UNIVERSITY OF BAGHDAD

College of Education for Pure Science (Ibn Al-Haitham)

Department of Computer Science

Operating System

Dr. Omar Zeyad Akif

2021-2022

Fourth Stage

Chapter One:

1.1.Introduction:

A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. All in all, it is a complex system. It is an extremely difficult job to write programs that keep tracking of all these components and use them correctly. For this reason, computers are equipped with a layer of software called the operating system, whose job is to manage all these devices and provide user programs with a simpler interface to the hardware.

O.S. Definitions:

Operating systems is a program that acts as an intermediary between a user of a computer and the computer hardware (H/W). The purpose of an operating system is to provide the environment in which the user can execute programs. The O.S. is a resource allocator (managers all resources, decides between conflicting requests for efficient and fair resource use) and is a control program (controls execution of programs) to prevent errors and improper use of the computer.

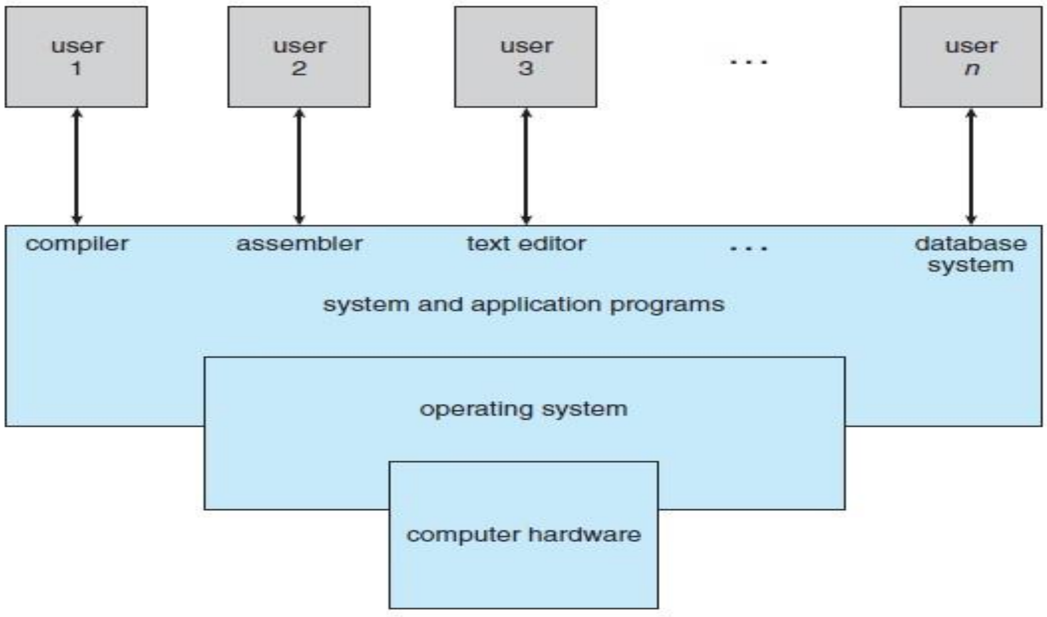


Figure 1-1 A computer system consists of hardware, system programs, and application programs

1.2. Computer System Components:

An O.S. is an important part of almost every computer system. A Computer System can be divided roughly into four components:

1. The (H/W) (CPU, memory, I/O devices).
2. Operating system (O.S.).
3. Application programs (Assembler, data base, compiler, text, and editor).
4. The users (people, machines, other computers).

1.3. O.S. goals:

1. The primary goal of an O.S. is to make the Computer System C/S convenient to use. Operating systems exist because they are supposed to make it easier to compute with them than without them. This view is particularly clear when you look at operating systems for small personal computers.

2. A secondary goal is to use the computer H/W in an efficient manner. This goal is particularly important for large, shared multiuser systems. These systems are typically expensive, so it is desirable to make them as efficient as possible.

