hardware context includes the processor (general-purpose and special-purpose) registers, notable among them being the PC, SP, and PSW registers. The software context includes open files, memory regions, etc. In addition, a process descriptor stores various process-related identifiers, current state, priorities, resources acquired, pointers to other data structures, etc.

	Pointers to various data structures
	Process state
	Process identification number
	User identification number
	Process operating mode
	Scheduling parameters
	Resources acquired
Hardware context	Processor register values PC (program counter) SP (stack pointer) Other registers
Software context	Signal information Open files Open sockets Memory regions

Figure 4.2: The structure of a typical process descriptor.

Briefly, the key information required for process management is stored in a process descriptor and in other data structures that are linked to the descriptor. The data structures collectively contain everything the operating system needs to know about a process. From the point of view of the operating system, the descriptor is the "process". As mentioned above, process descriptors are privileged resources, and application processes must not be allowed to manipulate descriptor contents. Consequently, descriptors are stored in the kernel space, and are entirely managed by the operating system.

4.5 Process Address Space

Modern operating systems allow concurrent executions of multiple processes. One duty of the operating system is to protect each application process from interference by the others, and to protect itself from them, so that a process cannot alter programs or data belonging to another process or to the kernel.

The operating system partly solves this protection problem by implementing a concept called address space. Loosely, an *address space* consists of many different entities, and these entities have distinct individual addresses in the address space. By *address* is meant here a name or a number that uniquely identifies an entity in the address space. An entity in the address space can only be referenced by using its address. For example, a collection of programs (with their data) that are allowed to be executed (and accessed) by a process, forms an address space; it is often called the *logical* or *private address space* of the process, and individual addresses are called *logical addresses*. When a process executes a program, it references entities in its private address space using the logical addresses of these entities (see Fig. 1.8 on page 22). That is, it uses logical addresses to specify addresses of the instructions and operands. Analogously, the operating system programs and data constitute an address space that we call the *kernel space*. Private address spaces of user processes collectively form what we call the *user space*, as shown in Fig. 4.3.

Each application process is assigned a separate private address space, which contains the programs (and data) that the process can execute (and access). Each process runs in its private address space, that is, it executes instructions and accesses data from that particular address space. It must not be able to interact with other processes (or the kernel) except through a set of communication primitives supported by the operating system. The system also has to ensure that a process confines its activities to its own private address space. If a process tries to cross its address-space boundary (erroneously or intentionally), the system should intercept the address-space violation and take appropriate actions deemed necessary for smooth functioning of the computer system.

The execution of an application process alternates between the user space and the kernel space. The process primarily executes application programs in the user space. When it executes operating system programs, it is in the kernel space and is considered to have "effectively" become a kernel process.

≫ Processes may be compared to people in a society. Some characterizations of a process, initially, were derived by comparing it with the life of a living organism. Processes have unique identifiers to recognize one another, have living space, store data, acquire and relinquish resources, exist in different states (executing, waiting, sleeping, etc.), communicate with each other, compete for resources, possess the capacity to simultaneously perform many activities, and behave purposefully during their life time.

≫ A program is a static sequence of instructions, whereas a process is a container for a set of resources used when executing the instance of the program.

≫ An address is meaningful only in reference to an address space.

≫ To avoid large size process descriptors, most operating systems use many other process-related data structures that are linked to small-sized process descriptors proper.

≫ We raise a warning flag for readers here. The term "logical" is generic. As we shall see later, it may not always bind a process to a private address space.

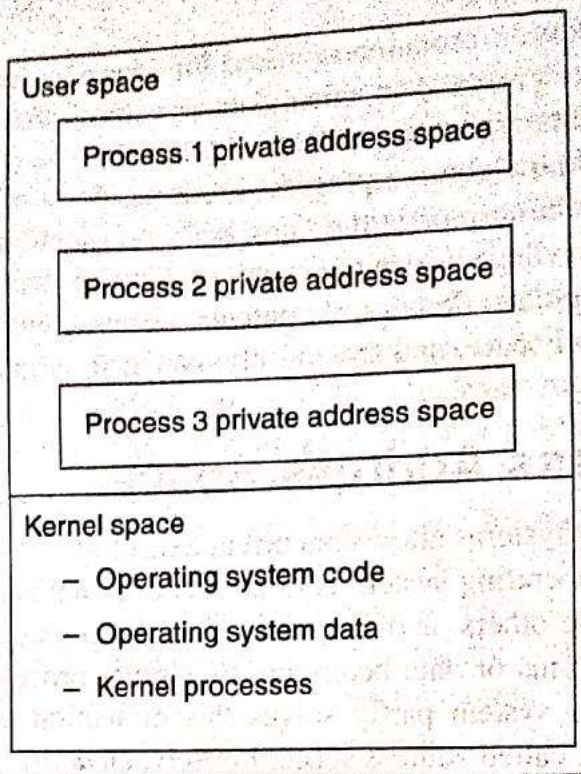

Figure 4.3: *An abstract view of a computer system at runtime.*

In contrast, the original kernel processes always execute in the kernel space, and never in the user space.

When the operating system creates a new process, it initializes the process address space as a part of the initial context building. When the process wants to execute a new program, the operating system reinitializes the process address space—in fact, it first destroys the old address space—and creates a new one for the process. However, the operating system may retain parts of the process descriptor information (e.g., process identification number) during the re-initialization, as done in UNIX systems.

In theory, an address space can be infinite, yet countable. However, in practice, because of the limited availability of resources, address spaces are restricted to finite sizes. These sizes depend on the underlying processor architecture, and are limited by the size of the PC register and other engineering constraints. In the following subsection, we discuss a typical composition of process address space.

4.5.1 Process Address Space Structure

The address space of a process contains programs and data from many sources. It includes an executable image of one application program and those of various libraries. These sources also supply program data, often called *static data*. In addition, a process will have different data, called *dynamic data*, when it executes the program. Thus, a process may expand and shrink its address space in its lifetime by way of allocating and releasing dynamic entities, respectively, to and from its address space. As static data belong to application programs and libraries, a process cannot release the data from its address space unless it also releases the corresponding programs.

≫ In UNIX systems, a process cannot add or release shared libraries to or from its private space, but can do so for dynamically loaded libraries.

