Chapter 9 Operating System Support

Learning Objectives

After studying this chapter, you should be able to:

- Summarize, at a top level, the key functions of an **operating system (OS)** .
- Discuss the evolution of operating systems for early simple batch systems to modern complex systems.
- Explain the differences among long-, medium-, and short-term scheduling.
- Understand the reason for memory **partitioning** and explain the various techniques that are used.
- Assess the relative advantages of paging and segmentation.

Define virtual memory.

Although the focus of this text is computer hardware, there is one area of software that needs to be addressed: the computer's OS. The OS is a program that manages the computer's resources, provides services for programmers, and schedules the execution of other programs. Some understanding of operating systems is essential to appreciate the mechanisms by which the CPU controls the computer system. In particular, explanations of the effect of interrupts and of the management of the memory hierarchy are best explained in this context.

The chapter begins with an overview and brief history of operating systems. The bulk of the chapter looks at the two OS functions that are most relevant to the study of computer organization and architecture: scheduling and memory management.

9.1 Operating System Overview

Operating System Objectives and Functions

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having two objectives:

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.

Let us examine these two aspects of an OS in turn.

THE OPERATING SYSTEM AS A USER/COMPUTER INTERFACE

The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in **Figure 9.1**. The user of those applications, the end user, generally is not concerned with the computer's architecture. Thus the end user views a computer system in terms of an application. That application can be expressed in a programming language and is developed by an application programmer. To develop an application program as a set of processor instructions that is completely responsible for controlling the computer hardware would be an overwhelmingly complex task. To ease this task, a set of system programs is provided. Some of these programs are referred to as **utilities.** These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer makes use of these facilities in developing an application, and the application, while it is running, invokes the utilities to perform certain functions. The most important system program is the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

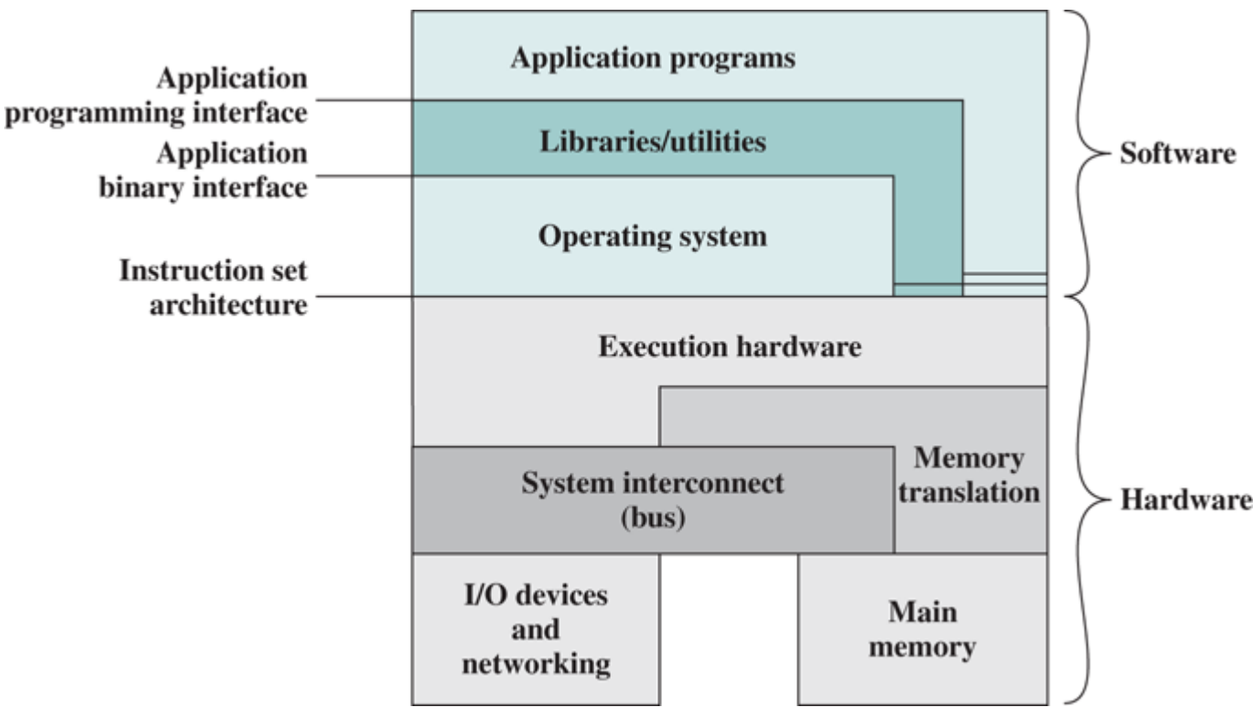


Figure 9.1 Computer Hardware and Software Structure

Briefly, the OS typically provides services in the following areas:

- **Program creation:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of **utility** programs that are not actually part of the OS but are accessible through the OS.
- **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles all of this for the user.
- **Access to I/O devices:** Each I/O device requires its own specific set of instructions or control signals for operation. The OS takes care of the details so that the programmer can think in terms of simple reads and writes.
- **Controlled access to files:** In the case of files, control must include an understanding of not only the nature of the I/O device (disk drive, tape drive) but also the file format on the storage medium. Again, the OS worries about the details. Further, in the case of a system with multiple simultaneous users, the OS can provide protection mechanisms to control access to the files.
- **System access:** In the case of a shared or public system, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as arithmetic overflow, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must make the response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.
- **Accounting:** A good OS collects usage statistics for various resources and monitors performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes. **Figure 9.1** also indicates three key interfaces in a typical computer system:
  - **Instruction set architecture (ISA):** The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software. Note that both application programs and utilities may access the ISA directly. For these programs, a subset of the instruction repertoire is available (user ISA). The OS has access to additional machine language instructions that deal with managing system resources (system ISA).
  - **Application binary interface (ABI):** The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.
  - **Application programming interface (API):** The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with **high-level language (HLL)** library calls. Any system calls are usually performed through libraries. Using an API enables application software to be ported easily, through recompilation, to other systems that support the same API.

THE OPERATING SYSTEM AS RESOURCE MANAGER

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

Can we say that the OS controls the movement, storage, and processing of data? From one point of view, the answer is yes: By managing the computer's resources, the OS is in control of the computer's basic functions. But this control is exercised in a curious way. Normally, we think of a control

mechanism as something external to that which is controlled, or at least as something that is a distinct and separate part of that which is controlled. (For example, a residential heating system is controlled by a thermostat, which is completely distinct from the heat-generation and heat-distribution apparatus.) This is not the case with the OS, which as a control mechanism is unusual in two respects:

- The OS functions in the same way as ordinary computer software; that is, it is a program executed by the processor.
- The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

Like other computer programs, the OS provides instructions for the processor. The key difference is in the intent of the program. The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs. But in order for the processor to do any of these things, it must cease executing the OS program and execute other programs. Thus, the OS relinquishes control for the processor to do some “useful” work and then resumes control long enough to prepare the processor to do the next piece of work. The mechanisms involved in all this should become clear as the chapter proceeds.

Figure 9.2 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes the **kernel**, or **nucleus** , which contains the most frequently used functions in the OS and, at a given time, other portions of the OS currently in use. The remainder of main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the OS and memory-management hardware in the processor, as we will see. The OS decides when an I/O device can be used by a program in execution, and controls access to and use of files. The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

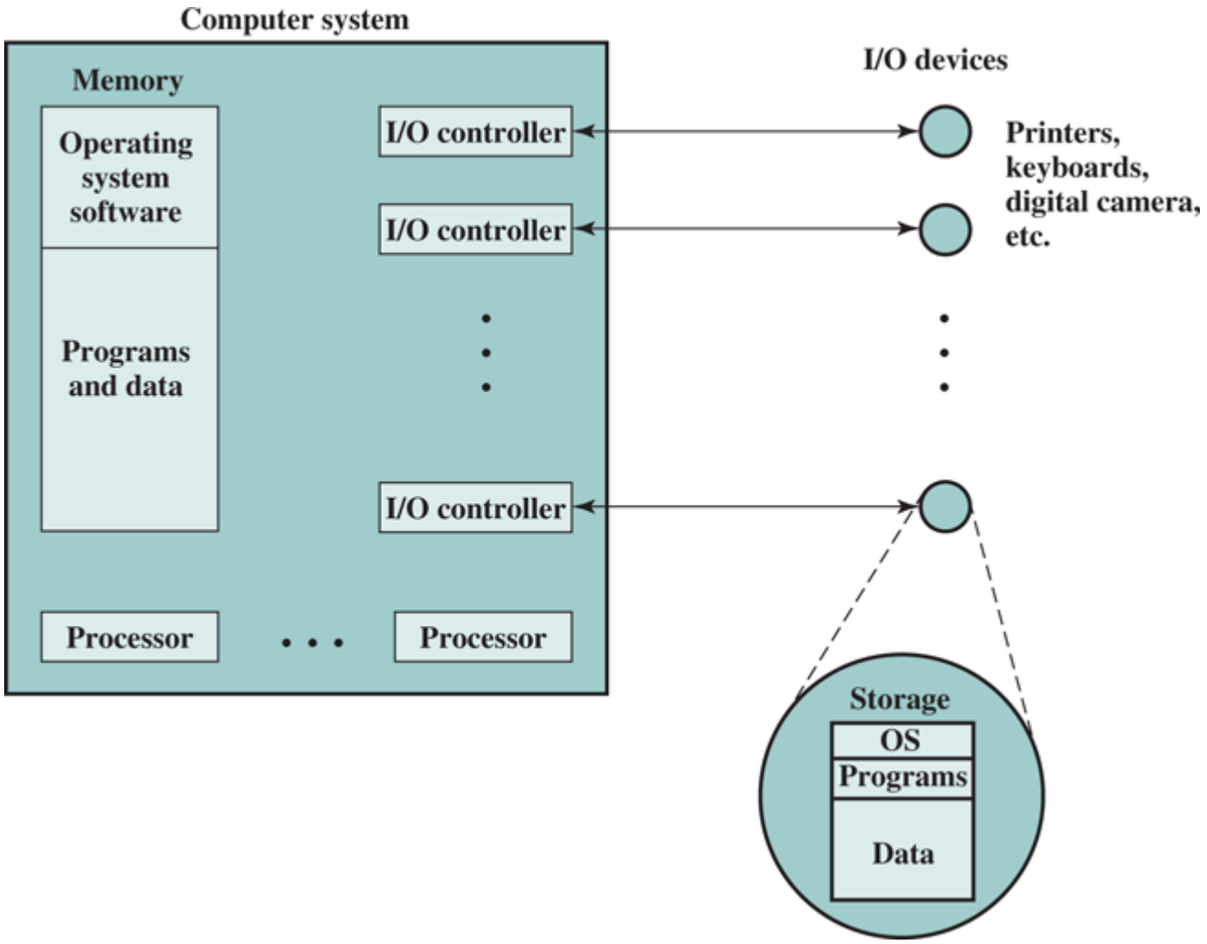


Figure 9.2 The Operating System as Resource Manager

Types of Operating Systems

Certain key characteristics serve to differentiate various types of operating systems. The characteristics fall along two independent dimensions. The first dimension specifies whether the system is batch or interactive. In an **interactive** system, the user/programmer interacts directly with the computer, usually through a keyboard/display terminal, to request the execution of a job or to perform a transaction. Furthermore, the user may, depending on the nature of the application, communicate with the computer during the execution of the job. A **batch system** is the opposite of interactive. The user's program is batched together with programs from other users and submitted by a computer operator. After the program is completed, results are printed out for the user. Pure batch systems are rare today, however, it will be useful to the description of contemporary operating systems to briefly examine batch systems.

An independent dimension specifies whether the system employs **multiprogramming** or not. With multiprogramming, the attempt is made to keep the processor as busy as possible, by having it work on more than one program at a time. Several programs are loaded into memory, and the processor switches rapidly among them. The alternative is a **uniprogramming** system that works only one program at a time.

EARLY SYSTEMS

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These processors were run from a console, consisting

of display lights, toggle switches, some form of input device, and a printer. Programs in processor code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. The programmer could proceed to examine registers and main memory to determine the cause of the error. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems:

- **Scheduling:** Most installations used a sign-up sheet to reserve processor time. Typically, a user could sign up for a block of time in multiples of a half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer idle time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.
- **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program), and then loading and linking together the object program and common functions. Each of these steps could involve mounting or dismounting tapes, or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence. Thus a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

SIMPLE BATCH SYSTEMS

Early processors were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, simple batch operating systems were developed. With such a system, also called a **monitor**, the user no longer has direct access to the processor. Rather, the user submits the job on cards or tape to a computer operator, who *batches* the jobs together sequentially and places the entire batch on an input device, for use by the monitor.

To understand how this scheme works, let us look at it from two points of view: that of the monitor and that of the processor. From the point of view of the monitor, the monitor controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (**Figure 9.3**). That portion is referred to as the **resident monitor**. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are printed out for delivery to the user.

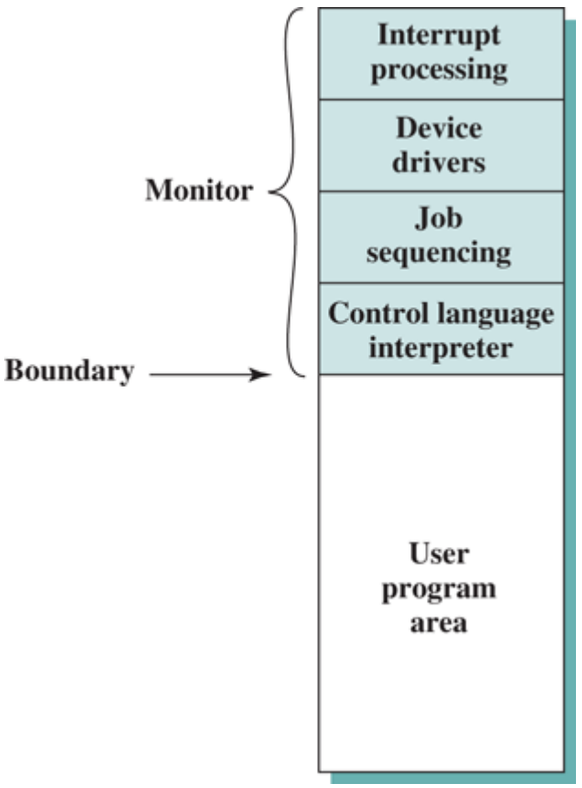


Figure 9.3 Memory Layout for a Resident Monitor

Now consider this sequence from the point of view of the processor. At a certain point in time, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read in to another portion of main memory. Once a job has been read in, the processor will encounter in the monitor a branch instruction that instructs the processor to continue execution at the start of the user program. The processor will then execute the instruction in the user's program until it encounters an ending or error condition. Either event causes the processor to fetch its next instruction from the monitor program. Thus the phrase "control is passed to a job" simply means that the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means that the processor is now fetching and executing instructions from the monitor program.

It should be clear that the monitor handles the scheduling problem. A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time.

How about the job setup time? The monitor handles this as well. With each job, instructions are included in a **job control language (JCL)**. This is a special type of programming language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in FORTRAN plus some data to be used by the program. Each FORTRAN instruction and each item of data is on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning "$". The overall format of the job looks like this:

```
$JOB
$FTN
⋮          } FORTRAN instructions
$LOAD
$RUN
⋮          } Data
$END
```

To execute this job, the monitor reads the $FTN line and loads the appropriate compiler from its mass storage (usually tape). The compiler translates the user's program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is referred to as "compile, load, and go." If it is stored on tape, then the $LOAD instruction is required. This instruction is read by the monitor, which regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory in place of the compiler and transfers control to it. In this manner, a large segment of main memory can be shared among different subsystems, although only one such subsystem could be resident and executing at a time.

We see that the monitor, or batch OS, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main memory in order to seize and relinquish control alternately. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor. The monitor would then abort the job, print out an error message, and load the next job.
- **Timer:** A timer is used to prevent a single job from monopolizing the system. The timer is set at the beginning of each job. If the timer expires, an interrupt occurs, and control returns to the monitor.
- **Privileged instructions:** Certain instructions are designated privileged and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error interrupt occurs. Among the privileged instructions are I/O instructions, so that the monitor retains control of all I/O devices. This prevents, for example, a user program from accidentally reading job control instructions from the next job. If a user program wishes to perform I/O, it must request that the monitor perform the operation for it. If a privileged instruction is encountered by the processor while it is executing a user program, the processor hardware considers this an error and transfers control to the monitor.
- **Interrupts:** Early computer models did not have this capability. This feature gives the OS more flexibility in relinquishing control to and regaining control from user programs.

Processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead. Even with this overhead, the simple batch system improves utilization of the computer.

MULTIPROGRAMMED BATCH SYSTEMS

Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor. **Figure 9.4** details a representative calculation. The calculation concerns a program that processes a file of records and performs, on average, 100 processor instructions per record. In this example the computer spends over 96% of its time waiting for I/O devices to finish transferring data! **Figure 9.5a** illustrates this situation. The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding.

Read one record from file	**15 μs**
Execute 100 instructions	**1 μs**
Write one record to file	**15 μs**
TOTAL	**31 μs**

Percent CPU utilization $= \frac{1}{31} = 0.032 = 3.2\%$

Figure 9.4 System Utilization Example

This inefficiency is not necessary. We know that there must be enough memory to hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two user programs. Now, when one job needs to wait for I/O, the processor can switch to the other job, which likely is not waiting for I/O (**Figure 9.5b**). Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them (**Figure 9.5c**). This technique is known as **multiprogramming**, or **multitasking** .[1] It is the central theme of modern operating systems.