These two goals convenience and efficiency are sometimes contradictory. In the past, efficiency considerations were often more important than convenience. Thus, much of operating-system theory concentrates on optimal use of computing resources.

1.4. The O.S. Functions:

O.S. performs many functions such as: -

1. Implementing the user interface.
2. Sharing H/W among users.
3. Allowing users to share data among themselves.
4. Preventing users from interfering with one another.
5. Scheduling resources among users.

6. Facilitating I/O.

7. Recovering from errors.

8. Accounting for resource usage.

9. Facilitating parallel operations.

10. Organizing data for secure and rapid access.

11. Handling network communications.

1.5. O.S. Categories:

The main categories of modern O.S. may be classified into three groups, which are distinguished by the nature of interaction that takes place between the computer and the user:

1.5.1. Batch -System

In this type of O.S. Users submit jobs on a regular schedule (e.g. daily, weekly, monthly) to a central place where the user of such system did not interact directly with C/S. To speed up processing, jobs with similar needs were batched together and were run through the computer as a group. Thus, the programmers would leave their programs with the operator. The major task of this type was to transfer control automatically from one job to the next. The O.S is always resident in memory as in the figure 1-2.

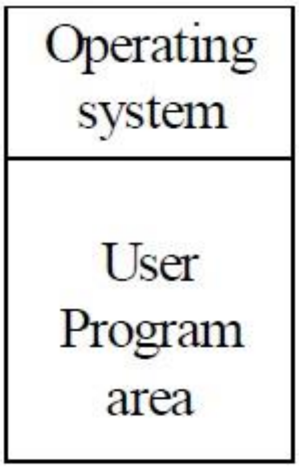


Figure 1-2 Memory layout for a simple batch system

The output from each job would be send back to the appropriate programmer.

An advantage of batch system is very simple.

Disadvantages:

There is no direct interaction between the user and the job while that job is executing.

The delay between job submission and job completion (called turnaround time) may result from amount of computing time needed.

1.5.2. Time-sharing system:

This type of O.S. provides online communication between the user and the system, where the user will give instruction to the O.S. or to the program directly (usually from terminal) and receives an immediate response, therefore some time called an interactive system. The Time-Sharing system allows many users simultaneously share the computer system where little CPU time is needed for each user. As the system switches rapidly from one user to the next user is given the impression that they each have their own computer, while actually one C/S shared among the many users.

Advantage: Reduce the CPU idle time.

Disadvantage: More Complex.

1.5.3. Real-Time system

A Real-Time system is used when there are rigid time requirements on the operation of a processor or the flow of data. A Real-time system guarantees that critical tasks complete on time. The secondary storage of any sort is usually limited; data instead being stored in short-term memory (Rom), (The Radar system is a good example for the real time system).

1.6. Performance Development:

O.S. attempted to schedule computational activities to ensure good performance, where many facilities had been added to O.S. some of these are:

1.6.1. On-Line and off-Line operations:

A special subroutine was written for each I/O device called a device-driver, and some peripherals (I/O devices) has been equipped for either On-Line operation, in which they are connected to the processor, or off-line operations in which they are run by control units not connected to the central C/S.

1.6.2. 2. Buffering

A buffer is an area or primary storage for holding data during I/O transfers, where the I/O transfer speed depends on many factors related to I/O Hardware but normally unrelated to processor operation. On input the data placed in the buffer by an I/O channel when the transfer is complete the data may be accessed by the processor. There are two types of buffering:

1.6.2.1. The single-buffered:

The channel deposits data in a buffer the processor will accessed that data the channel deposits the next data, etc. while the channel is depositing data no, processing on that data may occur.

1.6.2.2. The Double-buffering:

This system allows overlap of I/O operation with processing; while the channel is depositing data in one buffer the processor may be processing data in the other buffer. When the processor is finished processing data in one buffer it may process data in the second buffer. In buffering the CPU and I/O are both busy.