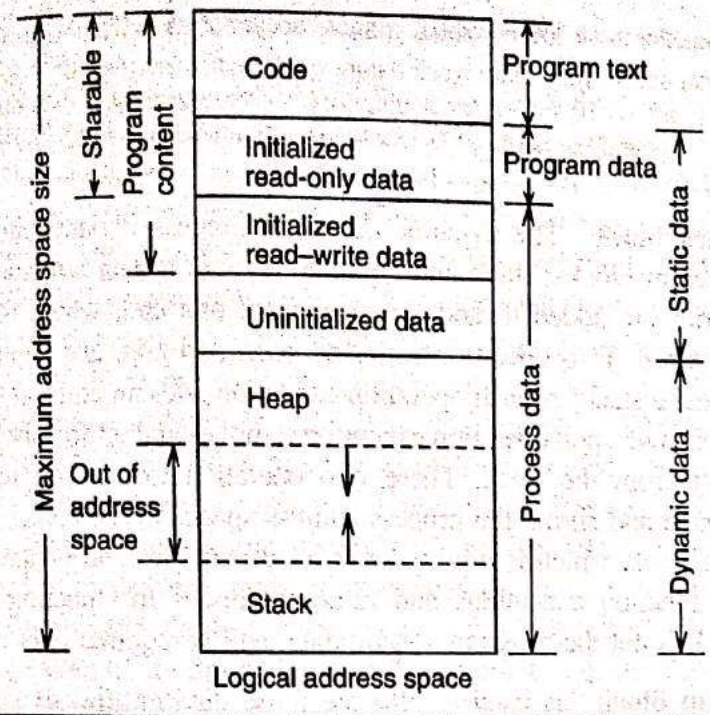

Figure 4.4: *Typical logical partitions of the process address space.*

An address space is typically partitioned into a number of blocks of variable sizes. In UNIX systems, a private address space is composed of four blocks (see Fig. 4.4): (1) code, (2) data, (3) stack, and (4) heap. The figure presents a logical picture; the code and data sections in practice are divided into many partitions, which may interleave in the address space.

- **Code block**, also called the *text section*, contains machine instructions that the CPU can execute. It is composed of codes from an executable image and many libraries. Most modern operating systems do not allow a process to modify the code block to enable multiple processes to share the code (see Fig. 4.1 on page 87). In fact, code sharing is automatically set up by the operating system, and is transparent to application processes. The processes behave as if each has its own copy of the block text section. Writes to the block are considered illegal, and are trapped by the operating system to forcefully terminate the process.

  ≫ As mentioned in Fig. 4.4 a code block is also called a text section of the program.

- **Data block** contains the program's own static data. It is divided into three sub-blocks. (1) The "initialized read-only" data is initialized by the program: its value is determined at the program compilation time and cannot be changed by a process. These read-only data may be shared by multiple processes. (2) The "initialized read-write" data is initialized by the program. (3) The "uninitialized" data is not kept as a part of the compiled program, and hence, the program does not initialize it. However, UNIX systems guarantee that all uninitialized program data are initialized to zeros when the process address space is setup. A process can modify any data that is not marked read-only. Figure 4.5 depicts which variable/constant declarations in C programs are part of which data sub-blocks.

```
Consider the following global variable declarations in C.
const int n = 10; // Data block category (1): initialized read-only data
int v = 10; // Data block category (2): initialized read-write data
int vec[10]; // Data block category (3): unitialized data
```

Figure 4.5: *Various data blocks of an application process in UNIX systems.*

- **Stack block** The dynamic data of a process is partitioned into two different blocks: stack and heap. A *stack* is a data structure in which entries are added to and removed from one end, which is called the *stack top*. Two operations, namely *push* and *pop*, are implemented to access a stack. A push operation execution adds an entry at the top of a stack, and a pop operation execution removes and returns the current top entry from the stack. These two operation executions, respectively, expand and shrink the process address space. Every process has a private stack, which is used to store local variables and parameter values for function executions and return-addresses for function calls. The stack is not the program's static data, and is required only at runtime.

- **Heap block** is used to allocate more data entities dynamically to the process address space. Unlike the stack, space allocations and deallocations are done in arbitrary order and size. These are not the static data of the program, and are required only at runtime. The allocation and deallocation of space to the heap expands and shrinks, respectively, the process address space.

≫ The code and read-only data are often called a program and the program's data, respectively, and multiple processes can safely share them. In fact, most operating systems set up the program and data sharing scheme without the knowledge of the program or the processes executing the program. The other parts on the address space are called process data (private data). A process may or may not share its data with other processes; neither can the operating system set up data sharing without explicit requests from the processes. The stack of one process is never shared with other processes. When two processes execute the same program, they have the same program but operate on different private data. The memory-management subsystem helps in routing references from the (same) program to respective private data copies.

Note that, as shown in Fig. 4.4 on page 91, there is a gap between the stack and the heap. This gap allows both of them to grow (and shrink) dynamically at runtime. The operating system has to make sure that the boundaries between the heap and the stack do not cross each other. When they do cross, an exception is generated, and the system terminates the process. Note that the area in the gap is not part of the address space, but is allocated to the address space on requests for expansion. Although the heap appears to be a compact space in Fig. 4.4, in reality, many unallocated chunks might be embedded in the heap. These chunks are out of the process address space too.

4.5.2 Kernel Address Space

The kernel space is also structured like the one shown in Fig. 4.4, but with a difference, especially for stacks. When an application process runs in its private address space, it uses the private stack and the private heap. When it executes in the kernel space, it uses a separate stack to store entries for kernel function executions. There is no single kernel stack. All active processes in the kernel use separate kernel stacks, which are different from their user space stacks. Kernel processes only have kernel stacks and no user-space stacks. In most operating systems, kernel stacks are small, pre-allocated, and of fixed sizes. The kernel also allocates and deallocates dynamic space to and from itself to manage its dynamic data. Unlike kernel stacks, all processes share this dynamic data when they operate in the kernel space. Of course, they all share the kernel code.

4.6 Process Execution Mode

Except for a small limited period at the very beginning of system bootstrapping, the CPU always executes machine instructions in the context of a process. It was mentioned in Section 1.6 on page 23 that most modern processors have at least two operating modes, namely, the user and the kernel. Accordingly, we say a process is executing in the user mode or in the kernel mode as indicated by the operating mode of the processor at that juncture.

All application processes run partly in the user mode and partly in the kernel mode. When in the user operating mode, a process can only execute instructions and access data from its private address space. The user-operating mode has lesser privileges than the kernel-operating mode. Certain processor features are not available to user mode processes. The processor hardware makes privileged resources and instructions inaccessible to user mode processes. This is to prevent application processes from interfering with the operating system and hardware resources directly in their private address spaces. For example, an application process cannot execute I/O instructions to read or write data directly from or to I/O devices.