[1] The term *multitasking* is sometimes reserved to mean multiple tasks within the same program that may be handled concurrently by the OS, in contrast to *multiprogramming,* which would refer to multiple processes from multiple programs. However, it is more common to equate the terms *multitasking* and *multiprogramming,* as is done in most standards dictionaries (e.g., IEEE Std 100-1992, *The New IEEE Standard Dictionary of Electrical and Electronics Terms*).

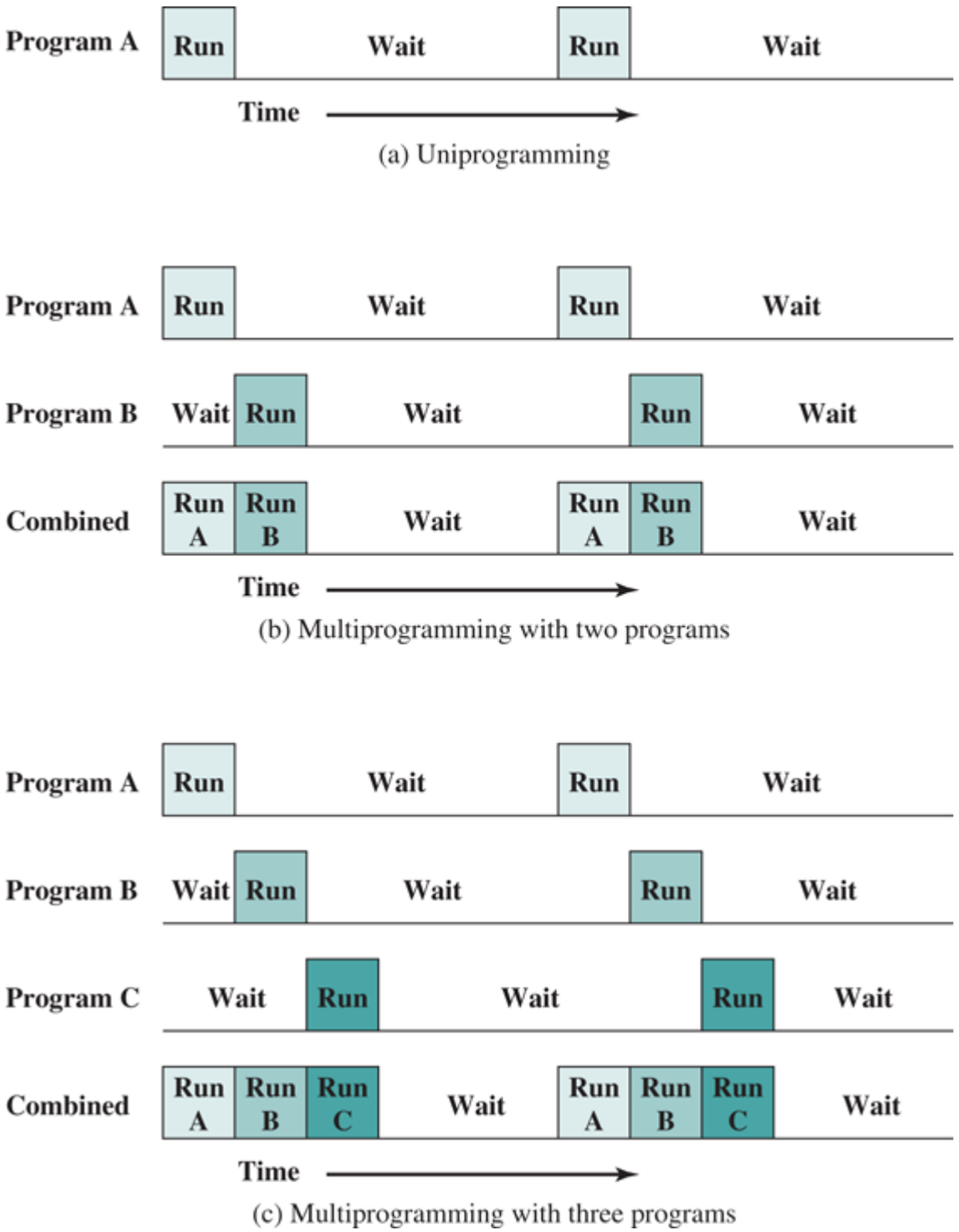


Figure 9.5 Multiprogramming Example

Example 9.1

This example illustrates the benefit of multiprogramming. Consider a computer with 250 Mbytes of available memory (not used by the OS), a disk, a terminal, and a printer. Three programs, JOB1, JOB2, and JOB3, are submitted for execution at the same time, with the attributes listed in **Table 9.1**. We assume minimal processor requirements for JOB1 and JOB2 and continuous disk and printer use by JOB3. For a simple batch environment, these jobs will be executed in sequence. Thus, JOB1 completes in 5 minutes. JOB2 must wait until the 5 minutes is over and then completes 15 minutes after that. JOB3 begins after 20 minutes and completes at 30 minutes from the time it was initially submitted. The average resource utilization, throughput, and response times are shown in the uniprogramming column of **Table 9.2**. Device-by-device utilization is illustrated in **Figure 9.6a**. It is evident that there is gross underutilization for all resources when

averaged over the required 30-minute time period.

Table 9.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration (min)	5	15	10
Memory required (M)	50	100	80
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

Table 9.2 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	Multiprogramming
Processor use (%)	20	40
Memory use (%)	33	67
Disk use (%)	33	67
Printer use (%)	33	67
Elapsed time (min)	30	15
Throughput rate (jobs/hr)	6	12
Mean response time (min)	18	10

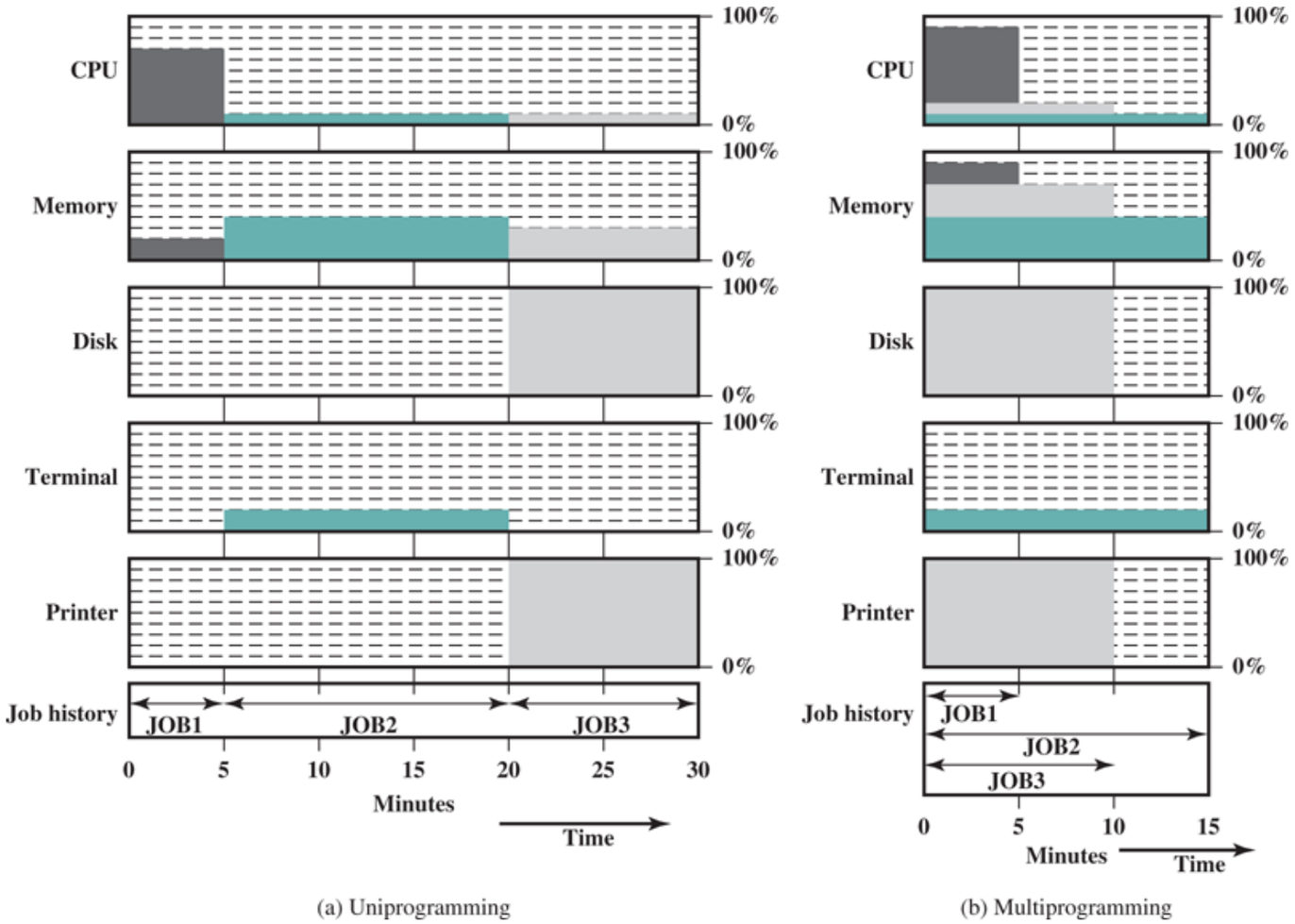


Figure 9.6 Utilization Histograms

Now suppose that the jobs are run concurrently under a multiprogramming OS. Because there is little resource contention between the jobs, all three can run in nearly minimum time while coexisting with the others in the computer (assuming that JOB2 and JOB3 are allotted enough processor time to keep their input and output operations active). JOB1 will still require 5 minutes to complete but at the end of that time, JOB2 will be one-third finished, and JOB3 will be half finished. All three jobs will have finished within 15 minutes. The improvement is evident when examining the multiprogramming column of **Table 9.2**, obtained from the histogram shown in **Figure 9.6b**.

As with a simple batch system, a multiprogramming batch system must rely on certain computer hardware features. The most notable additional feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA. With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller. When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the OS. The OS will then pass control to another job.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or **uniprogramming**, systems. To have several jobs ready to run, the jobs must be kept in main memory, requiring some form of **memory management**. In addition, if several jobs are ready to run, the processor must decide which one to run, which requires some algorithm for scheduling. These concepts are discussed later in this chapter.

TIME-SHARING SYSTEMS

With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated microcomputer. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing was developed.

Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because the processor's time is shared among multiple users. In a **time-sharing system**, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation. Thus, if there are *n* users actively requesting service at one time, each user will only see on the average 1/*n* of the effective computer speed, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be comparable to that on a dedicated computer.

Both batch multiprogramming and time sharing use multiprogramming. The key differences are listed in **Table 9.3**.

Table 9.3 Batch Multiprogramming versus Time Sharing

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

9.2 Scheduling

The key to multiprogramming is scheduling. In fact, four types of scheduling are typically involved (**Table 9.4**). We will explore these presently. But first, we introduce the concept of **process** . This term was first used by the designers of the Multics OS in the 1960s. It is a somewhat more general term than *job*. Many definitions have been given for the term *process*, including

Table 9.4 Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed.
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory.
Short-term scheduling	The decision as to which available process will be executed by the processor.
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device.

- A program in execution
- The "animated spirit" of a program
- That entity to which a processor is assigned

This concept should become clearer as we proceed.

Long-Term Scheduling

The long-term scheduler determines which programs are admitted to the system for processing. Thus, it controls the degree of multiprogramming (number of processes in memory). Once admitted, a job or user program becomes a process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler.

In a batch system, or for the batch portion of a general-purpose OS, newly submitted jobs are routed to disk and held in a batch queue. The long-term scheduler creates processes from the queue when it can. There are two decisions involved here. First, the scheduler must decide that the OS can take on one or more additional processes. Second, the scheduler must decide which job or jobs to accept and turn into processes. The criteria used may include priority, expected execution time, and I/O requirements.

For interactive programs in a time-sharing system, a process request is generated when a user attempts to connect to the system. Time-sharing users are not simply queued up and kept waiting until the system can accept them. Rather, the OS will accept all authorized comers until the system is saturated, using some predefined measure of saturation. At that point, a connection request is met with a message indicating that the system is full and the user should try again later.

Medium-Term Scheduling

Medium-term scheduling is part of the swapping function, described in **Section 9.3**. Typically, the swapping-in decision is based on the need to manage the degree of multiprogramming. On a system that does not use virtual memory, memory management is also an issue. Thus, the swapping-in decision will consider the memory requirements of the swapped-out processes.

Short-Term Scheduling

The long-term scheduler executes relatively infrequently and makes the coarse-grained decision of whether or not to take on a new process, and which one to take. The short-term scheduler, also known as the **dispatcher**, executes frequently and makes the fine-grained decision of which job to execute next.

PROCESS STATES

To understand the operation of the short-term scheduler, we need to consider the concept of a **process state**. During the lifetime of a process, its status will change a number of times. Its status at any point in time is referred to as a *state.* The term *state* is used because it connotes that certain information exists that defines the status at that point. At minimum, there are five defined states for a process (**Figure 9.7**):

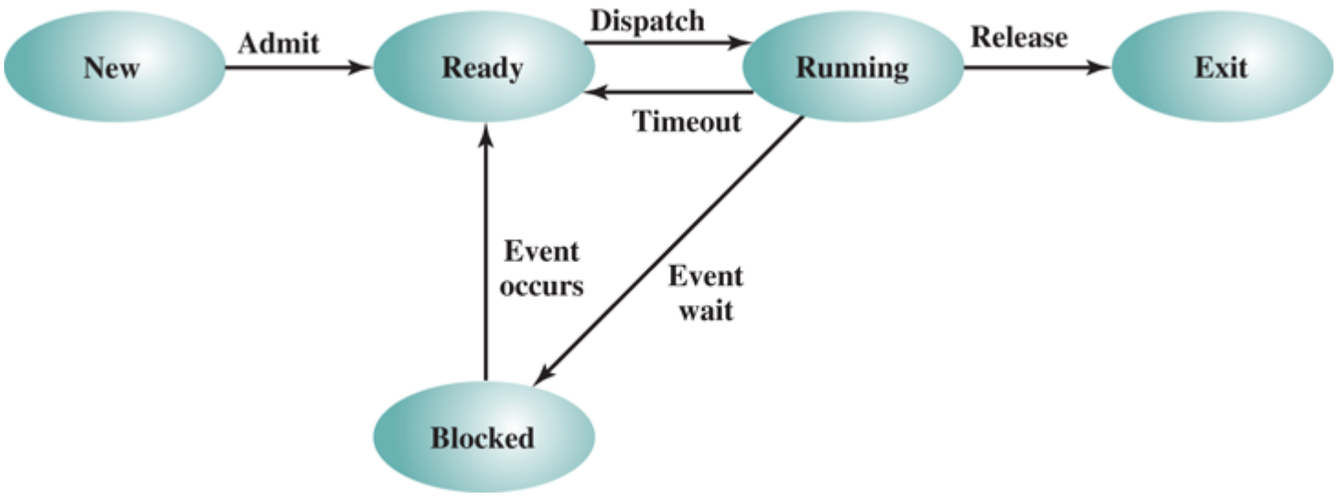


Figure 9.7 Five-State Process Model

- **New:** A program is admitted by the high-level scheduler but is not yet ready to execute. The OS will initialize the process, moving it to the ready state.
- **Ready:** The process is ready to execute and is awaiting access to the processor.
- **Running:** The process is being executed by the processor.
- **Waiting:** The process is suspended from execution waiting for some system resource, such as I/O.
- **Halted:** The process has terminated and will be destroyed by the OS.

For each process in the system, the OS must maintain information indicating the state of the process and other information necessary for process execution. For this purpose, each process is represented in the OS by a **process control block** (**Figure 9.8**), which typically contains:

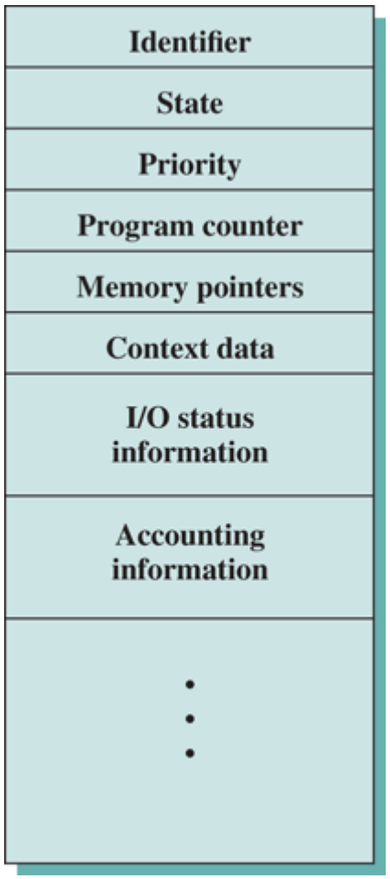


Figure 9.8 Process Control Block

- **Identifier:** Each current process has a unique identifier.
- **State:** The current state of the process (new, ready, and so on).
- **Priority:** Relative priority level.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** The starting and ending locations of the process in memory.
- **Context data:** These are data that are present in registers in the processor while the process is executing, and they will be discussed in Part Three. For now, it is enough to say that these data represent the "context" of the process. The context data plus the program counter are saved when the process leaves the running state. They are retrieved by the processor when it resumes execution of the process.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files assigned to the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

When the scheduler accepts a new job or user request for execution, it creates a blank process control block and places the associated process in the new state. After the system has properly filled in the process control block, the process is transferred to the ready state.