1.6.3. Spooling: (Simultaneous Peripheral Operation On-line)

Spooling uses the disk as a very large buffer for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them. Spooling is now a standard feature of most O.S. Spooling allows the computation of one job can overlap with the I/O of another jobs, therefore spooling can keep both CPU and the I/O devices working as much higher rates. The figure below has shown the spooling layout.

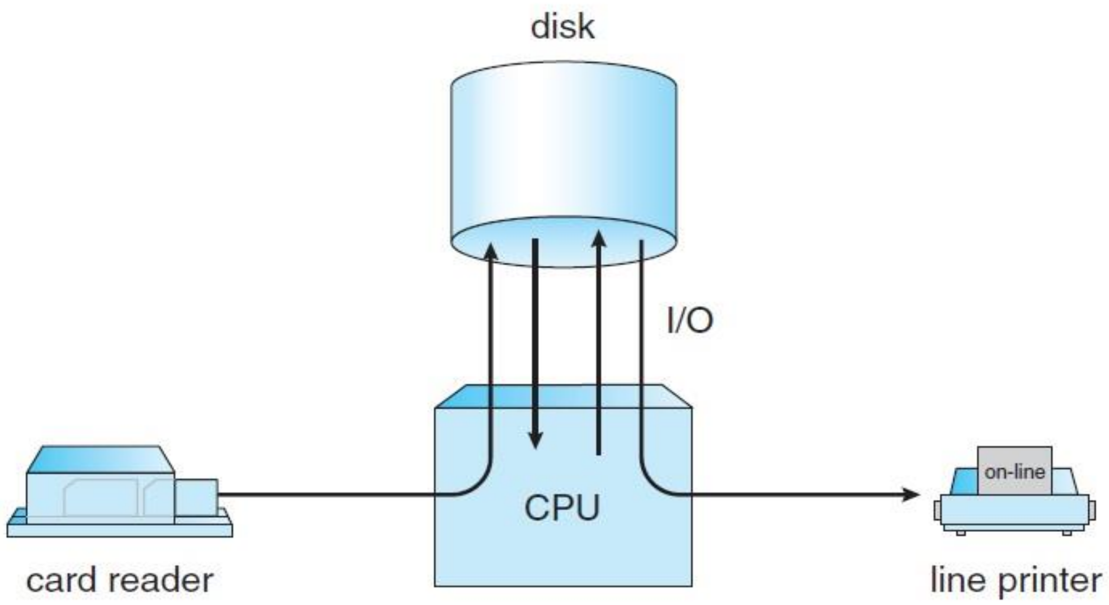


Figure 1-3 Spooling

1.7. Multiprogramming

Spooling provides an important data structure called a job pool kept on disk. The O.S. picks one job from the pool and begins to execute it. In multiprogramming system, when the job may have to wait for any reason such as an I/O request, the O.S. simply switches to and executes another job. When the second job needs to wait the CPU is switches to another job and So on. Then the CPU will never be idle. The figure shows the multiprogramming layout where the O.S. Keeps Several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool.