An application process alternates between the two operating modes. It does normal work in the user mode. When it needs services from the operating system, it avails the services in the kernel mode. There are special machine instructions that application processes execute to switch from the user mode to the kernel mode. The execution of these instructions places the process in the kernel space, and the process starts executing operating system programs (see Fig. 1.10 on page 24). When an application process executes operating system programs, we say the process is "as if" a kernel process: the CPU executes kernel programs on behalf of the process. The kernel does not have restrictions in executing any instruction or accessing any hardware resource. Therefore, a user process in the kernel mode can access any resource, and can execute any machine instruction including the privileged ones until it reverts to the user mode. The original kernel processes, in contrast, are always executed in the kernel mode.

≫ A user process switches to the kernel mode only when an interrupt occurs, or an exception is generated, or it makes a system call. This is called address space switch. The system call needs execution of special machine instructions by the process. The address switching in the other two cases (interrupt and exception) is automatic. We will study them in Chapter 12.

4.7 Process Identification Information

In a multiprocess system, there are many concurrent processes operating in the system at any point of time. Different processes represent different computations, and these are distinguishable entities. To interact with a process, we need a means to identify the particular process. A process is uniquely identified by its pid—the process identifier. The pid is the most important process attribute, and is normally a positive integer number that uniquely identifies a single process from all processes currently active in the system.

The operating system assigns a unique pid value to a process when it creates the process, and this value remains unchanged until it destroys the process. Once the system allocates a process descriptor for a newly created process, its location does not generally change. Consequently, the address of a descriptor may be used to identify a process. However, because of the limitation of physical resources (especially the main memory) we may need to recycle a limited number of

≫ Users identify a process by its pid. For example to delete or destroy a process, in UNIX systems, one can execute the **"kill -9 pid"** command.

descriptors for new processes. So, within a short span of time, many different processes may use the same descriptor one after another which might make it difficult to distinguish a terminated process from a new process. Thus, in practice, pid values are kept independent of process descriptor addresses, and the values are stored in process descriptors (see Fig. 4.2 on page 88). In most systems, pid values are 32-bit or 64-bit unsigned integers. Eventually pid values need to be recycled, but the rate of recycle is comparatively slow. The operating system guarantees that the pid values of all current processes are distinct, and that they are active values. A new process always acquires a non-active pid value.

≫ In UNIX systems, 0 is not a valid pid value.

The pid values allow processes to be referred in a position-independent manner. Process descriptors may reside anywhere in the kernel, and, if necessary, they may be relocated on the fly. Given a pid value, we need to find the corresponding process. We need a map between active pid values and the corresponding process descriptors. The operating system implements this map from pid values to process descriptors.

≫ When a process executes a new application program, the pid value transcends to the new program execution. That is, the new program execution has the same pid. Hence, transcending the same pid helps users to track program executions via the same process.

For each process, in addition to the pid, the operating system also maintains many different identifiers. These identifiers are used for controlling a process's accesses to system resources and for interprocess communications. These identifiers will be discussed in the context of Linux operating system in Chapter 17. Here we discuss only two other identifiers.

4.7.1 User Identifier

Each process is owned by a legal user. The identity of the owner is stored in a process attribute, named user identifier (uid), see Fig. 4.2 on page 88. The uid and its other related identifiers, normally used to determine a process's privileges to utilize various resources, are discussed in detail in Chapter 13.

4.7.2 Process Group Identifier

A *process group* is a collection of processes that are engaged in performing a common task. The group can expand and shrink in its lifetime. The group is uniquely identified by what we call a *process group identifier* (*pgid*, for short). Processes in a group may have the same *pgid* value, but will have different pid values. Different groups are guaranteed distinct *pgid* values. A group may have a designated *group leader* process that performs particular tasks for the group. The *pgid* is primarily used for interprocess communications and to multicast messages to all processes in the group. Given a *pgid* value, it is necessary to find all the processes in the group. The operating system implements a mapping function for this purpose.

4.8 Interprocess Relationship

Processes may be related to one another by different relationships.[2] Each process is created by another, with the exception of the very first one. The

[2] A relationship is an association between two or more entities.

former is called a *child* of the latter, which is called the *parent*. All children of a process are *siblings*. The parent–child relationships form a tree of processes, and the tree has a designated root process, as shown in Fig. 4.6. A process with no child is a leaf process; otherwise, it is an internal process. The parent-child relationships are implemented through a set of pointers that are stored in a process descriptor. This set includes a parent pointer, a child pointer, and a sibling pointer (see Fig. 4.7). The parent pointer points to the descriptor of the parent. (The parent pointer of the root process points to NULL as the root has no parent.) The child pointer points to the eldest child alive or to NULL if there is no child alive. The sibling pointer points to the next younger sibling of the process that is alive. When a process terminates, it is removed from the tree.

≫ When an internal process in the process-tree completes its execution, all its children become orphans. In UNIX systems, orphans become de facto children of the root of the tree; they call the root the "init" process. There are other systems where all processes under the terminating process are forcefully terminated by the system by way of cascading terminations.

4.9 Process State

The *state* of an agent defines a condition, a situation, or a mode of the agent. Here, by a *state* of a process we mean a gross indication of the current condition and situation of the process. In fact, the program a process is executing, the values of all the data in its address space, and the process context collectively determine the state of the process. For the sake of the operating system, these states of processes are partitioned into a few categories. These categories are themselves called *process states* in operating system terminology. Each category is identified by a name, and is represented by a value. The current state value of a process identifies the category of the process. As execution of a process proceeds, the process changes its state. The most commonly used state values are described below.

≫ In a multithreaded process, the process does not run, but the threads do. Consequently, threads within the process will have states.

1. **Starting:** This state value signifies that the process has just been created. The operating system is setting up the initial execution context and the