SCHEDULING TECHNIQUES

To understand how the OS manages the scheduling of the various jobs in memory, let us begin by considering the simple example in **Figure 9.9**. The figure shows how main memory is partitioned at a given point in time. The kernel of the OS is, of course, always resident. In addition, there are a number of active processes, including **A** and **B**, each of which is allocated a portion of memory.

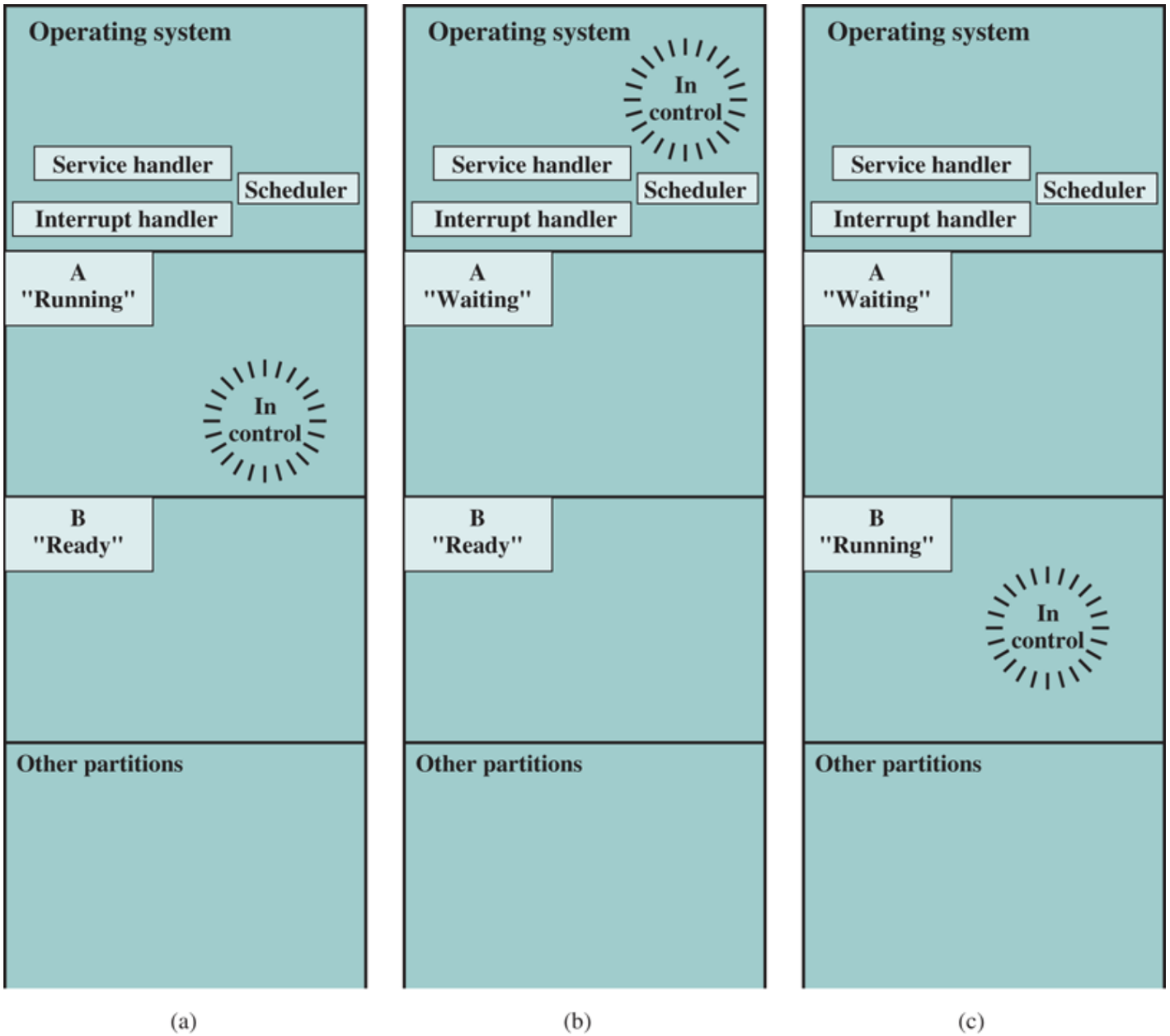


Figure 9.9 Scheduling Example

We begin at a point in time when process **A** is running. The processor is executing instructions from the program contained in **A**'s memory partition. At some later point in time, the processor ceases to execute instructions in **A** and begins executing instructions in the OS area. This will happen for one of three reasons:

1. Process **A** issues a service call (e.g., an I/O request) to the OS. Execution of **A** is suspended until this call is satisfied by the OS.
2. Process **A** causes an *interrupt.* An interrupt is a hardware-generated signal to the processor. When this signal is detected, the processor ceases to execute **A** and transfers to the interrupt handler in the OS. A variety of events related to **A** will cause an interrupt. One example is an error, such as attempting to execute a privileged instruction. Another example is a timeout; to prevent any one process from monopolizing the processor, each process is only granted the processor for a short period at a time.
3. Some event unrelated to process **A** that requires attention causes an interrupt. An example is the completion of an I/O operation.

In any case, the result is the following. The processor saves the current context data and the program counter for **A** in **A**'s process control block and then begins executing in the OS. The OS may perform some work, such as initiating an I/O operation. Then the short-term-scheduler portion of the OS decides which process should be executed next. In this example, **B** is chosen. The OS instructs the processor to restore **B**'s context data and proceed with the execution of **B** where it left off.

This simple example highlights the basic functioning of the short-term scheduler. **Figure 9.10** shows the major elements of the OS involved in the multiprogramming and scheduling of processes. The OS receives control of the processor at the interrupt handler if an interrupt occurs and at the service-call handler if a service call occurs. Once the interrupt or service call is handled, the short-term scheduler is invoked to select a process for execution.

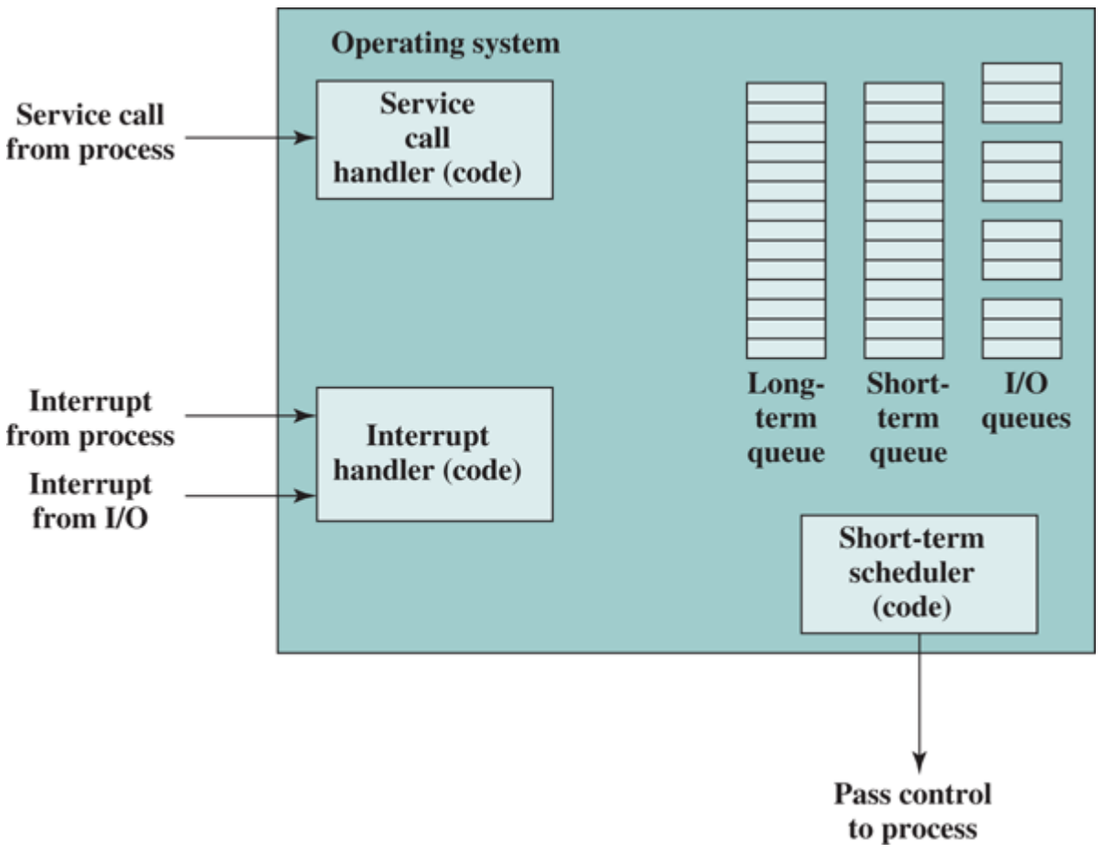


Figure 9.10 Key Elements of an Operating System for Multiprogramming

To do its job, the OS maintains a number of queues. Each queue is simply a waiting list of processes waiting for some resource. The **long-term queue** is a list of jobs waiting to use the system. As conditions permit, the high-level scheduler will allocate memory and create a process for one of the waiting items. The **short-term queue** consists of all processes in the ready state. Any one of these processes could use the processor next. It is up to the short-term scheduler to pick one. Generally, this is done with a round-robin algorithm, giving each process some time in turn. Priority levels may also be used. Finally, there is an **I/O queue** for each I/O device. More than one process may request the use of the same I/O device. All processes waiting to use each device are lined up in that device's queue.

Figure 9.11 suggests how processes progress through the computer under the control of the OS. Each process request (batch job, user-defined interactive job) is placed in the long-term queue. As resources become available, a process request becomes a process and is then placed in the ready

state and put in the short-term queue. The processor alternates between executing OS instructions and executing user processes. While the OS is in control, it decides which process in the short-term queue should be executed next. When the OS has finished its immediate tasks, it turns the processor over to the chosen process.

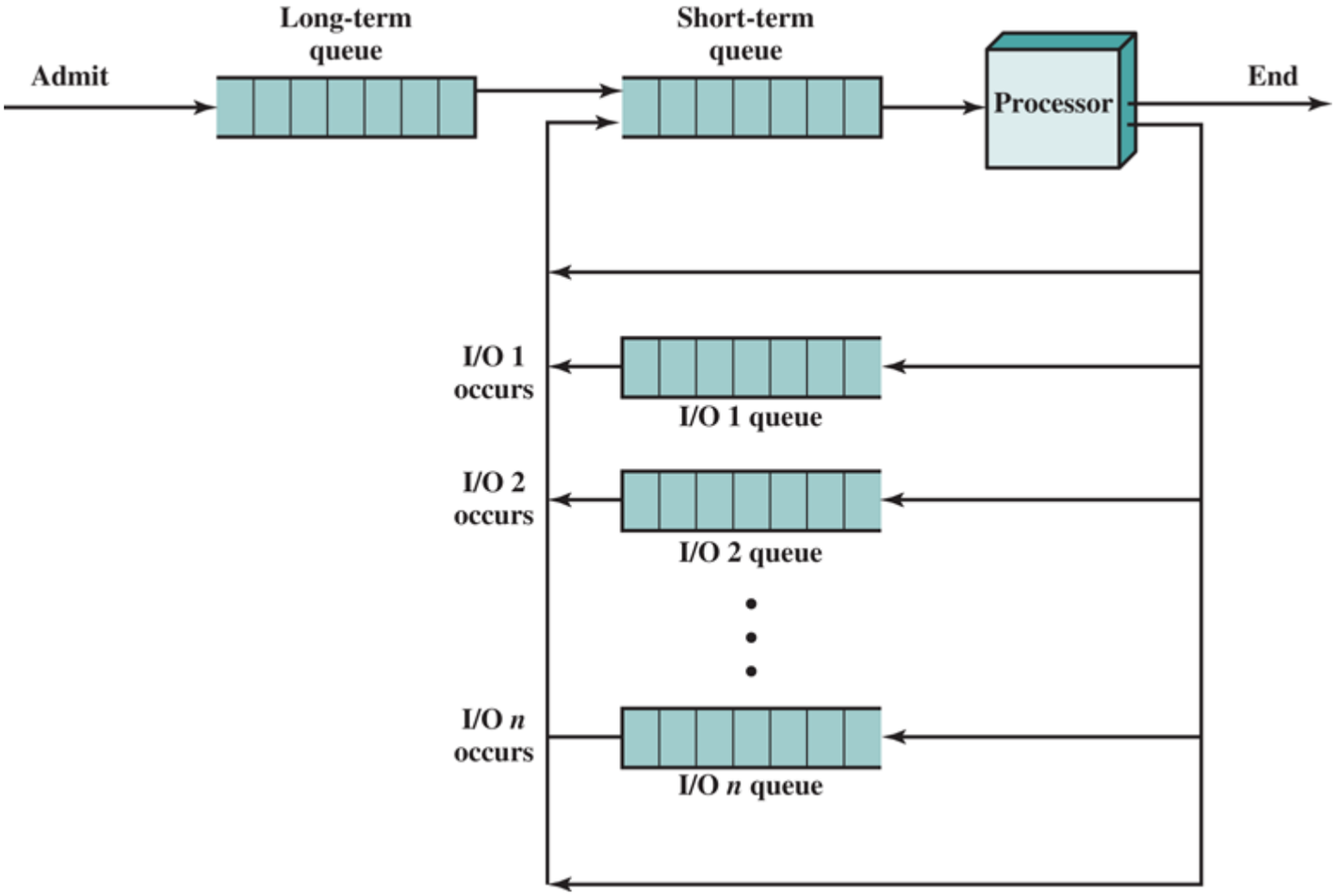


Figure 9.11 Queuing Diagram Representation of Processor Scheduling

As was mentioned earlier, a process being executed may be suspended for a variety of reasons. If it is suspended because the process requests I/O, then it is placed in the appropriate I/O queue. If it is suspended because of a timeout or because the OS must attend to pressing business, then it is placed in the ready state and put into the short-term queue.

Finally, we mention that the OS also manages the I/O queues. When an I/O operation is completed, the OS removes the satisfied process from that I/O queue and places it in the short-term queue. It then selects another waiting process (if any) and signals for the I/O device to satisfy that process's request.

9.3 Memory Management

In a uniprogramming system, main memory is divided into two parts: one part for the OS (resident monitor) and one part for the program currently being executed. In a multiprogramming system, the "user" part of memory is subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the OS and is known as **memory management**.

Effective memory management is vital in a multiprogramming system. If only a few processes are in memory, then for much of the time all of the processes will be waiting for I/O and the processor will be idle. Thus, memory needs to be allocated efficiently to pack as many processes into memory as possible.

Swapping

Referring back to **Figure 9.11**, we have discussed three types of queues: the long-term queue of requests for new processes, the short-term queue of processes ready to use the processor, and the various I/O queues of processes that are not ready to use the processor. Recall that the reason for this elaborate machinery is that I/O activities are much slower than computation and therefore the processor in a uniprogramming system is idle most of the time.

But the arrangement in **Figure 9.11** does not entirely solve the problem. It is true that, in this case, memory holds multiple processes and that the processor can move to another process when one process is waiting. But the processor is so much faster than I/O that it will be common for *all* the processes in memory to be waiting on I/O. Thus, even with multiprogramming, a processor could be idle most of the time.

What to do? Main memory could be expanded, and so be able to accommodate more processes. But there are two flaws in this approach. First, main memory is expensive, even today. Second, the appetite of programs for memory has grown as fast as the cost of memory has dropped. So larger memory results in larger processes, not more processes.

Another solution is **swapping**, depicted in **Figure 9.12**. We have a long-term queue of process requests, typically stored on disk. These are brought in, one at a time, as space becomes available. As processes are completed, they are moved out of main memory. Now the situation will arise that none of the processes in memory are in the ready state (e.g., all are waiting on an I/O operation). Rather than remain idle, the processor *swaps* one of these processes back out to disk into an *intermediate queue.* This is a queue of existing processes that have been temporarily kicked out of memory. The OS then brings in another process from the intermediate queue, or it honors a new process request from the long-term queue. Execution then continues with the newly arrived process.