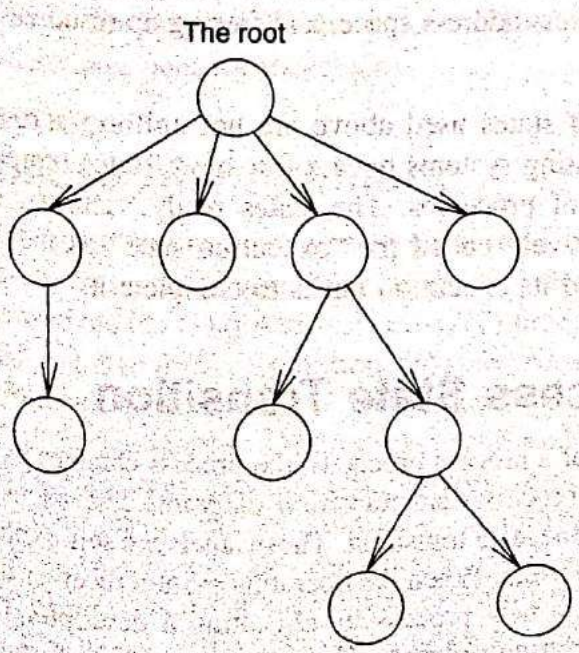

Figure 4.6: *A typical rooted tree of all processes.*

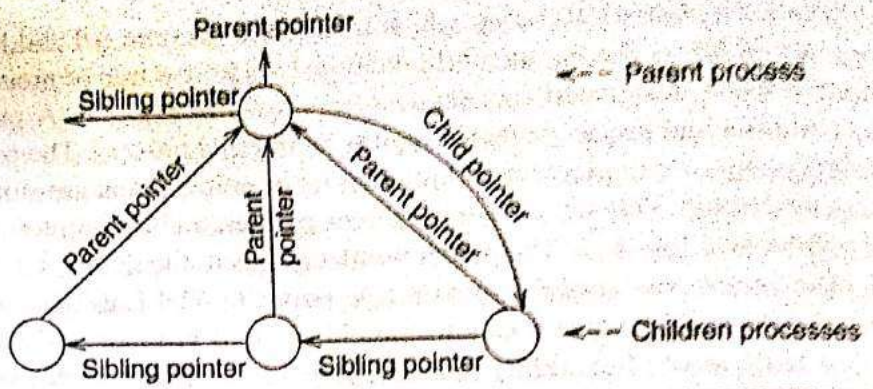

Figure 4.7: *The parent-child relationship among processes—a representation.*

initial private address space for the process, and will be allocating it some resources.

2. **Ready:** This state value signifies that the process is ready to run, but the CPU not yet being available, is waiting in the CPU queue, where it will wait until the operating system allocates it the CPU.
3. **Running:** This state value indicates that the process is running on the CPU. The CPU is executing instructions from the current address space (process private address space or the kernel space) and the process execution is progressing. The operating system allocates new resources to the process as need arises.
4. **Waiting:** This state value indicates that the process is currently waiting for some event (such as the completion of an I/O operation) to take place in the system. It may not resume its execution until the event occurs. Until then it sleeps in an event queue, and the operating system will awaken it when the event indeed occurs.
5. **Exiting:** This state value indicates that the process execution is complete, either normally or abnormally. The operating system is cleaning up the process address space, and freeing up resources acquired by the process.

The names of states used above are not uniform across operating systems. Some operating systems have a few more states representing different (sub) conditions of processes. The states ready, running, and waiting are sometimes collectively called the "execution state" as they denote that the process has started its execution but is to complete it.

4.10 Process State Transition

As the execution of a process proceeds, the process changes its state. Figure 4.8 depicts a typical process-state transition diagram. The edges are labelled by actions that cause the state transition. Those labels are self-explanatory. Initially, a process is non-existent. When the operating system receives a process creation request, it creates a new process by allocating a descriptor from the pool of unused process descriptor objects. The new process is then in the starting state.

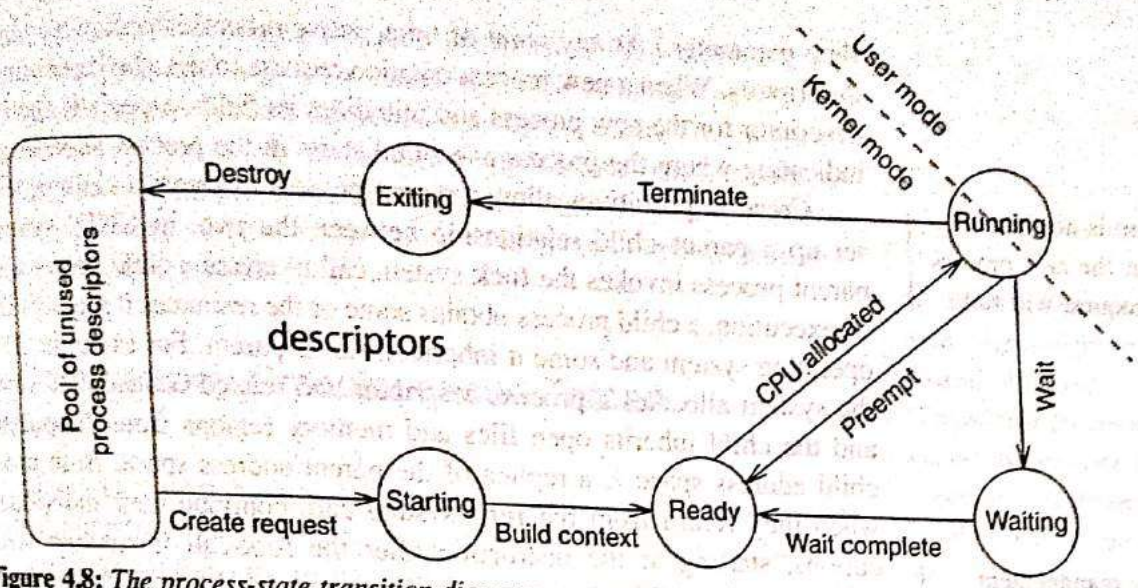

Figure 4.8: *The process-state transition diagram.*

and the system builds up the initial execution context. (In UNIX systems, a process is created by another process called the parent process. The system duplicates the parent's address space for the child.) The new process then becomes ready to run, and waits in the CPU queue until the CPU is allocated to it. When it gets the CPU, the process starts running, that is, the CPU executes instructions from the "current logical" address space of the process. After a while, either the process terminates and enters the exiting state, or it releases the CPU and waits for some event to occur. When the event does occur, the process again becomes ready to run, and rejoins the CPU queue. In addition, in some situations, the CPU allocated to a running process is preempted (i.e., it is taken away) by the operating system, and the process enters the ready state and joins the CPU queue. When a process enters the exiting state, the operating system cleans up the process; for example, it releases all the resources acquired by the process. The system finally destroys the process. (The process descriptor is returned to the descriptor pool for recycling.)

4.11 Process Management

Process management is the most important subsystem in an operating system. This subsystem controls activities of the processes. It implements management primitives for (1) process creation, (2) process termination, (3) process suspension, (4) process resumption, (5) interprocess communication, and (6) synchronization. Processes, to exchange data among themselves, use interprocess communication primitives; synchronization primitives are intended to coordinate their execution flows with respect to one another. Interprocess communication and synchronization primitives will be discussed in Chapters 6 and 7, respectively. Other primitives are briefly discussed below.

Operating systems typically implement a fixed array of process descriptors (or pointers to descriptors). (The size of the array is a configurable compile

≫ In uniprocessor systems, at the most one process may be running on the CPU while other processes, if any, are in the ready or waiting states. In multiprocessor systems, many processes can run simultaneously, and they run on different CPUs. When a process is not running, it is in the kernel mode. When it is in the running state, it is either in the kernel mode or in the user mode, depending on which program it is currently executing. Therefore, mode switching happens only when it is running, and state change happens only when it is in the kernel mode (see Fig. 4.8).

time parameter.) At any point of time, active processes occupy some of those descriptors. When a new process creation request comes, the kernel finds a free descriptor for the new process and initializes its fields, especially the PC value indicating where the process execution starts in the process address space.

≫ If there is no free descriptor, the new process creation request will fail.

Creation primitives allow a parent process to create a child process, and set up a parent–child relationship between the two. In UNIX systems, the parent process invokes the **fork** system call to create a child process. To start its execution, a child process obtains some of the resources it requires from the operating system and some it inherits from its parent. For example, in UNIX, the system allocates a process descriptor and related control data structures, and the child inherits open files and memory regions from the parent. The child address space is a replica of the parent address space. Both processes, when they return from the **fork** system call, continue their individual executions, starting at the instruction after the fork-call instruction with one difference: the child receives zero as the return code of the **fork** but the parent receives the child's pid as the return code. (You may recall from Section 4.7 on page 93 that zero is not a valid pid value in UNIX systems.) The kernel data structure values for the two processes are the same except for a few such as pid and the parent pid. Both the parent and its child may run concurrently, or the parent may wait until the child completes its execution. They have different private address spaces, but they execute the same program. The operating system does automatically set up the program sharing mechanism. Each of them may continue executing the same program, or may load a different program in their address spaces for the execution. In UNIX systems, a process executes an **exec** system call to execute a new program; the system call involves destroying the old address space and setting another up instead.

≫ Process management is the core function of the operating system. It simply creates processes, supplies them with resources when needed, coordinates their access to resources, and eventually removes the processes whose works are completed.

Termination primitives help a process to terminate itself, or a parent process to terminate a child. In UNIX systems, a process terminates itself by invoking the **exit** system call to inform the operating system that it has finished its execution. The system cleans up the terminating process by closing all open files, and releasing all acquired resources. In UNIX systems, the terminating process becomes de facto dead (a zombie) at this point. Later, the parent receives the execution status of the child, and the child process becomes terminated. At this point, the child process descriptor is freed.

Suspension primitives allow a process to suspend conditionally or unconditionally its own execution. In UNIX systems, a **wait** system call suspends a process until one of its children terminates. A **sleep** system call unconditionally suspends a process execution for a specified duration. Resumption primitives are called to resume execution of a previously suspended process.

4.12 Threads and their Management

In traditional operating systems, a process executes a single sequence of instructions from its private address space. We call this "execution flow". At any point in time, the CPU executes at the most one machine instruction on

behalf of the process. Two program executions (even if they involve executing the same program) are represented by two different processes. Resources acquired by one process cannot be used by the other. This imposes a severe restriction on some high performance applications that need many concurrent activities (multiple flows) in the same address space. Modern multithreaded operating systems come to their help and support multiple flows in the application process.

4.12.1 Concurrency Granularity and Thread Abstraction

Processes are units of work assignment, resource allocation, and concurrency in traditional operating systems. Each process is a single strand or flow of program execution, and holds an exclusive address space. These systems allow concurrency at the process level only. Process level concurrency is coarse grained, and is often not suitable for high performance applications. There is no concurrency in a single process address space. In some modern operating systems, address spaces are "decoupled" from processes as far as the execution flow is concerned. A single address space can have multiple strands of executions. That is, those systems allow concurrency within a single process address space, and many strands share the same address space at the same time. Consequently, unlike in traditional operating systems, a process can execute multiple sequences of instructions from the same address space. Each strand of execution sequence is called a *thread*. Minimally, each thread has its own PC, a stack, and some data registers. It is an abstraction of an agent that executes instructions, and nothing else. A thread is effectively a summary of a sequence of instruction executions: it has a beginning, a sequence, and an end. At any given point in time during a thread execution, there is a single point of execution (see Fig. 4.9). In the figure, thread 1 starts its execution at function1, thread 2 at function2, and thread 3 at function3.

≫ Some authors use the term "lightweight process" instead of thread. (Traditional processes are known as heavyweight processes.) A thread is similar to a traditional process in the sense that each is a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program execution, and takes advantage of the resources allocated for the program execution and of the environment of its execution.

4.12.2 Thread and Process Relationship

A thread is an independent strand in a process, and the process is said to own that thread, and the thread is contained in the process (see Fig. 4.9). Threads

≫ A thread is strictly a private resource of a process, and multiple processes cannot share it. The threads in a process are not visible to other processes.

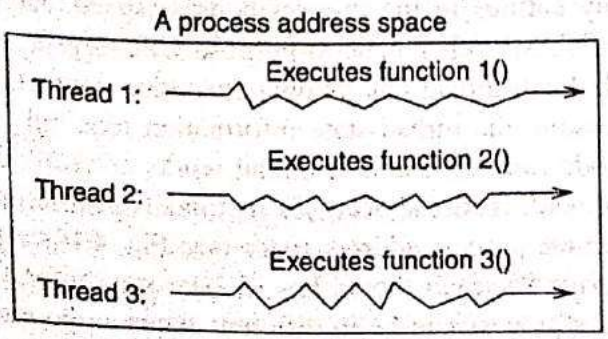

Figure 4.9: *The multiple threads of execution flows in a process address space.*

separate the notion of program execution from the rest of the traditional definition of a process. A process no more executes programs, and is merely a holder of resources; it creates an environment in which its threads run. A single thread executes a program concurrently with other threads in the same process. The (execution) state is no more associated with the process, but with its threads.

≫ A thread is the unit of work assignment within a process, even though the process is the unit of resource allocation in the system.

Note that a thread is not a process, and it cannot run on its own. A thread always runs within a process. A process may have many threads, but a thread always has one container process. A traditional process is one that has a single thread. A thread termination does not necessarily imply that the container process execution is over. When the last thread terminates, the process execution is complete and the process is said to have terminated.

≫ A thread is an independent sequence of instructions that can only be executed within the scope of a process.

4.12.3 The Benefits of Threads

Process threading is a convenient means for exploiting concurrency within the same process. In Fig. 4.9, the process owns three threads and these threads run concurrently and can perform same or different tasks. Threads are units of concurrency in the process address space. One may use threads when a program has many independent tasks that can be executed concurrently. Threads acquire resources in the context of the container process, share all the resources allocated to the process, and cooperate with one another in a single program execution. Extensive sharing of resources among (sibling) threads makes thread creation/destruction an inexpensive operation compared to process creation/destruction. Creation of a process involves setting up a new address space; creation of a thread does not.