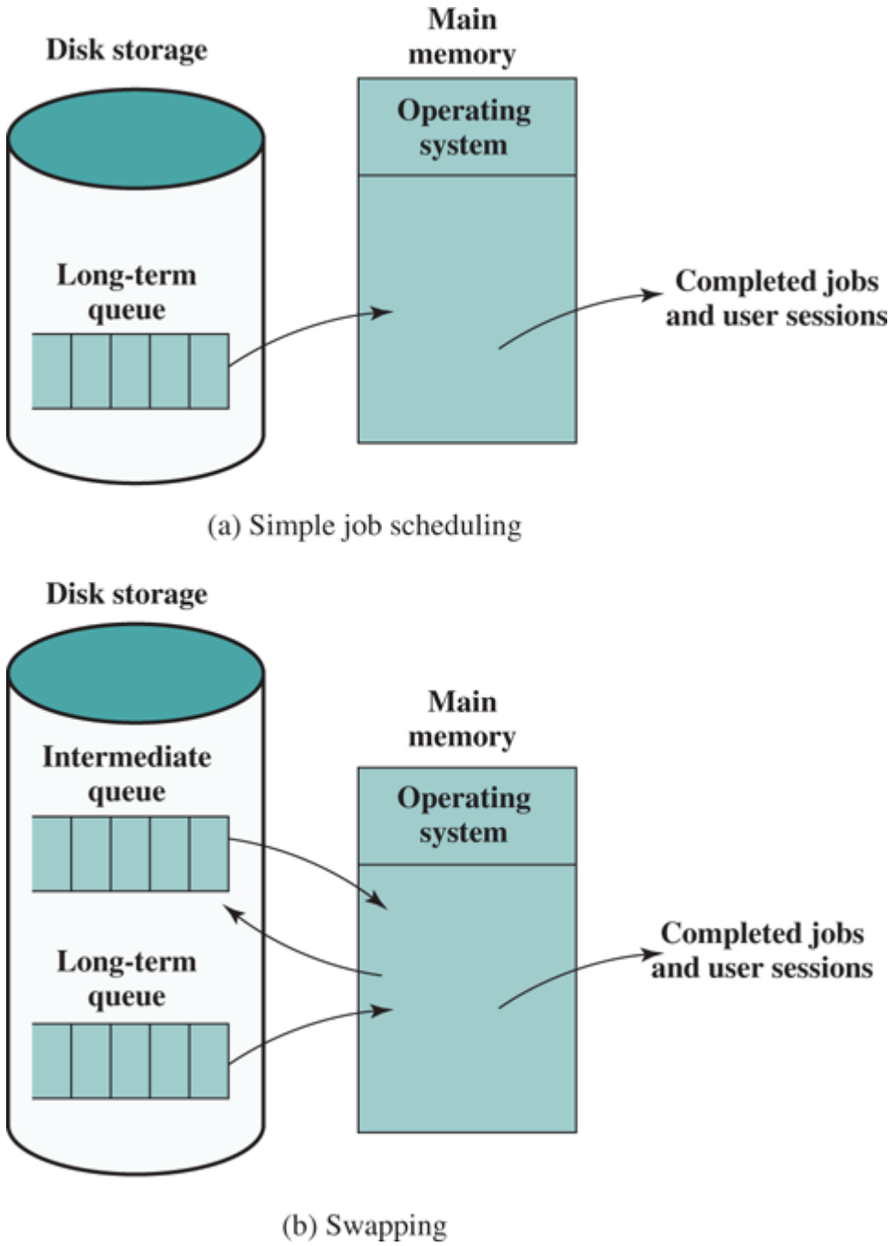
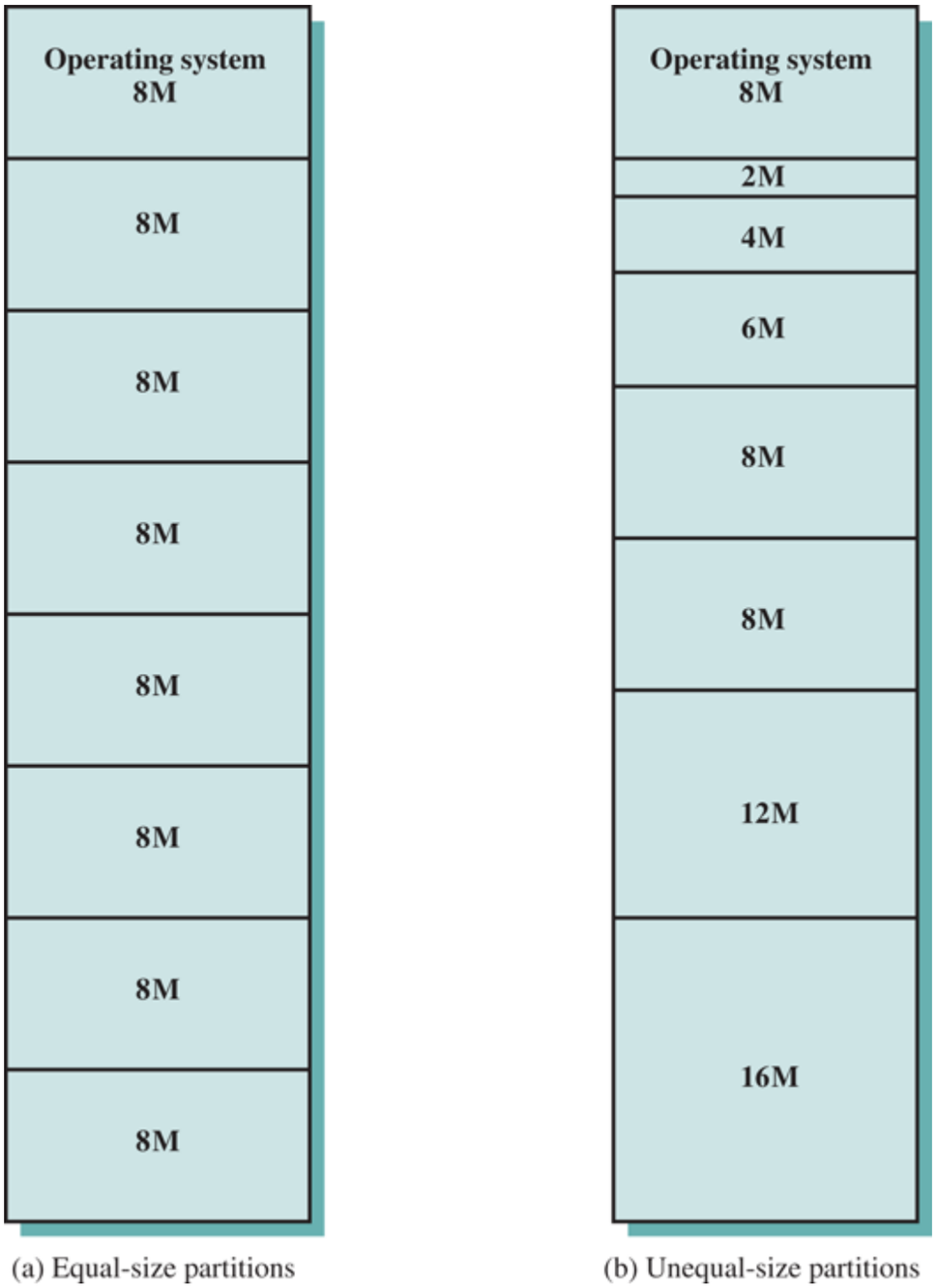


Figure 9.12 The Use of Swapping

Swapping, however, is an I/O operation, and therefore there is the potential for making the problem worse, not better. But because disk I/O is generally the fastest I/O on a system (e.g., compared with tape or printer I/O), swapping will usually enhance performance. A more sophisticated scheme, involving virtual memory, improves performance over simple swapping. This will be discussed shortly. But first, we must prepare the ground by explaining partitioning and paging.

Partitioning

The simplest scheme for partitioning available memory is to use *fixed-size partitions,* as shown in **Figure 9.13**. Note that, although the partitions are of fixed size, they need not be of equal size. When a process is brought into memory, it is placed in the smallest available partition that will hold it.



(a) Equal-size partitions (b) Unequal-size partitions

Figure 9.13 Example of Fixed Partitioning of a 64-Mbyte Memory

Even with the use of unequal fixed-size partitions, there will be wasted memory. In most cases, a process will not require exactly as much memory as provided by the partition. For example, a process that requires 3M bytes of memory would be placed in the 4M partition of **Figure 9.13b**, wasting 1M that could be used by another process.

A more efficient approach is to use *variable-size partitions.* When a process is brought into memory, it is allocated exactly as much memory as it requires and no more.

Example 9.2

An example, using 64 Mbytes of main memory, is shown in **Figure 9.14**. Initially, main memory is empty, except for the OS (a). The first three processes are loaded in, starting where the OS ends and occupying just enough space for each process (b, c, d). This leaves a “hole” at the end of

memory that is too small for a fourth process. At some point, none of the processes in memory is ready. The OS swaps out process 2 (e), which leaves sufficient room to load a new process, process 4 (f). Because process 4 is smaller than process 2, another small hole is created. Later, a point is reached at which none of the processes in main memory is ready, but process 2, in the ready-suspend state, is available. Because there is insufficient room in memory for process 2, the OS swaps process 1 out (g) and swaps process 2 back in (h).

As this example shows, this method starts out well, but eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on, memory becomes more and more fragmented, and memory utilization declines. One technique for overcoming this problem is **compaction**: From time to time, the OS shifts the processes in memory to place all the free memory together in one block. This is a time-consuming procedure, wasteful of processor time.

Before we consider ways of dealing with the shortcomings of partitioning, we must clear up one loose end. Consider **Figure 9.14**; it should be obvious that a process is not likely to be loaded into the same place in main memory each time it is swapped in. Furthermore, if compaction is used, a process may be shifted while in main memory. A process in memory consists of instructions plus data. The instructions will contain addresses for memory locations of two types:

- Addresses of data items
- Addresses of instructions, used for branching instructions

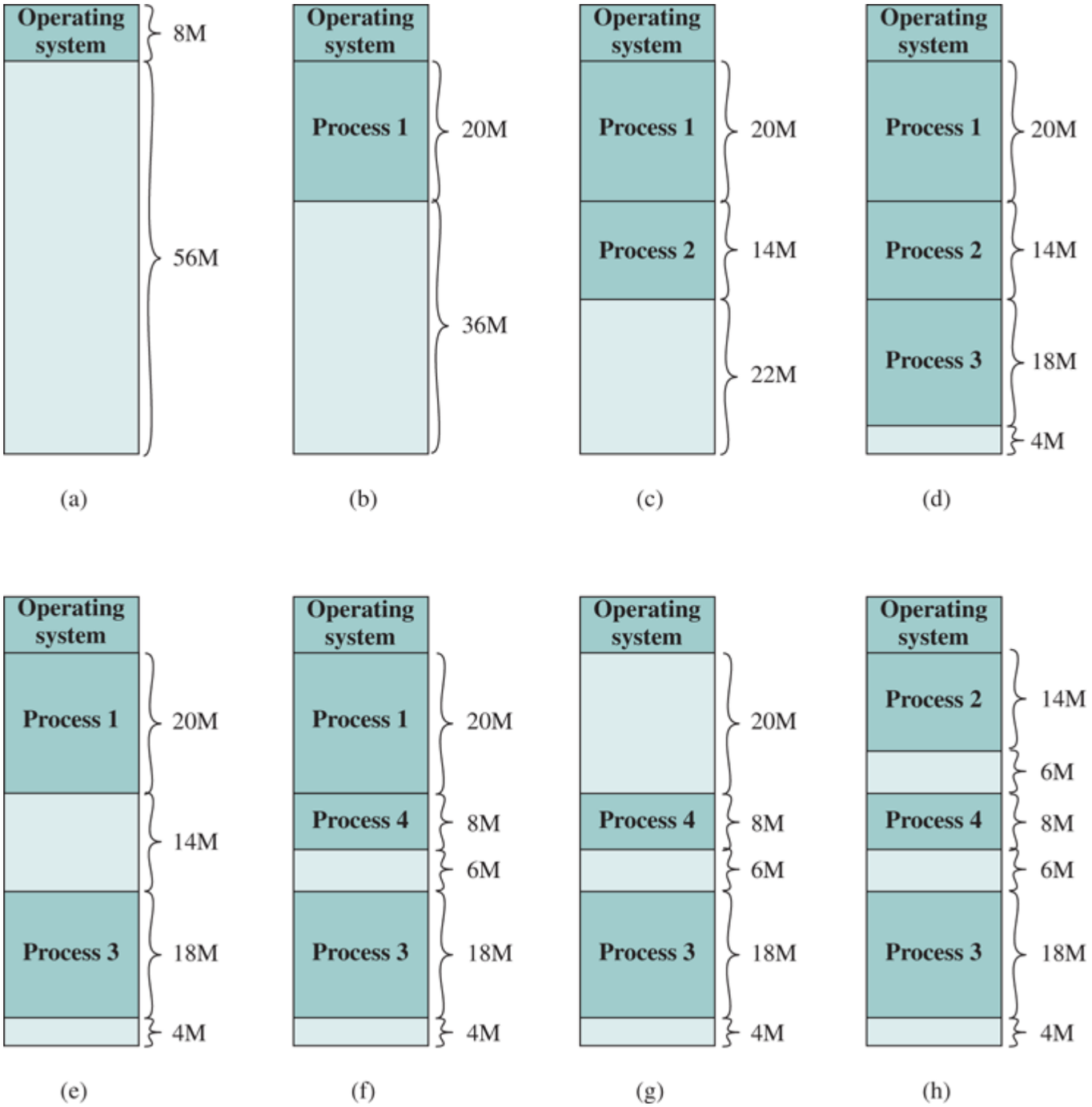


Figure 9.14 The Effect of Dynamic Partitioning

But these addresses are not fixed. They will change each time a process is swapped in. To solve this problem, a distinction is made between logical addresses and physical addresses. A **logical address** is expressed as a location relative to the beginning of the program. Instructions in the program contain only logical addresses. A **physical address** is an actual location in main memory. When the processor executes a process, it automatically converts from logical to physical address by adding the current starting location of the process, called its **base address** , to each logical address. This is another example of a processor hardware feature designed to meet an OS requirement. The exact nature of this hardware feature depends on the memory management strategy in use. We will see several examples later in this chapter.

Paging

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory. Suppose, however, that memory is partitioned into equal fixed-size chunks that are relatively small, and that each process is also divided into small fixed-size chunks of some size. Then the chunks of a program, known as **pages** , could be assigned to available chunks of memory, known as **frames**, or page frames. At most, then, the wasted space in memory for that process is a fraction of the last page.

Figure 9.15 shows an example of the use of pages and frames. At a given point in time, some of the frames in memory are in use and some are free. The list of free frames is maintained by the OS. Process A, stored on disk, consists of four pages. When it comes time to load this process, the OS finds four free frames and loads the four pages of process **A** into the four frames.

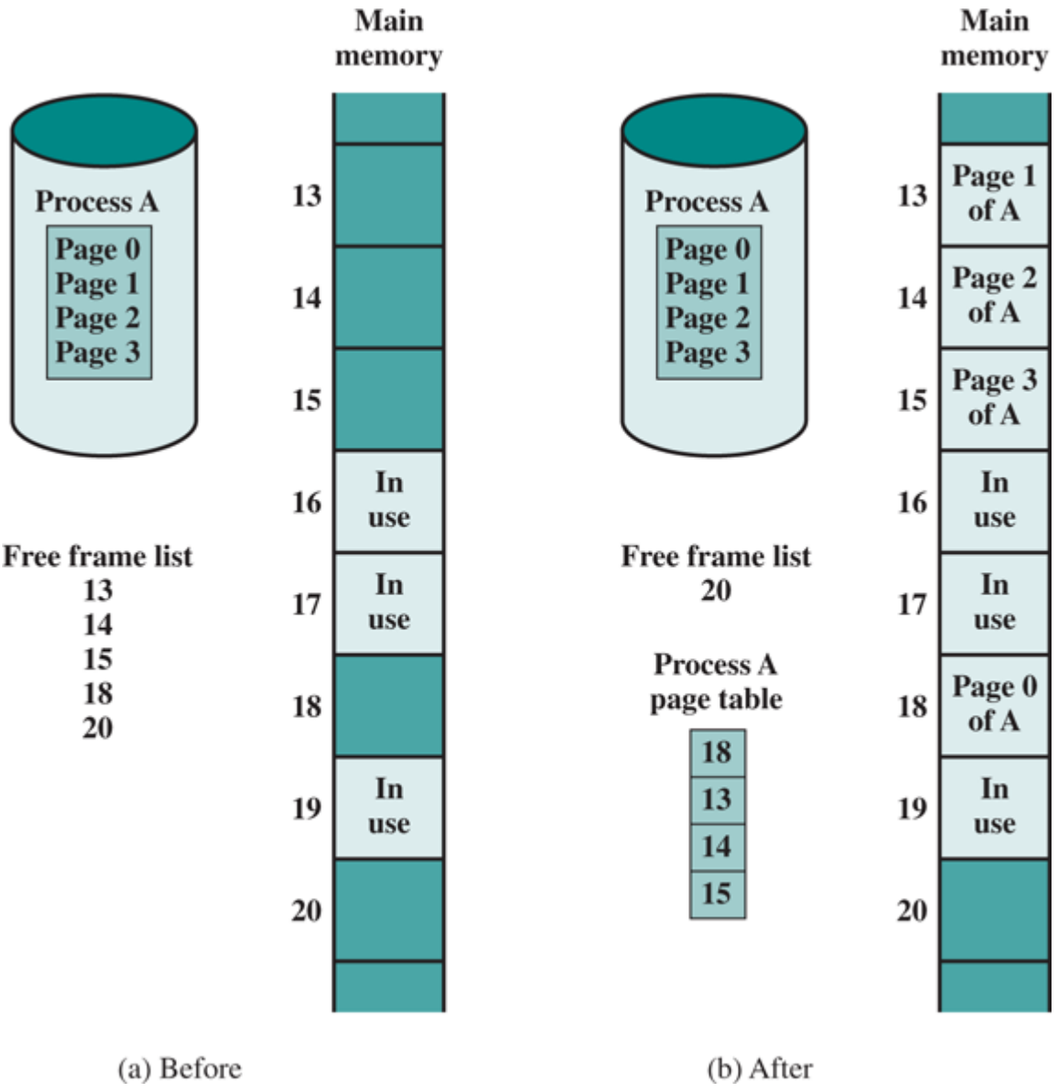


Figure 9.15 Allocation of Free Frames

Now suppose, as in this example, that there are not sufficient unused contiguous frames to hold the process. Does this prevent the OS from loading **A**? The answer is no, because we can once again use the concept of logical address. A simple base address will no longer suffice. Rather, the OS maintains a **page table** for each process. The page table shows the frame location for each page of the process. Within the program, each logical address consists of a page number and a relative address within the page. Recall that in the case of simple partitioning, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address. With paging, the logical-to-physical address translation is still done by processor hardware.

The processor must know how to access the page table of the current process. Presented with a logical address (page number, relative address), the processor uses the page table to produce a physical address (frame number, relative address). An example is shown in **Figure 9.16**.

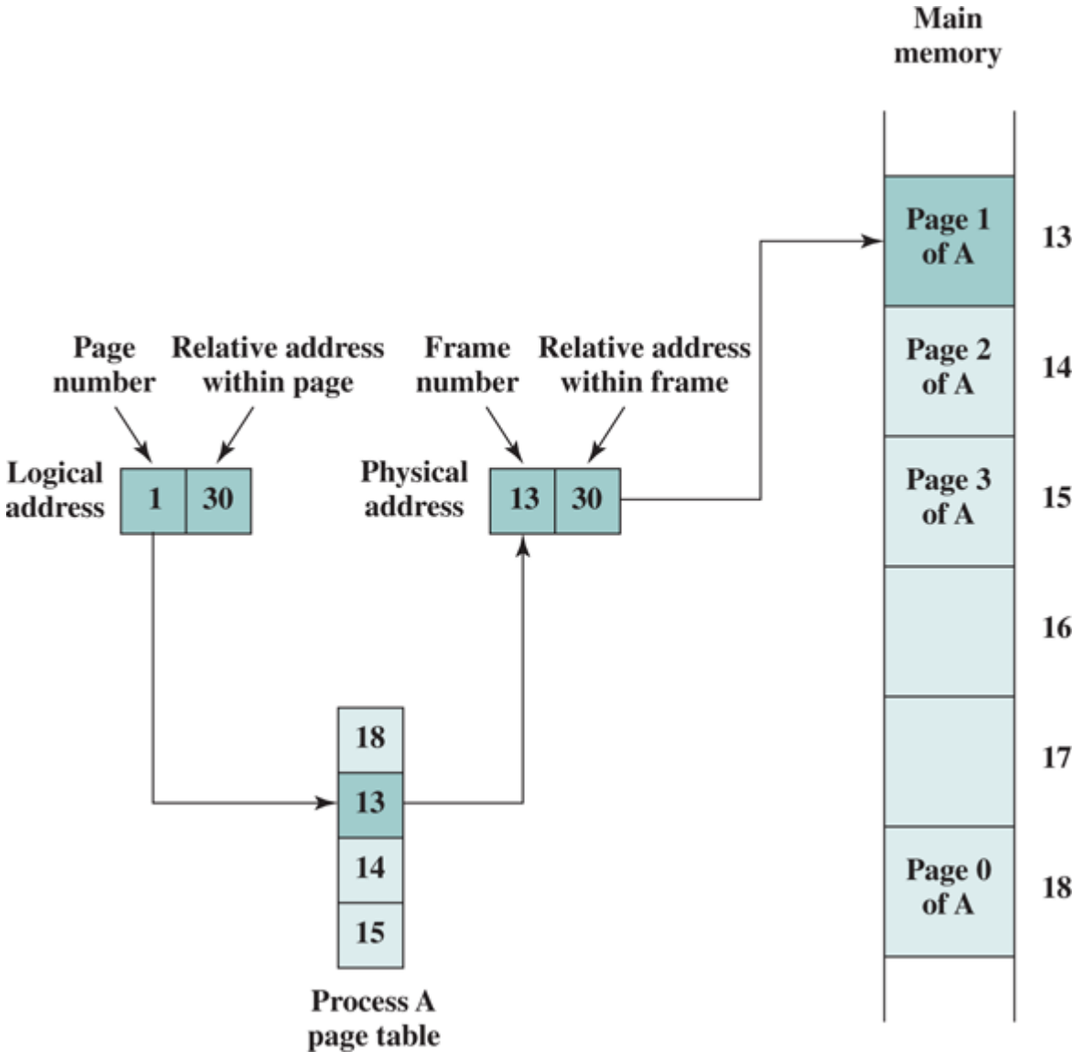


Figure 9.16 Logical and Physical Addresses

This approach solves the problems raised earlier. Main memory is divided into many small equal-size frames. Each process is divided into frame-size pages: smaller processes require fewer pages, larger processes require more. When a process is brought in, its pages are loaded into available frames, and a page table is set up.

Virtual Memory

DEMAND PAGING

With the use of paging, truly effective multiprogramming systems came into being. Furthermore, the simple tactic of breaking a process up into pages led to the development of another important concept: virtual memory.

To understand virtual memory, we must add a refinement to the paging scheme just discussed. That refinement is **demand paging** , which simply means that each page of a process is brought in only when it is needed, that is, on demand.

Consider a large process, consisting of a long program plus a number of arrays of data. Over any short period of time, execution may be confined to a small section of the program (e.g., a subroutine), and perhaps only one or two arrays of data are being used. This is the principle of locality, which we introduced in **Chapter 4**. It would clearly be wasteful to load in dozens of pages for that process when only a few pages will be used before the program is suspended. We can make better use of memory by loading in just a few pages. Then, if the program branches to an instruction on a page not in main memory, or if the program references data on a page not in memory, a **page fault** is triggered. This tells the OS to bring in the desired page.

Thus, at any one time, only a few pages of any given process are in memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory. However, the OS must be clever about how it manages this scheme. When it brings one page in, it must throw another page out; this is known as **page replacement**. If it throws out a page just before it is about to be used, then it will just have to go get that page again almost immediately. Too much of this leads to a condition known as **thrashing**: the processor spends most of its time swapping pages rather than executing instructions. The avoidance of thrashing was a major research area in the 1970s and led to a variety of complex but effective algorithms. In essence, the OS tries to guess, based on recent history, which pages are least likely to be used in the near future.

Aleksandr Lukin/123RF

Page Replacement Algorithm Simulators

A discussion of page replacement algorithms is beyond the scope of this chapter. A potentially effective technique is least recently used (LRU), the same algorithm discussed in **Chapter 4** for cache replacement. In practice, LRU is difficult to implement for a virtual memory paging scheme. Several alternative approaches that seek to approximate the performance of LRU are in use.

With demand paging, it is not necessary to load an entire process into main memory. This fact has a remarkable consequence: *It is possible for a process to be larger than all of main memory.* One of the most fundamental restrictions in programming has been lifted. Without demand paging, a programmer must be acutely aware of how much memory is available. If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be loaded one at a time. With demand paging, that job is left to the OS and the hardware. As far as the programmer is concerned, he or she is dealing with a huge memory, the size associated with disk storage.

Because a process executes only in main memory, that memory is referred to as **real memory**. But a programmer or user perceives a much larger memory—that which is allocated on the disk. This latter is therefore referred to as **virtual memory**. Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory.

PAGE TABLE STRUCTURE

The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table. Because the page table is of variable length, depending on the size of

the process, we cannot expect to hold it in registers. Instead, it must be in main memory to be accessed. **Figure 9.16** suggests a hardware implementation of this scheme. When a particular process is running, a register holds the starting address of the page table for that process. The page number of a virtual address is used to index that table and look up the corresponding frame number. This is combined with the offset portion of the virtual address to produce the desired real address.

In most systems, there is one page table per process. But each process can occupy huge amounts of virtual memory. For example, in the VAX architecture, each process can have up to $2^{31} = 2$ Gbytes of virtual memory. Using $2^9 = 512 - \text{byte pages}$, that means that as many as 2^{22} page table entries are required *per process.* Clearly, the amount of memory devoted to page tables alone could be unacceptably high. To overcome this problem, most virtual memory schemes store page tables in virtual memory rather than real memory. This means that page tables are subject to paging just as other pages are. When a process is running, at least a part of its page table must be in main memory, including the page table entry of the currently executing page. Some processors make use of a two-level scheme to organize large page tables. In this scheme, there is a page directory, in which each entry points to a page table. Thus, if the length of the page directory is *X*, and if the maximum length of a page table is *Y*, then a process can consist of up to $X \times Y$ pages. Typically, the maximum length of a page table is restricted to be equal to one page. We will see an example of this two-level approach when we consider the Intel x86 later in this chapter.

An alternative approach to the use of one- or two-level page tables is the use of an inverted page table structure (**Figure 9.17**). Variations on this approach are used on the PowerPC, UltraSPARC, and the IA-64 architecture. An implementation of the Mach OS on the RT-PC also uses this technique.

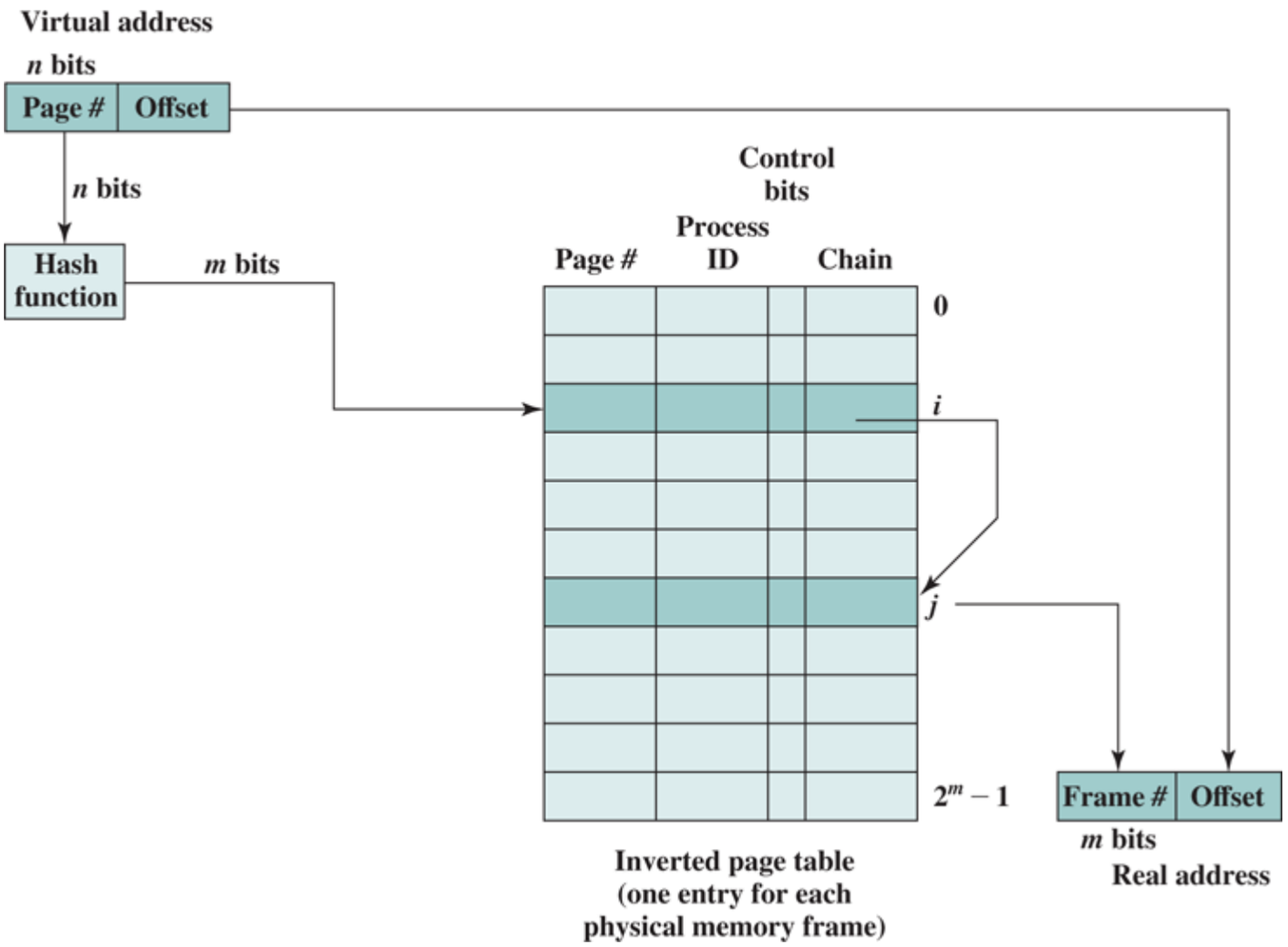


Figure 9.17 Inverted Page Table Structure

In this approach, the page number portion of a virtual address is mapped into a hash value using a simple hashing function.[2] The hash value is a pointer to the inverted page table, which contains the page table entries. There is one entry in the inverted page table for each real memory page frame, rather than one per virtual page. Thus a fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported. Because more than one virtual address may map into the same hash table entry, a chaining technique is used for managing the overflow. The hashing technique results in chains that are typically short—between one and two entries. The page table's structure is called *inverted* because it indexes page table entries by frame number rather than by virtual page number.

[2] A hash function maps numbers in the range 0 through *M* into numbers in the range 0 through *N*, where $M > N$. The output of the hash function is used as an index into the hash table. Since more than one input maps into the same output, it is possible for an input item to map to a hash table entry that is already occupied. In that case, the new item must *overflow* into another hash table location. Typically, the new item is placed in the first succeeding empty space, and a pointer from the original location is provided to chain the entries together.

Translation Lookaside Buffer

In principle, then, every virtual memory reference can cause two physical memory accesses: one to fetch the appropriate page table entry, and one to fetch the desired data. Thus, a straightforward virtual memory scheme would have the effect of doubling the memory access time. To overcome this problem, most virtual memory schemes make use of a special cache for page table entries, usually called a **translation lookaside buffer (TLB)**. This cache functions in the same way as a memory cache and contains those page table entries that have been most recently used. **Figure 9.18** is a flowchart that shows the use of the TLB. By the principle of locality, most virtual memory references will be to locations in recently used pages. Therefore, most references will involve page table entries in the cache. Numerous studies have shown that this scheme can significantly improve performance [MITT17b].

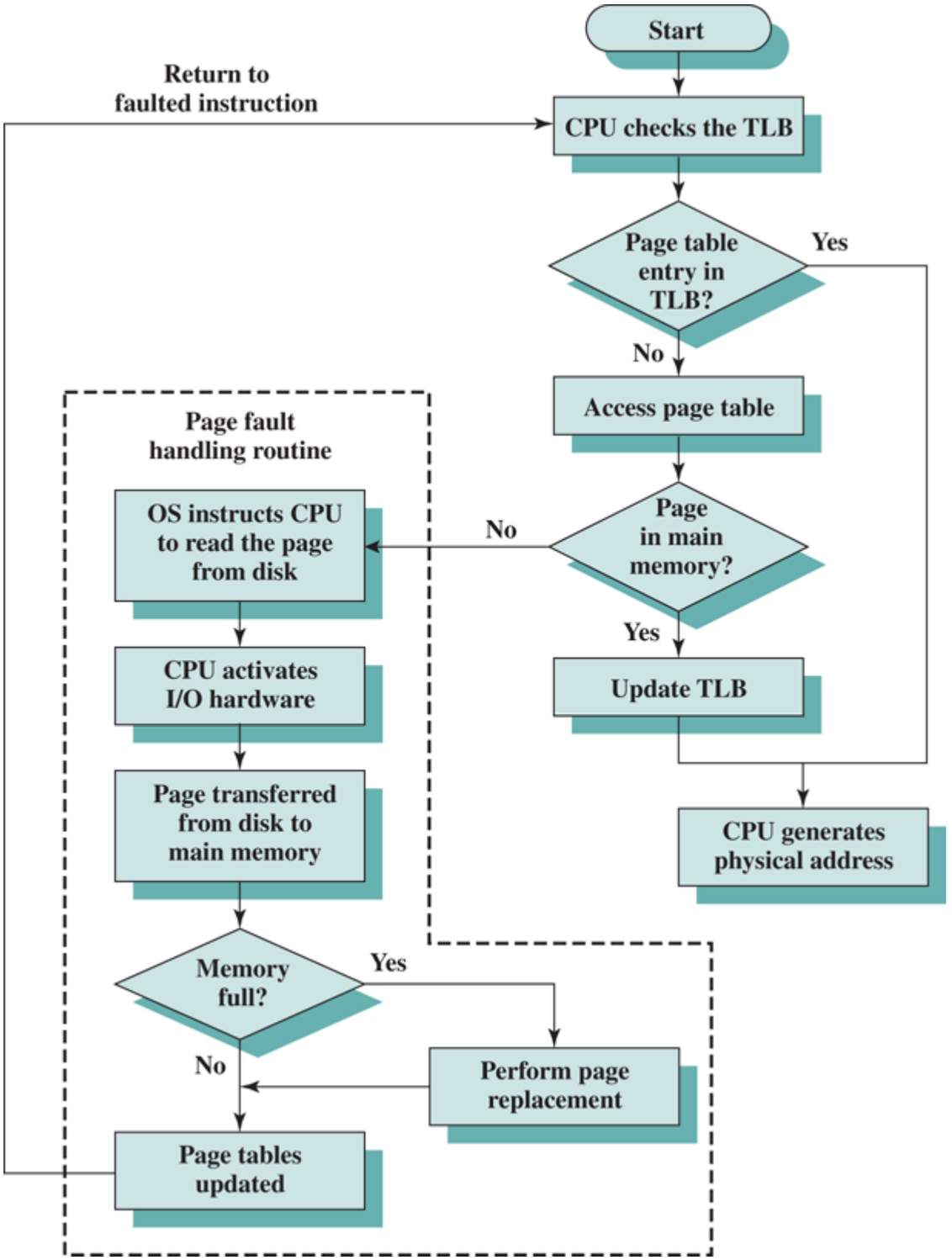


Figure 9.18 Operation of Paging and Translation Lookaside Buffer (TLB)

Note that the virtual memory mechanism must interact with the cache system (not the TLB cache, but the main memory cache). This is illustrated in **Figure 9.19**. A virtual address will generally be in the form of a page number, offset. First, the memory system consults the TLB to see if the matching page table entry is present. If it is, the real (physical) address is generated by combining the frame number with the offset. If not, the entry is accessed from a page table. Once the real address is generated, which is in the form of a tag and a remainder, the cache is consulted to see if the block containing that word is present (see **Figure 5.3**). If so, it is returned to the processor. If not, the word is retrieved from main memory. The TLB is sometimes implemented as content-addressable memory (CAM). The CAM

search key is the virtual address and the search result is a physical address. If the requested address is present in the TLB, the CAM search yields a match quickly and the retrieved physical address can be used to access memory.