≫ A thread starts its execution at a function identified by the creator, and ends when the function returns or it executes a thread-exit kind of instruction. When any thread executes a process-exit kind of instruction, the operating system terminates the process with all its threads. If any thread executes an **exec** system call to execute a new program, all other threads are gone, and it starts executing the new program in the same process.

A multithreaded application may create many threads in the executing process. (The process starts its life with one or a few default threads.) A thread can create another thread; and they are considered sibling threads. There is no parent-child relationship among threads. Thread creation and startup are semantically equivalent to an asynchronous function call; the creator and the created threads run asynchronously at their own speeds. All threads share the same address space (of the process), and all resources (such as open files, memory regions) acquired by the process (in any thread). However, each thread needs some "private" space within the process. For sequential flow of control, a thread must carve out some of its own private resources within the container process. Even though any thread can make references to any entities in the process address space, each thread has its "own" copy of PC, SP, other general-purpose CPU registers, and a private user space stack. Each thread has its own execution context, and we need a control block to store the thread-state information required for thread management. The code running within a thread works correctly only within the context of that thread. A thread accesses its thread-specific information, and the global data in the process address space (see Fig. 4.10). As mentioned in Section 4.9 on page 95, each thread has its own state information, and different threads in a process can be in different states at the same time.

≫ Threads are independent strands, but they are not truly independent. Although a thread has its *own* copy of some resources such as the stack, any other sibling thread can read or write elements in the private copy!

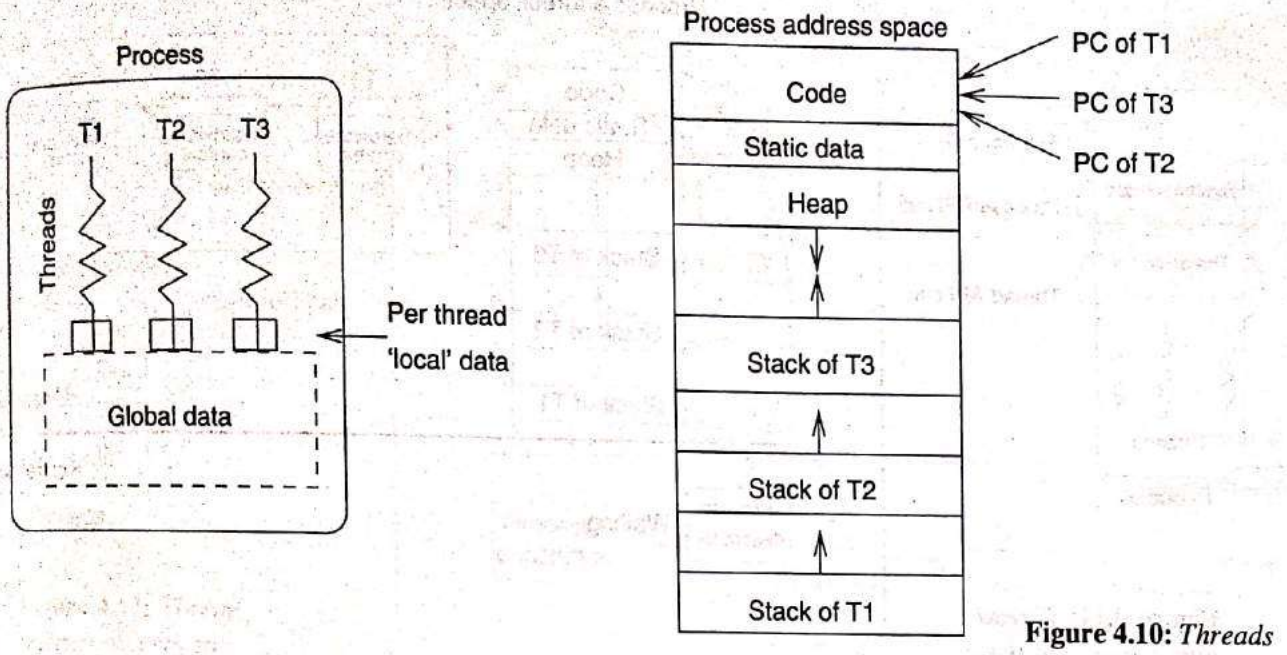

Figure 4.10: *Threads sharing process data.*

4.12.4 Thread Synchronization

Traditional processes are single threaded. The term thread gets its significance only when the system is multithreaded. In a multithreaded system, threads within a process can independently perform different tasks at the same time. This is illustrated in Fig. 4.10. As threads share all data and resources in a process address space, one thread can influence executions of other threads in the process. Accesses to the global data and resources may need to be synchronized. Application programs themselves, and not the operating system, provide this synchronization.

≫ Process level concurrency is course grained; thread level concurrency is fine grained and need more synchronization solutions. Threads encapsulates concurrency within a process, and the process encapsulates protection within the system.

4.12.5 Thread Management

At any instant of time, the CPU can execute a single thread of a process on a traditional processor. When the need arises, the CPU is swithed among sibling threads. This is called *thread switching*, in contrast to context switching where the CPU is switched between (threads of) two different processes. Extensive sharing of resources among sibling threads makes thread switching an inexpensive operation compared to context switching. Thread switching requires only switching between subsets of register contents, including PC and SP registers. Thread switching can be implemented in user space or kernel space or both.

Thread Library

In some systems, thread switching is implemented in a system library. In those systems, thread switching is done in the user space, and not in the kernel space. Operating system service is not required for thread switching. The library manages thread-specific register sets. In fact, the operating