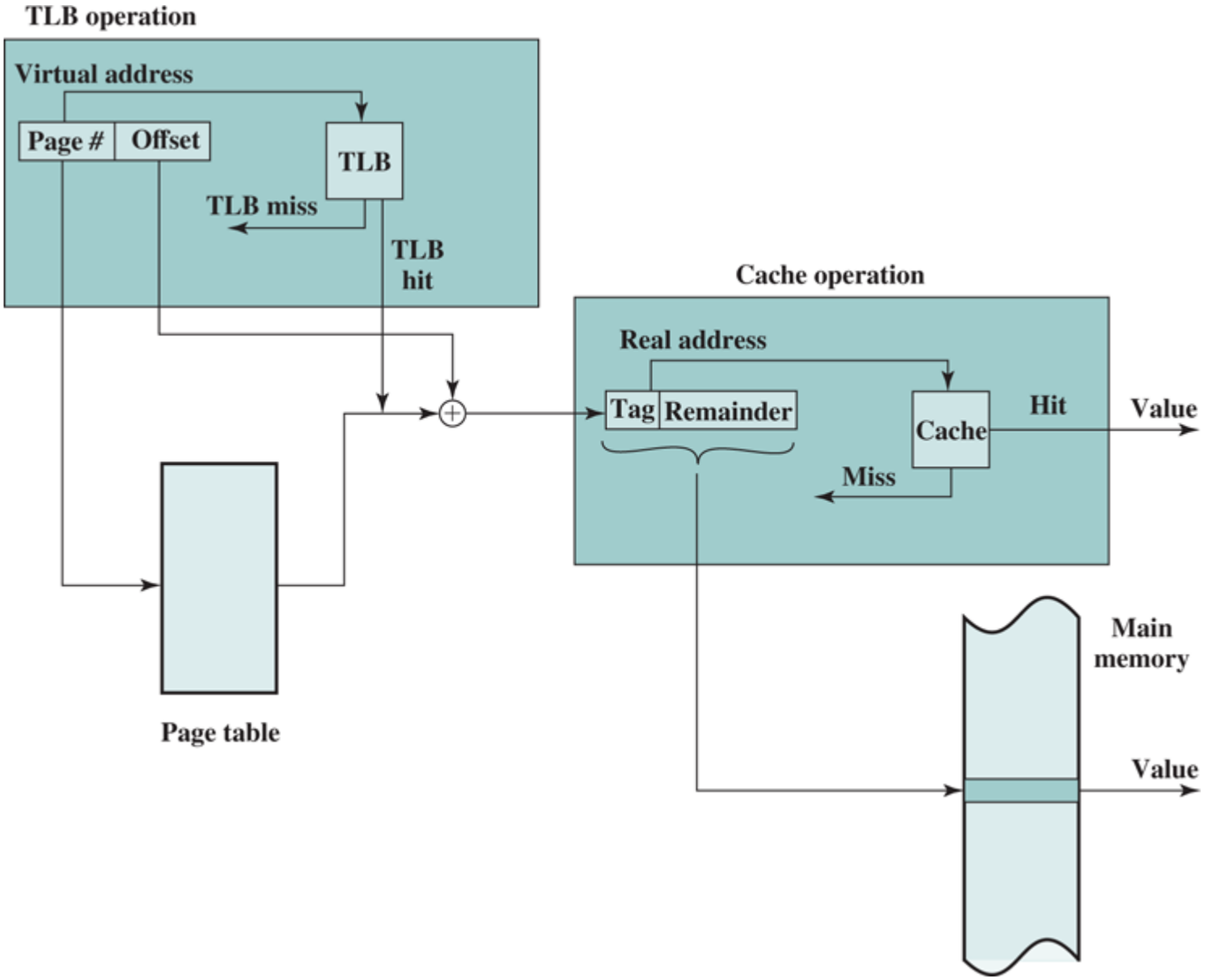


Figure 9.19 Translation Lookaside Buffer and Cache Operation

The reader should be able to appreciate the complexity of the processor hardware involved in a single memory reference. The virtual address is translated into a real address. This involves reference to a page table, which may be in the TLB, in main memory, or on disk. The referenced word may be in cache, in main memory, or on disk. In the latter case, the page containing the word must be loaded into main memory and its block loaded into the cache. In addition, the page table entry for that page must be updated.

Segmentation

There is another way in which addressable memory can be subdivided, known as *segmentation.* Whereas paging is invisible to the programmer and serves the purpose of providing the programmer with a larger address space, segmentation is usually visible to the programmer and is provided as a convenience for organizing programs and data, and as a means for associating privilege and protection attributes with instructions and data.

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments are of variable, indeed dynamic, size. Typically, the programmer or the OS will assign programs and data to different segments. There may be a number of program segments for various types of programs, as well as a number of data segments. Each segment may be assigned access and usage rights. Memory references consist of a (segment number, offset) form of address.

This organization has a number of advantages to the programmer over a non-segmented address space:

1. It simplifies the handling of growing data structures. If the programmer does not know ahead of time how large a particular data structure will become, it is not necessary to guess. The data structure can be assigned its own segment, and the OS will expand or shrink the segment as needed.
2. It allows programs to be altered and recompiled independently without requiring that an entire set of programs be relinked and reloaded. Again, this is accomplished using multiple segments.
3. It lends itself to sharing among processes. A programmer can place a utility program or a useful table of data in a segment that can be addressed by other processes.
4. It lends itself to protection. Because a segment can be constructed to contain a well-defined set of programs or data, the programmer or a system administrator can assign access privileges in a convenient fashion.

These advantages are not available with paging, which is invisible to the programmer. On the other hand, we have seen that paging provides for an efficient form of memory management. To combine the advantages of both, some systems are equipped with the hardware and OS software to provide both.

9.4 Intel x86 Memory Management

Since the introduction of the 32-bit architecture, microprocessors have evolved sophisticated memory management schemes that build on the lessons learned with medium- and large-scale systems. In many cases, the microprocessor versions are superior to their larger-system antecedents. Because the schemes were developed by the microprocessor hardware vendor and may be employed with a variety of operating systems, they tend to be quite general purpose. A representative example is the scheme used on the Intel x86 architecture.

Address Spaces

The x86 includes hardware for both segmentation and paging. Both mechanisms can be disabled, allowing the user to choose from four distinct views of memory:

- **Unsegmented unpaged memory:** In this case, the virtual address is the same as the physical address. This is useful, for example, in low-complexity, high-performance controller applications.
- **Unsegmented paged memory:** Here memory is viewed as a paged linear address space. Protection and management of memory is done via paging. This is favored by some operating systems (e.g., Berkeley UNIX).
- **Segmented unpaged memory:** Here memory is viewed as a collection of logical address spaces. The advantage of this view over a paged approach is that it affords protection down to the level of a single byte, if necessary. Furthermore, unlike paging, it guarantees that the translation table needed (the segment table) is on-chip when the segment is in memory. Hence, segmented unpaged memory results in predictable access times.
- **Segmented paged memory:** Segmentation is used to define logical memory partitions subject to access control, and paging is used to manage the allocation of memory within the partitions. Operating systems such as UNIX System V favor this view.

Segmentation

When segmentation is used, each virtual address (called a logical address in the x86 documentation) consists of a 16-bit segment reference and a 32-bit offset. Two bits of the segment reference deal with the protection mechanism, leaving 14 bits for specifying a particular segment. Thus, with unsegmented memory, the user's virtual memory is $2^{32} = 4\text{Gbytes}$. With segmented memory, the total virtual memory space as seen by a user is $2^{46} = 64\text{terabytes}$ (Tbytes). The physical address space employs a 32-bit address for a maximum of 4 Gbytes.

The amount of virtual memory can actually be larger than the 64 Tbytes. This is because the processor's interpretation of a virtual address depends on which process is currently active. Virtual address space is divided into two parts. One-half of the virtual address space ($8\text{K segments} \times 4\text{Gbytes}$) is global, shared by all processes; the remainder is local and is distinct for each process.

Associated with each segment are two forms of protection: privilege level and access attribute. There are four privilege levels, from most protected (level 0) to least protected (level 3). The privilege level associated with a data segment is its "classification"; the privilege level associated with a program segment is its "clearance." An executing program may only access data segments for which its clearance level is lower than (more privileged) or equal to (same privilege) the privilege level of the data segment.

The hardware does not dictate how these privilege levels are to be used; this depends on the OS

design and implementation. It was intended that privilege level 1 would be used for most of the OS, and level 0 would be used for that small portion of the OS devoted to memory management, protection, and access control. This leaves two levels for applications. In many systems, applications will reside at level 3, with level 2 being unused. Specialized application subsystems that must be protected because they implement their own security mechanisms are good candidates for level 2. Some examples are database management systems, office automation systems, and software engineering environments.

In addition to regulating access to data segments, the privilege mechanism limits the use of certain instructions. Some instructions, such as those dealing with memory-management registers, can only be executed in level 0. I/O instructions can only be executed up to a certain level that is designated by the OS; typically, this will be level 1.

The access attribute of a data segment specifies whether read/write or read-only accesses are permitted. For program segments, the access attribute specifies read/execute or read-only access.

The address translation mechanism for segmentation involves mapping a virtual address into what is referred to as a linear address (**Figure 9.20b**). A virtual address consists of the 32-bit offset and a 16-bit segment selector (**Figure 9.20a**). An instruction fetching or storing an operand specifies the offset and a register containing the segment selector. The segment selector consists of the following fields:

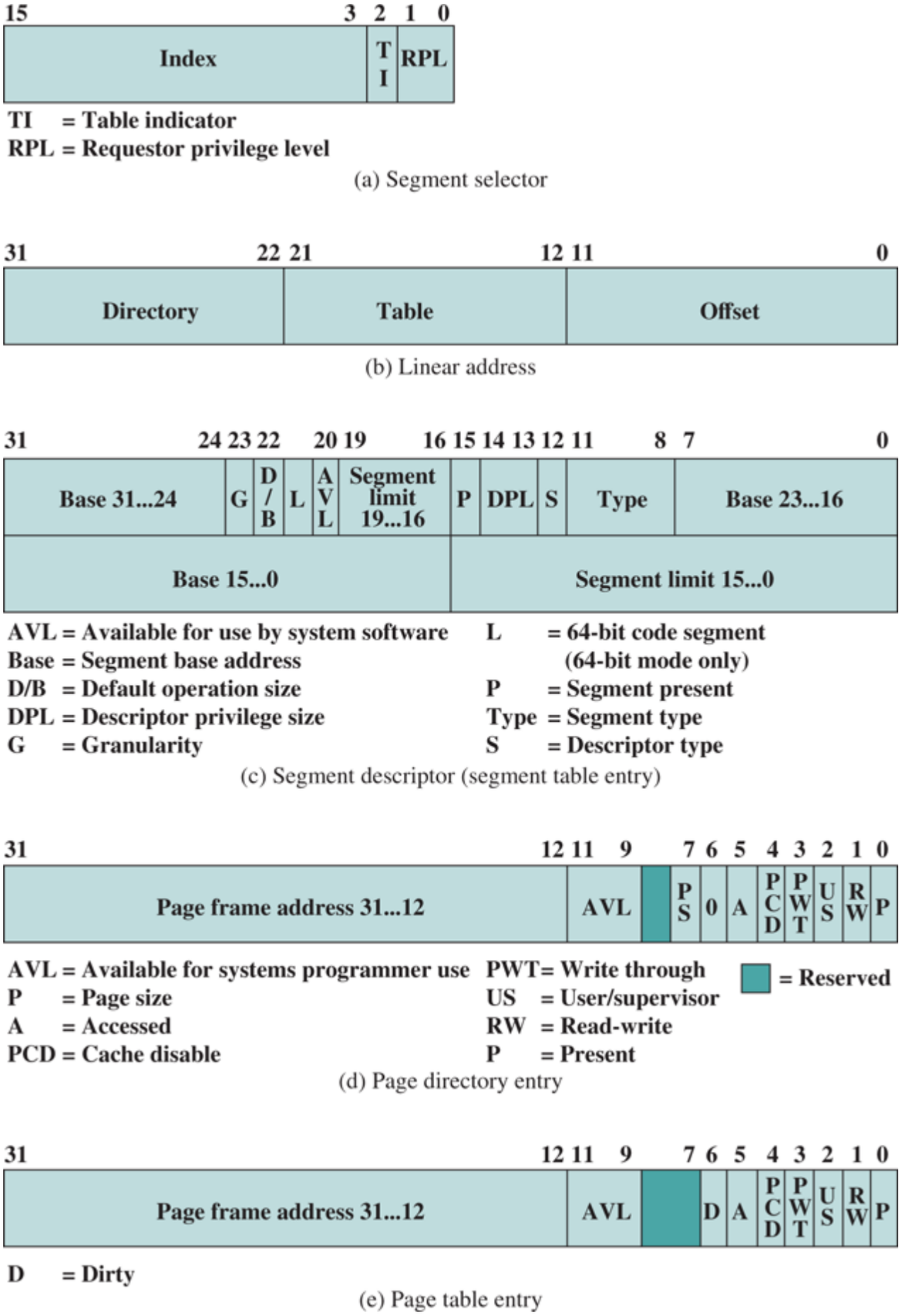


Figure 9.20 Intel x86 Memory Management Formats

- **Table Indicator (TI):** Indicates whether the global segment table or a local segment table should be used for translation.

Segment Number: The number of the segment. This serves as an index into the segment table.

- **Requested Privilege Level (RPL):** The privilege level requested for this access.

Each entry in a segment table consists of 64 bits, as shown in **Figure 9.20c**. The fields are defined in **Table 9.5**.

Table 9.5 x86 Memory Management Parameters

Segment Descriptor (Segment Table Entry)
Base Defines the starting address of the segment within the 4-Gbyte linear address space.
D/B bit In a code segment, this is the D bit and indicates whether operands and addressing modes are 16 or 32 bits.
Descriptor Privilege Level (DPL) Specifies the privilege level of the segment referred to by this segment descriptor.
Granularity bit (G) Indicates whether the Limit field is to be interpreted in units by one byte or 4 Kbytes.
Limit Defines the size of the segment. The processor interprets the limit field in one of two ways, depending on the granularity bit: in units of one byte, up to a segment size limit of 1 Mbyte, or in units of 4 Kbytes, up to a segment size limit of 4 Gbytes.
S bit Determines whether a given segment is a system segment or a code or data segment.
Segment Present bit (P) Used for nonpaged systems. It indicates whether the segment is present in main memory. For paged systems, this bit is always set to 1.
Type Distinguishes between various kinds of segments and indicates the access attributes.
Page Directory Entry and Page Table Entry

Accessed bit (A) This bit is set to 1 by the processor in both levels of page tables when a read or write operation to the corresponding page occurs.
Dirty bit (D) This bit is set to 1 by the processor when a write operation to the corresponding page occurs.
Page Frame Address Provides the physical address of the page in memory if the present bit is set. Since page frames are aligned on 4K boundaries, the bottom 12 bits are 0, and only the top 20 bits are included in the entry. In a page directory, the address is that of a page table.
Page Cache Disable bit (PCD) Indicates whether data from page may be cached.
Page Size bit (PS) Indicates whether page size is 4 Kbyte or 4 Mbyte.
Page Write Through bit (PWT) Indicates whether write-through or write-back caching policy will be used for data in the corresponding page.
Present bit (P) Indicates whether the page table or page is in main memory.
Read/Write bit (RW) For user-level pages, indicates whether the page is read-only access or read/write access for user-level programs.
User/Supervisor bit (US) Indicates whether the page is available only to the operating system (supervisor level) or is available to both operating system and applications (user level).