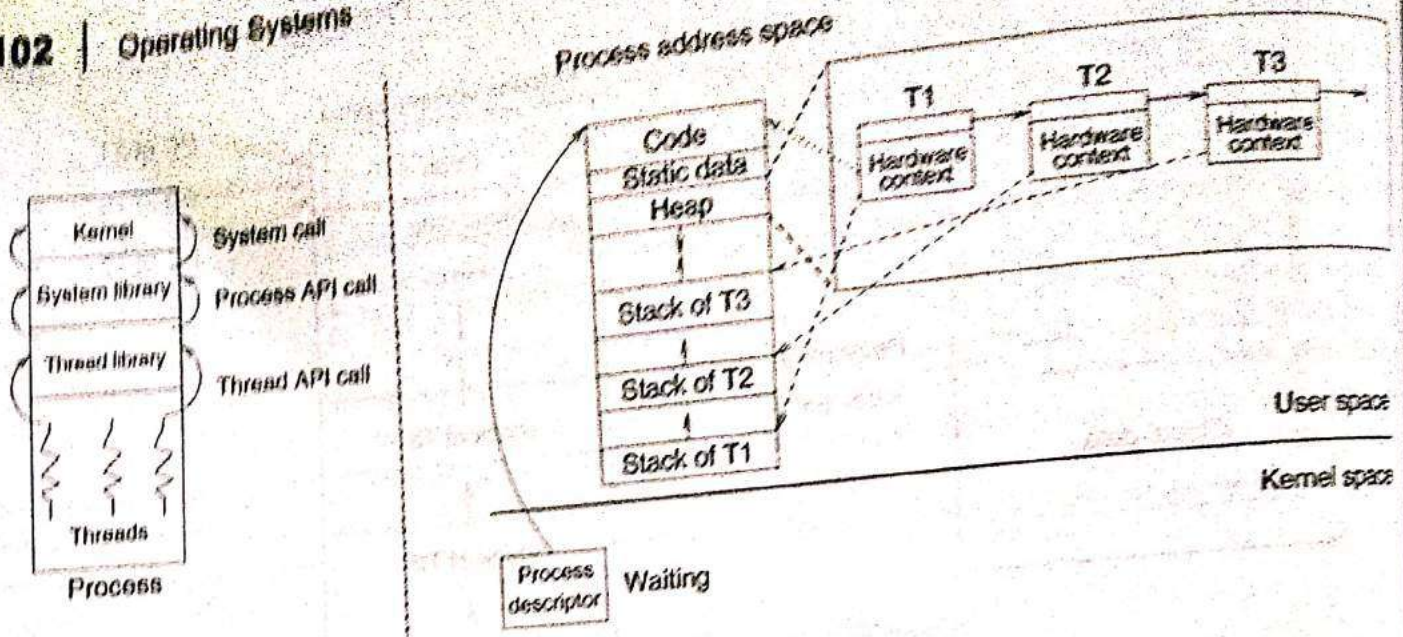

Figure 4.11: *Thread interaction with the kernel via library.*

≫ The operating system maintains the process state, but the library maintains the states of all threads. The system is said to be single-threaded, but application processes are multithreaded. The system schedules processes on the CPU, but the library schedules threads for their executions when the process has the CPU. When the operating system preempts the CPU from the process, all its threads are blocked.

≫ A conventional processor executes only one thread at a time. Many sibling threads though, can run concurrently on a single hyper-threaded processor as if they are in a multiprocessor system.

system is not aware of the presence of multiple threads in applications. It sees only processes. See Fig. 4.11. If any thread executes a system call, the system treats it as a call from the owner process, and the entire process has to wait until the system call returns. (Of course, the thread library can convert each blocking system call to a non-blocking system call, and switch to another thread until the operating system finishes the system call processing.) In such systems, the operating system does not have thread management capabilities. The library implements APIs related to thread handling. The thread management data structures reside in the heap as shown in the right part of Fig. 4.11. The process stores a few references to those structures in the "static" data section.

Multithread Kernel

There are other operating systems where the kernel is aware of process threading. Management of threads—creation, scheduling, and destruction—is done by the operating system. See Fig. 4.12. These systems implement thread handling system calls. The operating system is aware of every one of the threads created by processes, and a thread descriptor object in the kernel represents the thread. Thread switching is purely a kernel action in such systems and hence is more time consuming than switching threads in the user space. Nonetheless, creation of new threads and CPU switching among sibling threads are inexpensive compared to the creation of processes and CPU switching among processes, respectively. Thread switching involves saving a register set of one thread and reloading that of another. Threads in a process are scheduled for execution independently of one another. Even if one thread is blocked, others can execute the applications. These systems are truly multithreaded: both kernel- and application processes can have multiple threads.

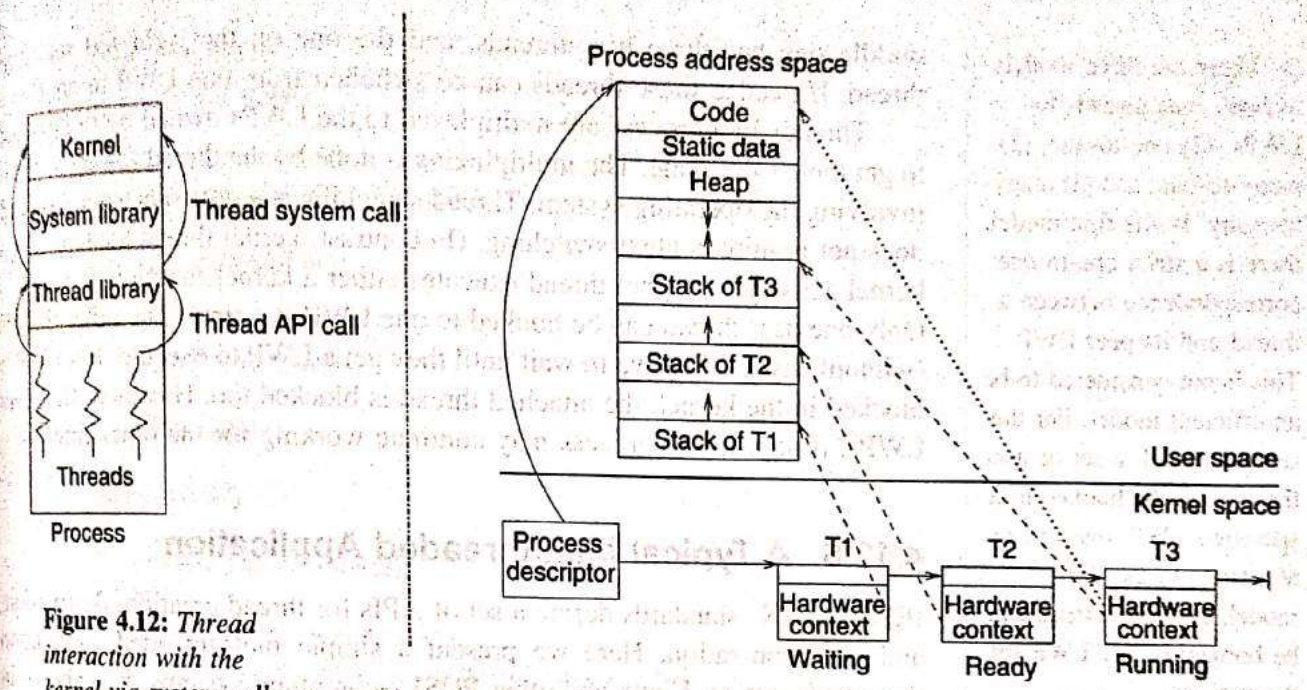

Figure 4.12: *Thread interaction with the kernel via system call.*

A Mixed System

Some operating systems such as Solaris 2 support a mix of "user-level" as well as "kernel-level" threads to have the best features of both: a hybrid thread-handling support. These systems distinguish between two kinds of threads: user threads and kernel threads. The two are connected through an intermediate abstraction often called lightweight processes (LWPs). See Fig. 4.13. The operations (creation, scheduling, and destruction) to manage user threads are implemented in a system library in the user space. The operating system has no knowledge about user threads. It supports system calls for the management of LWPs. The operating system sees LWPs and kernel threads, and application processes see both LWPs and user threads. Threads do actual work via LWPs. In such systems, the user thread is the unit of work assignment in a process, the kernel thread is the unit of CPU allocation by the operating system, and the process is the unit of resource allocation in the system. Each process contains at least one LWP. As shown in the figure, the process has three LWPs: the LWP on the left has two user threads, the

≫ Kernel threads (and hence LWPs are scheduled on CPUs by the kernel, but user threads are scheduled (also known as hooked) on LWPs by the library.

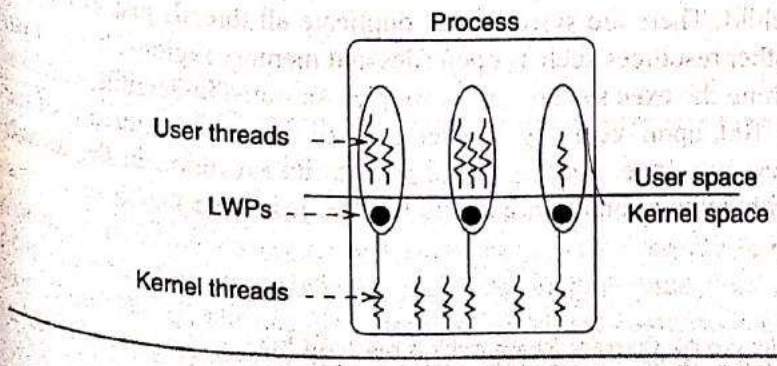

Figure 4.13: *Hybrid thread management scheme.*

middle one has three user threads, and the one on the right has one user thread. If needed these threads can be switched from one LWP to another.

Threads in a process are multiplexed to the LWPs owned by the process to get their work done. The multiplexing is done by the thread library without involving the operating system. Thread switching is a user space activity, and does not require context switching. (In contrast, kernel thread switching is a kernel activity. A kernel thread executes either a kernel function or a LWP.) Only one user thread can be hooked to one LWP at a time. The other threads (without any LWP) have to wait until they get a LWP to execute. If a LWP is blocked in the kernel, the attached thread is blocked too. However, the other LWPs, if any, in the process may continue working for the other threads.

≫ *There are three models to hook user threads to LWPs: (1) one-to-one, (2) many-to-one, and (3) many-to-many. In the first model, there is a strict one-to-one correspondence between a thread and its peer LWP. This is not considered to be an efficient model. For the second model, a set of user threads can be hooked to a specified LWP, one thread at a time. In the third model, any user thread can be hooked to any LWP (of the process).*

4.12.6 A Typical Multithreaded Application

IEEE POSIX[3] standards define a set of APIs for thread creation, destruction, and synchronization. Here we present a simple multithreaded application that would run on Linux and other POSIX-compliant systems. It is shown in Fig. 4.14, and is written in the C programming language.

The application creates ten threads by invoking the **pthread_create** API, and each of them executes the myPrint function concurrently. They each have a different argument to the function. Each thread prints the function argument and returns from the function. The return is equivalent to thread-exit. The main thread waits (by invoking the **pthread_join** API) until all sibling threads exit. When the main thread returns from the main function, the process exits. Note that all threads run concurrently, and their prints may get mingled; different runs of the application may have different output minglings. The behaviour of a multithread application may not be deterministic, and the behaviour depends on how the threads are scheduled for execution.

≫ *Based on thread- and address-space abstractions, run-time systems may be classified into four types: (1) one address space and one thread system (e.g., MS-DOS), (2) one address space and multiple threads (e.g., the Java virtual machine), (3) multiple address spaces with one thread per address space (e.g., early UNIX), and (4) multiple address spaces with multiple threads per address space (e.g., Windows, Linux, and many modern operating systems).*

4.12.7 Issues with Multithreading

There are a few known issues with multithreading. Different systems solve the issues differently.

1. When a thread executes a **fork** system call to create a new process, does the child process duplicate all threads from the parent or just the calling thread? Some systems create one- or a few default threads in the child. There are systems that duplicate all threads just as they do for other resources such as open files and memory regions. In contrast, handling the **exec** system call is simple. As stated in Section 4.12.3 on page 100, upon receiving an **exec** call, all the previous threads of the process are gone, and the process starts its execution in the default multithreading condition as if its life has just started.

≫ *In Linux system, when a process with multiple threads executes the fork, the system creates a new process with exactly one thread—an exact replica of the thread that invoked the fork.*

[3] POSIX stands for Portable Operating System and it is based on UNIX.

```
#include <stdio.h>
#include <pthread.h>

const numThreads = 10;
char threadArgs [numThreads] [100];

void* myPrint (void* arg)
{
    char* val = (char*) arg;
    printf (" %s\n", val);

    return NULL;
}
int main (void)
{
    pthread_t t[numThreads];
    int i;
    for (i=0; i<numThreads; i++) {
      sprintf (threadArgs [i], "My thread number is %d\n", i);
      pthread_create (&t[i], 0, myPrint, (void*) threadArgs [i]);
    }
    /* wait for all threads to finish */
    for (i=0; i<numThreads; i++)   pthread_join(t[i], 0);
    return 0;
}
```

Figure 4.14: *A typical multithreaded application.*

2. How does a multithreaded process handle its signals? Is a signal delivered to a process received by any thread, a few particular threads, or all threads? We will revisit these questions in Section 6.4.1 after we have discussed what signals are.
3. How threads are scheduled? At what level are they scheduled, the user level or the kernel level? We will discuss these issues in the next chapter.
4. As mentioned in Section 4.12.4 on page 101, protecting shared data is an important concern in multithreading. Unprotected access could corrupt data and concurrent accesses may lead to unwanted situations such as deadlock. These issues and some basic solutions are discussed in Chapter 7.

Summary

This chapter introduces the most fundamental concept of an operating system—a process. A process is a program in action. An operating system manages a program execution by associating a process to the execution. The dynamism (i.e., runtime activities) of an operating system is abstracted as processes. There are both user and kernel processes. Kernel processes execute only operating system programs strictly in the kernel mode, but user processes execute both applications- and operating-system programs. The kernel processes perform tasks specific to the operating system.