Paging

Segmentation is an optional feature and may be disabled. When segmentation is in use, addresses

used in programs are virtual addresses and are converted into linear addresses, as just described. When segmentation is not in use, linear addresses are used in programs. In either case, the following step is to convert that linear address into a real 32-bit address.

To understand the structure of the linear address, you need to know that the x86 paging mechanism is actually a two-level table lookup operation. The first level is a page directory, which contains up to 1024 entries. This splits the 4-Gbyte linear memory space into 1024 page groups, each with its own page table, and each 4 Mbytes in length. Each page table contains up to 1024 entries; each entry corresponds to a single 4-Kbyte page. Memory management has the option of using one page directory for all processes, one page directory for each process, or some combination of the two. The page directory for the current task is always in main memory. Page tables may be in virtual memory.

Figure 9.20 shows the formats of entries in page directories and page tables, and the fields are defined in **Table 9.5**. Note that access control mechanisms can be provided on a page or page group basis.

The x86 also makes use of a translation lookaside buffer. The buffer can hold 32 page table entries. Each time that the page directory is changed, the buffer is cleared.

Figure 9.21 illustrates the combination of segmentation and paging mechanisms. For clarity, the translation lookaside buffer and memory cache mechanisms are not shown.

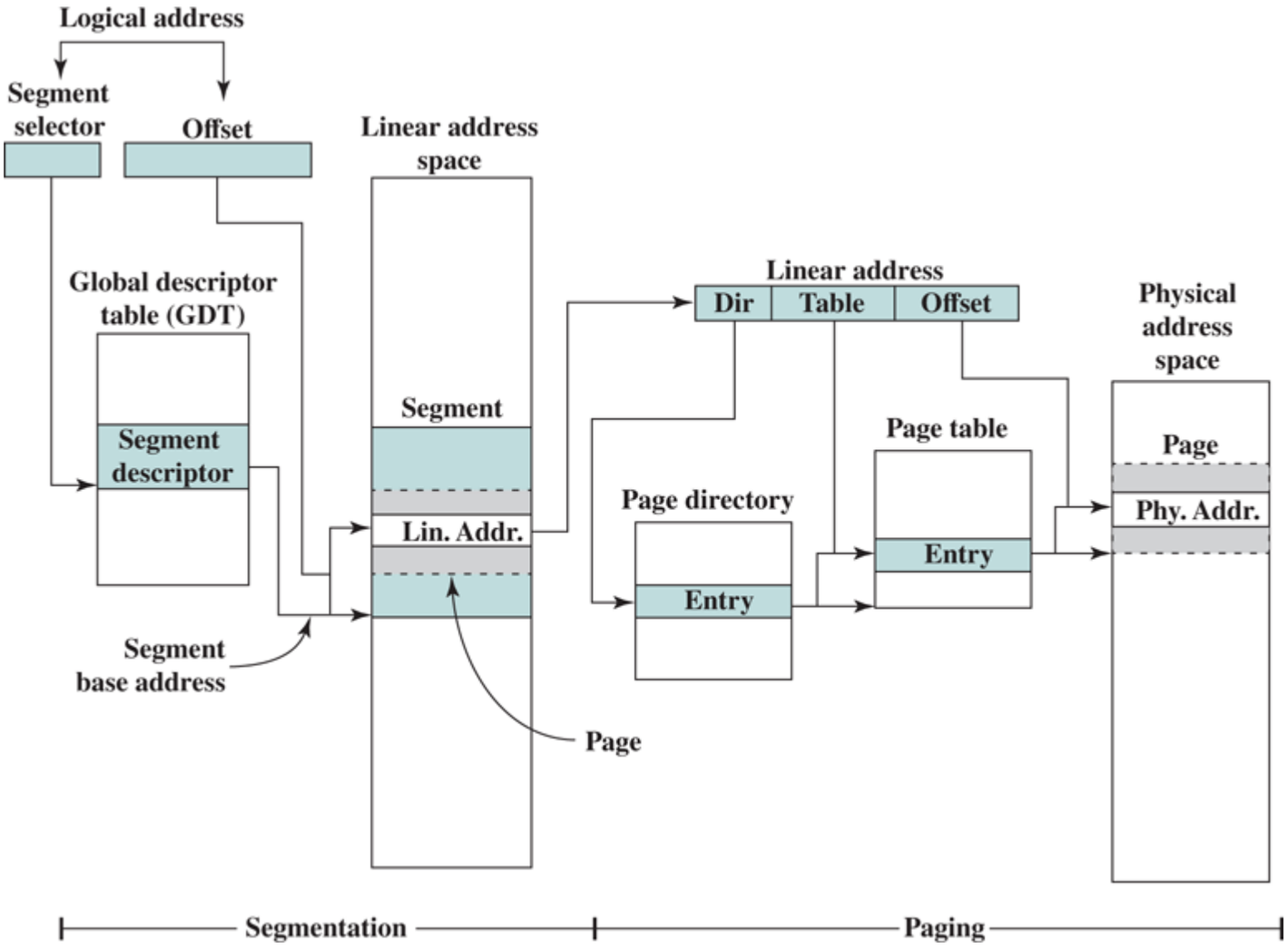


Figure 9.21 Intel x86 Memory Address Translation Mechanisms

Finally, the x86 includes a new extension not found on the earlier 80386 or 80486, the provision for two page sizes. If the PSE (page size extension) bit in control register 4 is set to 1, then the paging unit permits the OS programmer to define a page as either 4 Kbyte or 4 Mbyte in size.

When 4-Mbyte pages are used, there is only one level of table lookup for pages. When the hardware accesses the page directory, the page directory entry (**Figure 9.20d**) has the PS bit set to 1. In this case, bits 9 through 21 are ignored and bits 22 through 31 define the base address for a 4-Mbyte page in memory. Thus, there is a single page table.

The use of 4-Mbyte pages reduces the memory-management storage requirements for large main memories. With 4-Kbyte pages, a full 4-Gbyte main memory requires about 4 Mbytes of memory just for the page tables. With 4-Mbyte pages, a single table, 4 Kbytes in length, is sufficient for page memory management.

9.5 ARM Memory Management

ARM provides a versatile virtual memory system architecture that can be tailored to the needs of the embedded system designer.

Memory System Organization

Figure 9.22 provides an overview of the memory management hardware in the ARM for virtual memory. The virtual memory translation hardware uses one or two levels of tables for translation from virtual to physical addresses, as explained subsequently. The translation lookaside buffer (TLB) is a cache of recent page table entries. If an entry is available in the TLB, then the TLB directly sends a physical address to main memory for a read or write operation. As explained in **Chapter 5**, data is exchanged between the processor and main memory via the cache. If a logical cache organization is used (**Figure 5.5a**), then the ARM supplies that address directly to the cache as well as supplying it to the TLB when a cache miss occurs. If a physical cache organization is used (**Figure 5.5b**), then the TLB must supply the physical address to the cache.

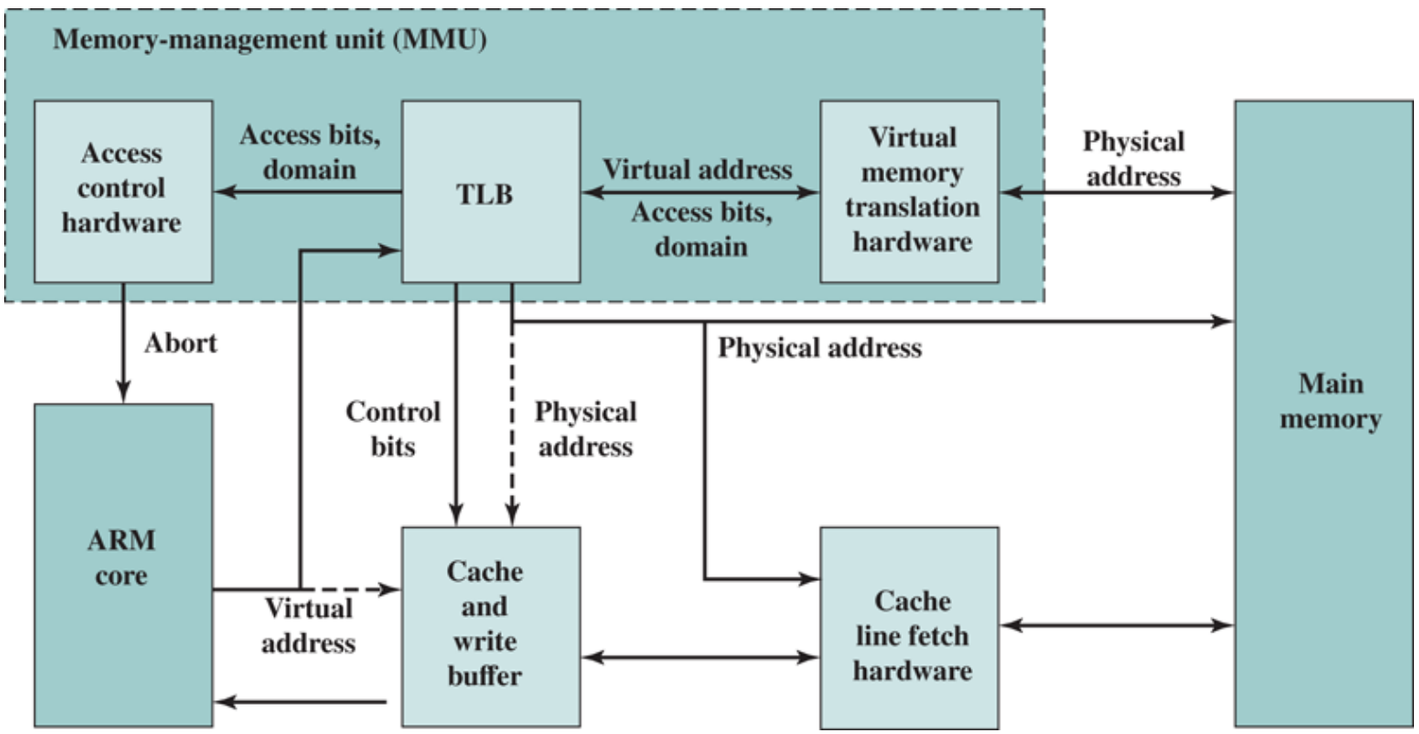


Figure 9.22 ARM Memory System Overview

Entries in the translation tables also include access control bits, which determine whether a given process may access a given portion of memory. If access is denied, access control hardware supplies an abort signal to the ARM processor.

Virtual Memory Address Translation

The ARM supports memory access based on either sections or pages:

- **Supersections (optional):** Consist of 16-MB blocks of main memory.
- **Sections:** Consist of 1-MB blocks of main memory.

- **Large pages:** Consist of 64-kB blocks of main memory.
- **Small pages:** Consist of 4-kB blocks of main memory.

Sections and supersections are supported to allow mapping of a large region of memory while using only a single entry in the TLB. Additional access control mechanisms are extended within small pages to 1kB subpages, and within large pages to 16kB subpages. The translation table held in main memory has two levels:

- **Level 1 table:** Holds level 1 descriptors that contain the base address and translation properties for a Section and Supersection; and translation properties and pointers to a level 2 table for a large page or a small page.
- **Level 2 table:** Holds level 2 descriptors that contain the base address and translation properties for a Small page or a Large page. A level 2 table requires 1 kB of memory.

The memory-management unit (MMU) translates virtual addresses generated by the processor into physical addresses to access main memory, and also derives and checks the access permission. Translations occur as the result of a TLB miss, and start with a first-level fetch. A section-mapped access only requires a first-level fetch, whereas a page-mapped access also requires a second-level fetch.

Figure 9.23 shows the two-level address translation process for small pages. There is a single level 1 (L1) page table with 4K 32-bit entries. Each L1 entry points to a level 2 (L2) page table with 256 32-bit entries. Each of the L2 entry points to a 4-kB page in main memory. The 32-bit virtual address is interpreted as follows: The most significant 12 bits are an index into the L1 page table. The next 8 bits are an index into the relevant L2 page table. The least significant 12 bits index a byte in the relevant page in main memory.

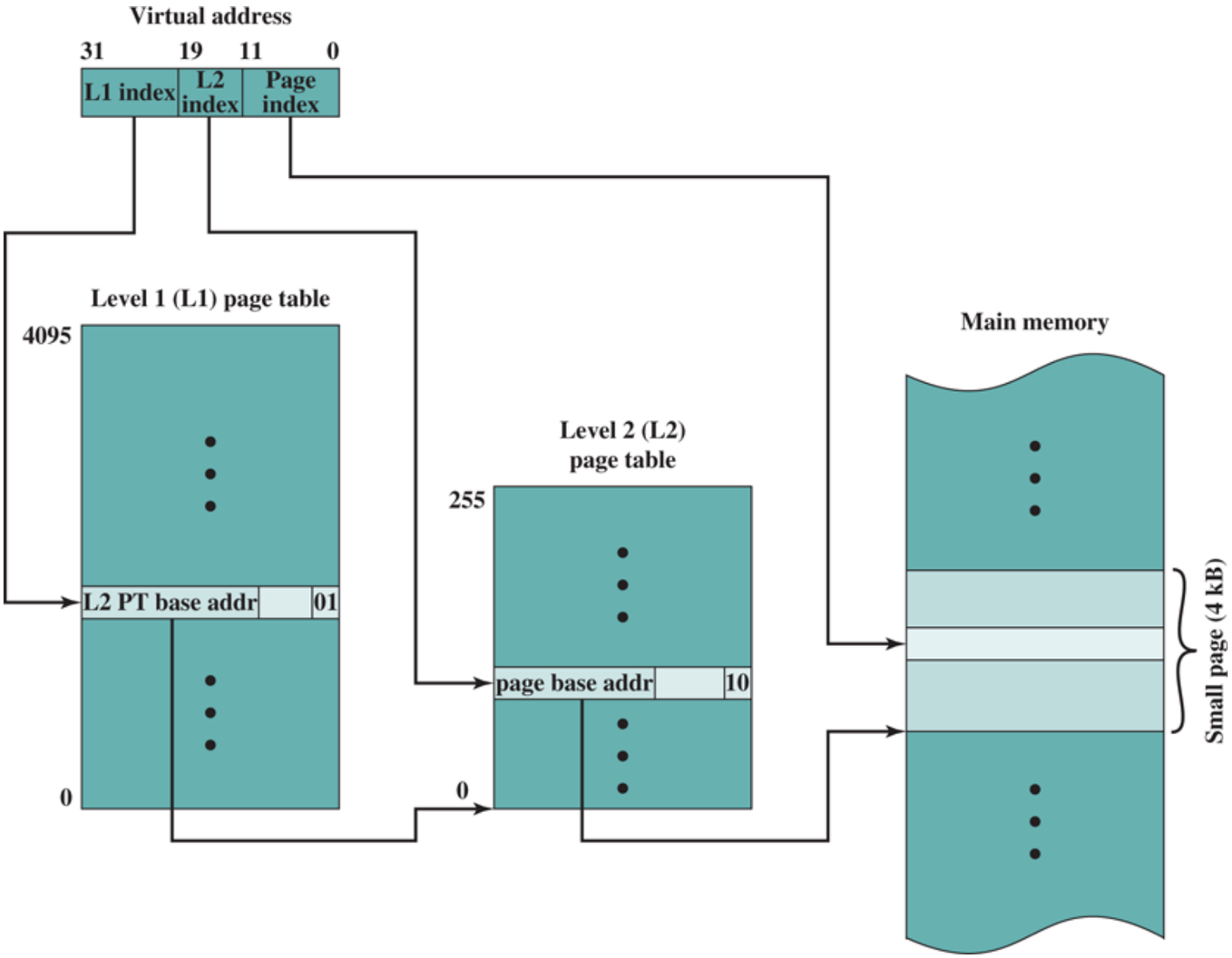
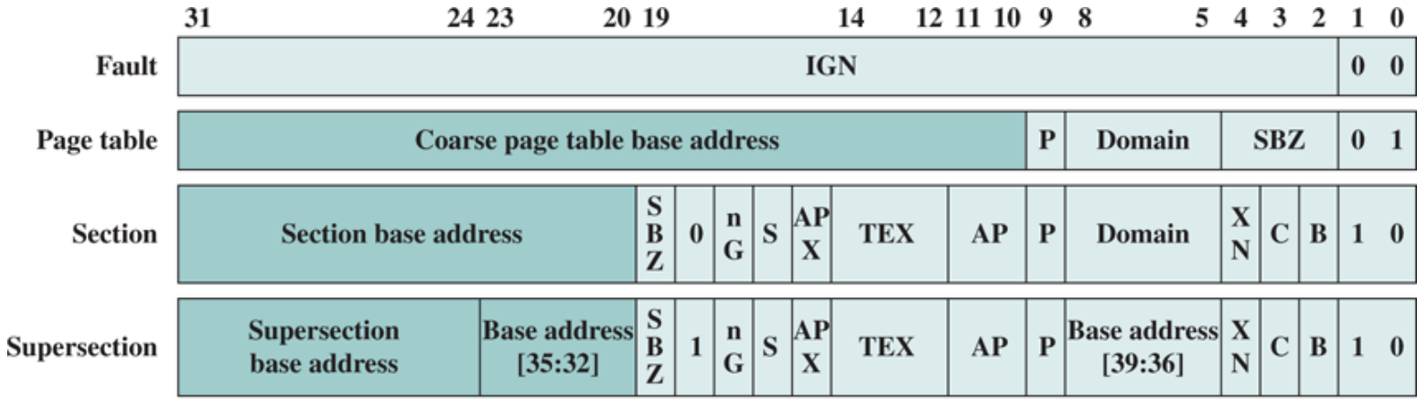


Figure 9.23 ARM Virtual Memory Address Translation for Small Pages

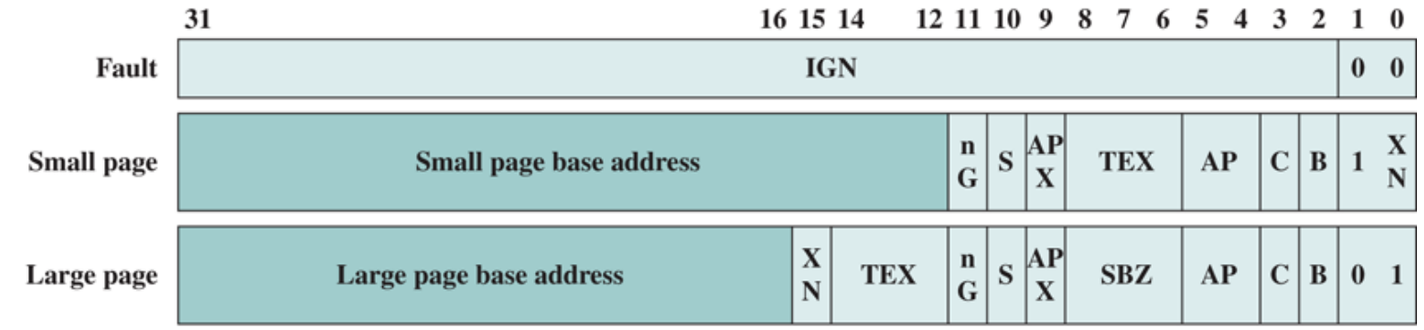
A similar two-page lookup procedure is used for large pages. For sections and supersections, only the L1 page table lookup is required.

Memory-Management Formats

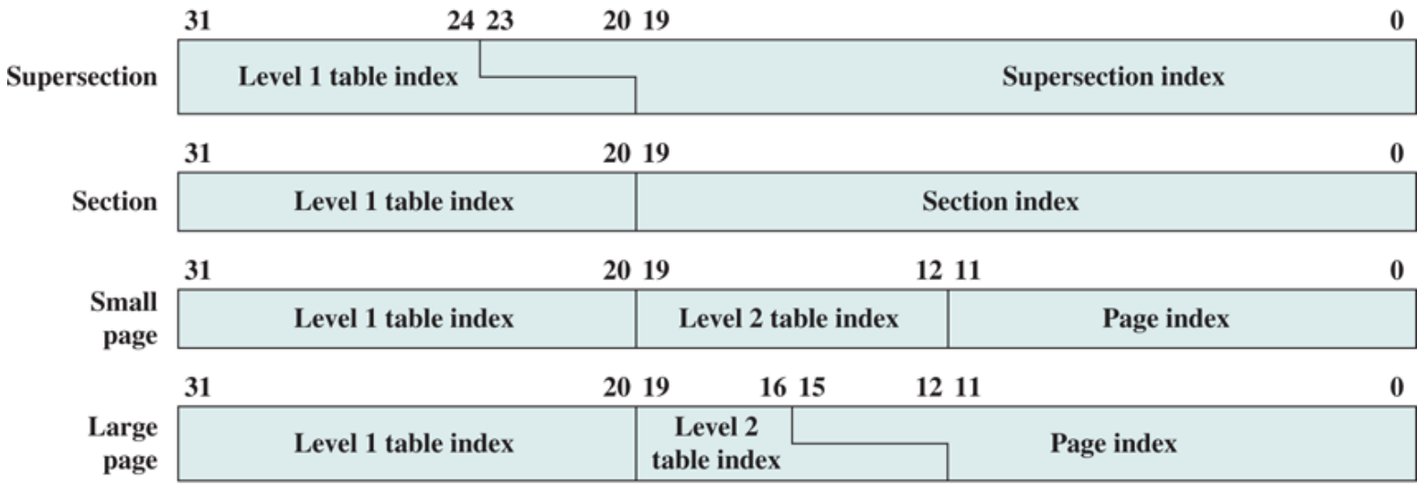
To get a better understanding of the ARM memory management scheme, we consider the key formats, as shown in **Figure 9.24**. The control bits shown in this figure are defined in **Table 9.6**.



(a) Alternative first-level descriptor formats



(b) Alternative second-level descriptor formats



(c) Virtual memory address formats

Figure 9.24 ARM Memory-Management Formats

Table 9.6 ARM Memory-Management Parameters

Access Permission (AP), Access Permission Extension (APX) These bits control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, a Permission Fault is raised.

Bufferable (B) bit Determines, with the TEX bits, how the write buffer is used for cacheable memory.
Cacheable (C) bit Determines whether this memory region can be mapped through the cache.
Domain Collection of memory regions. Access control can be applied on the basis of domain.
not Global (nG) Determines whether the translation should be marked as global (0), or process specific (1).
Shared (S) Determines whether the translation is for not-shared (0), or shared (1) memory.
SBZ Should be zero.
Type Extension (TEX) These bits, together with the B and C bits, control accesses to the caches, how the write buffer is used, and if the memory region is shareable and therefore must be kept coherent.
Execute Never (XN) Determines whether the region is executable (0) or not executable (1).

For the L1 table, each entry is a descriptor of how its associated 1-MB virtual address range is mapped. Each entry has one of four alternative formats:

- $\textbf{Bits}[1:0] = \textbf{00}$: The associated virtual addresses are unmapped, and attempts to access them generate a translation fault.
- $\textbf{Bits}[1:0] = \textbf{01}$: The entry gives the physical address of an L2 page table, which specifies how the associated virtual address range is mapped.
- $\textbf{Bits}[1:0] = \textbf{01}$: **and** $\textbf{bit19} = \textbf{0}$: The entry is a section descriptor for its associated virtual addresses.
- $\textbf{Bits}[1:0] = \textbf{01}$: **and** $\textbf{bit19} = \textbf{1}$: The entry is a supersection descriptor for its associated virtual addresses.

Entries with $\text{bits}\ [1:0] = 11$ are reserved.

For memory structured into pages, a two-level page table access is required. Bits [31:10] of the L1 page entry contain a pointer to a L2 page table. For small pages, the L2 entry contains a 20-bit pointer to the base address of a 4-kB page in main memory.

For large pages, the structure is more complex. As with virtual addresses for small pages, a virtual address for a large page structure includes a 12-bit index into the level one table and an 8-bit index into the L2 table. For the 64-kB large pages, the page index portion of the virtual address must be 16 bits. To accommodate all of these bits in a 32-bit format, there is a 4-bit overlap between the page index field and the L2 table index field. ARM accommodates this overlap by requiring that each page table entry in a L2 page table that supports large pages be replicated 16 times. In effect, the size of the L2 page table is reduced from 256 entries to 16 entries, if all of the entries refer to large pages. However, a given L2 page can service a mixture of large and small pages, hence the need for the replication of large page entries.

For memory structured into sections or supersections, a one-level page table access is required. For sections, bits [31:20] of the L1 entry contain a 12-bit pointer to the base of the 1-MB section in main memory.

For supersections, bits [31:24] of the L1 entry contain an 8-bit pointer to the base of the 16-MB section in main memory. As with large pages, a page table entry replication is required. In the case of supersections, the L1 table index portion of the virtual address overlaps by 4 bits with the supersection index portion of the virtual address. Therefore, 16 identical L1 page table entries are required.

The range of physical address space can be expanded by up to eight additional address bits (bits [23:20] and [8:5]). The number of additional bits is implementation dependent. These additional bits can be interpreted as extending the size of physical memory by as much as a factor of $2^8 = 256$. Thus, physical memory may in fact be as much as 256 times as large as the memory space available to each individual process.

Access Control

The AP access control bits in each table entry control access to a region of memory by a given process. A region of memory can be designated as no access, read only, or read-write. Further, the region can be designated as privileged access only, reserved for use by the OS and not by applications.

ARM also employs the concept of a domain, which is a collection of sections and/or pages that have particular access permissions. The ARM architecture supports 16 domains. The domain feature allows multiple processes to use the same translation tables while maintaining some protection from each other.

Each page table entry and TLB entry contains a field that specifies which domain the entry is in. A 2-bit field in the Domain Access Control Register controls access to each domain. Each field allows the access to an entire domain to be enabled and disabled very quickly, so that whole memory areas can be swapped in and out of virtual memory very efficiently. Two kinds of domain access are supported:

- **Clients:** Users of domains (execute programs and access data) that must observe the access permissions of the individual sections and/or pages that make up that domain.
- **Managers:** Control the behavior of the domain (the current sections and pages in the domain, and the domain access), and bypass the access permissions for table entries in that domain.

One program can be a client of some domains, and a manager of some other domains, and have no access to the remaining domains. This allows very flexible memory protection for programs that

access different memory resources.

9.6 Key Terms, Review Questions, and Problems

Key Terms

batch system

demand paging

interactive operating system

interrupt

job control language (JCL)

kernel

logical address

long-term scheduling

medium-term scheduling

memory management

memory protection

multiprogramming

multitasking

nucleus

operating system (OS)

page table

paging

partitioning

physical address

privileged instruction

process

process control block

process state

real memory

resident monitor

segmentation

short-term scheduling

swapping

thrashing

time-sharing system

translation lookaside buffer (TLB)

utility

virtual memory

Review Questions

9.1 What is an operating system?
9.2 List and briefly define the key services provided by an OS.
9.3 List and briefly define the major types of OS scheduling.
9.4 What is the difference between a process and a program?
9.5 What is the purpose of swapping?
9.6 If a process may be dynamically assigned to different locations in main memory, what is the implication for the addressing mechanism?
9.7 Is it necessary for all of the pages of a process to be in main memory while the process is executing?
9.8 Must the pages of a process in main memory be contiguous?
9.9 Is it necessary for the pages of a process in main memory to be in sequential order?
9.10 What is the purpose of a translation lookaside buffer?

Problems

9.1 Suppose that we have a multiprogrammed computer in which each job has identical characteristics. In one computation period, *T*, for a job, half the time is spent in I/O and the other half in processor activity. Each job runs for a total of *N* periods. Assume that a simple round-robin priority is used, and that I/O operations can overlap with processor operation. Define the following quantities:

- Turnaround time = actual to complete a job..
- Throughput = average number of jobs completed per time period T.
- Processor utilization = percentage of time that the processor is active (not waiting).

Compute these quantities for one, two, and four simultaneous jobs, assuming that the period *T* is distributed in each of the following ways:

a. I/O first half, processor second half;
b. I/O first and fourth quarters, processor second and third quarters.

9.2 An I/O-bound program is one that, if run alone, would spend more time waiting for I/O than using the processor. A processor-bound program is the opposite. Suppose a short-term scheduling algorithm favors those programs that have used little processor time in the recent past. Explain why this algorithm favors I/O-bound programs and yet does not permanently deny processor time to processor-bound programs.
9.3 A program computes the row sums

$$C_i = \sum_{j=1}^{N} a_{ij}$$

of an array *A* that is 100 by 100. Assume that the computer uses demand paging with a page size of 1000 words, and that the amount of main memory allotted for data is five page frames. Is there any difference in the page fault rate if *A* were stored in virtual memory by rows or columns? Explain.

9.4 Consider a fixed partitioning scheme with equal-size partitions of 2 bytes and a total main memory size of 2^{24}bytes. A process table is maintained that includes a pointer to a partition for each resident process. How many bits are required for the pointer?
9.5 Consider a dynamic partitioning scheme. Show that, on average, the memory contains half as many holes as segments.
9.6 Suppose the page table for the process currently executing on the processor looks like the following. All numbers are decimal, everything is numbered starting from zero, and all addresses are memory byte addresses. The page size is 1024 bytes.

Virtual page number	Valid bit	Reference bit	Modify bit	Page frame number
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a. Describe exactly how, in general, a virtual address generated by the CPU is translated into a physical main memory address.
b. What physical address, if any, would each of the following virtual addresses correspond to? (Do not try to handle any page faults, if any.)
    i. 1052
    ii. 2221
    iii. 5499

9.7 Give reasons that the page size in a virtual memory system should be neither very small nor very large.
9.8 A process references five pages, A, B, C, D, and E, in the following order:
A; B; C; D; A; B; E; A; B; C; D; E
Assume that the replacement algorithm is first-in-first-out and find the number of page transfers during this sequence of references starting with an empty main memory with three page frames. Repeat for four page frames.
9.9 The following sequence of virtual page numbers is encountered in the course of execution on a computer with virtual memory:
3 4 2 6 4 7 1 3 2 6 3 5 1 2 3
Assume that a least recently used page replacement policy is adopted. Plot a graph of page hit ratio (fraction of page references in which the page is in main memory) as a function of main-memory page capacity *n* for $1 \leq n \leq 8$. Assume that main memory is initially empty.
9.10 In the VAX computer, user page tables are located at virtual addresses in the system space. What is the advantage of having user page tables in virtual rather than main memory? What is the disadvantage?
9.11 Suppose the program statement

```
for (i = 1 ; i6 = n ; i+)
a[i]=b[i]+c[i];
```

is executed in a memory with page size of 1000 words. Let $n = 1000$. Using a machine that has a full range of register-to-register instructions and employs index registers, write a hypothetical program to implement the foregoing statement. Then show the sequence of page references during execution.

9.12 The IBM System/370 architecture uses a two-level memory structure and refers to the two levels as segments and pages, although the segmentation approach lacks many of the features described earlier in this chapter. For the basic 370 architecture, the page size may be either 2 Kbytes or 4 Kbytes, and the segment size is fixed at either 64 Kbytes or 1 Mbyte. For the 370/XA and 370/ESA architectures, the page size is 4 Kbytes and the segment size is 1 Mbyte. Which advantages of segmentation does this scheme lack? What is the benefit of segmentation for the 370?

9.13 Consider a computer system with both segmentation and paging. When a segment is in memory, some words are wasted on the last page. In addition, for a segment size *s* and a page size *p*, there are *s*/*p* page table entries. The smaller the page size, the less waste in the last page of the segment, but the larger the page table. What page size minimizes the total overhead?

9.14 A computer has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20 ns are required to access it. If it is in main memory but not in the cache, 60 ns are needed to load it into the cache, and then the reference is started again. If the word is not in main memory, 12 ms are required to fetch the word from disk, followed by 60 ns to copy it to the cache, and then the reference is started again. The cache hit ratio is 0.9 and the main-memory hit ratio is 0.6. What is the average time in ns required to access a referenced word on this system?

9.15 Assume a task is divided into four equal-sized segments and that the system builds an eight-entry page descriptor table for each segment. Thus, the system has a combination of segmentation and paging. Assume also that the page size is 2 Kbytes.

a. What is the maximum size of each segment?
b. What is the maximum logical address space for the task?
c. Assume that an element in physical location 00021ABC is accessed by this task. What is the format of the logical address that the task generates for it? What is the maximum physical address space for the system?

9.16 Assume a microprocessor capable of accessing up to 2^{32}bytes of physical main memory. It implements one segmented logical address space of maximum size 2^{31}bytes. Each instruction contains the whole two-part address. External memory management units (MMUs) are used, whose management scheme assigns contiguous blocks of physical memory of fixed size 2^{22}bytes to segments. The starting physical address of a segment is always divisible by 1024. Show the detailed interconnection of the external mapping mechanism that converts logical addresses to physical addresses using the appropriate number of MMUs, and show the detailed internal structure of an MMU (assuming that each MMU contains a 128-entry directly mapped segment descriptor cache) and how each MMU is selected.

9.17 Consider a paged logical address space (composed of 32 pages of 2 Kbytes each) mapped into a 1-Mbyte physical memory space.

a. What is the format of the processor's logical address?
b. What is the length and width of the page table (disregarding the "access rights" bits)?
c. What is the effect on the page table if the physical memory space is reduced by half?

9.18 In IBM's mainframe operating system, OS/390, one of the major modules in the kernel is the System Resource Manager (SRM). This module is responsible for the allocation of resources among address spaces (processes). The SRM gives OS/390 a degree of

sophistication unique among operating systems. No other mainframe OS, and certainly no other type of OS, can match the functions performed by SRM. The concept of resource includes processor, real memory, and I/O channels. SRM accumulates statistics pertaining to utilization of processor, channel, and various key data structures. Its purpose is to provide optimum performance based on performance monitoring and analysis. The installation sets forth various performance objectives, and these serve as guidance to the SRM, which dynamically modifies installation and job performance characteristics based on system utilization. In turn, the SRM provides reports that enable the trained operator to refine the configuration and parameter settings to improve user service.

This problem concerns one example of SRM activity. Real memory is divided into equal-sized blocks called frames, of which there may be many thousands. Each frame can hold a block of virtual memory referred to as a page. SRM receives control approximately 20 times per second and inspects each and every page frame. If the page has not been referenced or changed, a counter is incremented by 1. Over time, SRM averages these numbers to determine the average number of seconds that a page frame in the system goes untouched. What might be the purpose of this and what action might SRM take?

9.19 For each of the ARM virtual address formats shown in **Figure 9.24** , show the physical address format.

9.20 Draw a figure similar to **Figure 9.23** for ARM virtual memory translation when main memory is divided into sections.