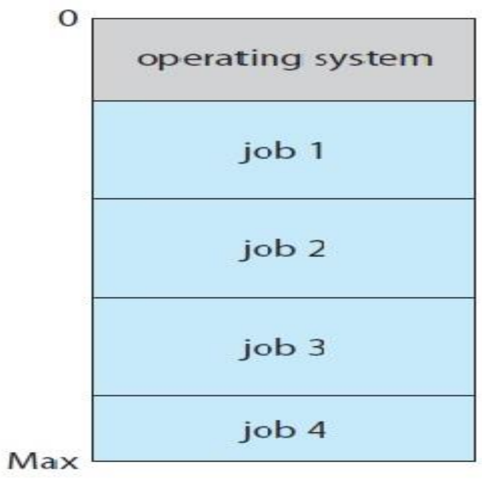


Figure 1-4 Memory layout for a multiprogramming system

1.8. Parallel Systems

Most systems today are a single-processor system that is they have one main CPU. There is a trend to have multiprocessor system, where such communication sharing the computer Bus, the clock, and sometimes memory and peripheral devices, as in the figure behind. The advantage of this type of systems is to increase the throughput (the number of jobs completed in unit of time). Multiprocessors can also save money compared to multiple single systems because the processors can share peripherals, cabinets, and power supplies. Another reason for multiprocessor systems is that they increase reliability. The most common multiple-processor systems now use the ***symmetric multiprocessing*** model, in which each processor runs an identical copy of the operating system, and these copies communicate with one another as needed. Some systems use ***asymmetric multiprocessing,*** in which each processor is assigned a specific task. A master processor controls the system; the other processors either look to the master for instruction or have predefined tasks. This scheme defines a master-slave relationship. The master processor schedules and allocates work to the slave processors.

1.9. Distributed systems

A resent tread in C/S is to distribute computation among several processors. In Contrast to the parallel system, the processors do not share memory and clock. The processors communicate with one another through various communication lines, such as high speed buses or telephone lines. This type of systems called a distributed system. There is a variety of reasons for building distributed systems, the major ones being these:

1. Resource sharing. If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another.
2. Computation speedup. If a particular computation can be partitioned into a number of sub computations that can run concurrently, then a distributed

system may allow us to distribute the computation among the various sites to run that computation.

3. Concurrently. In addition, if a particular site is currently overloaded with jobs, some of them may be moved to other, lightly loaded, sites. This movement of jobs is called *load sharing.*
4. Reliability. If one site fails in a distributed system, the remaining sites can potentially continue operating.

Chapter Two

2.1. Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (Figure 2.1). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

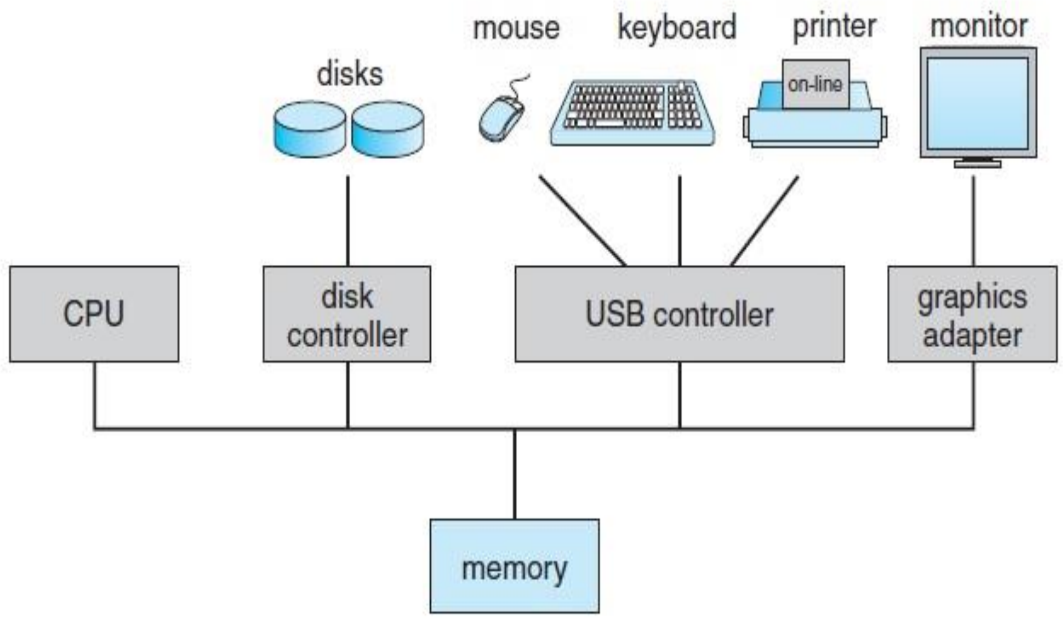


Figure 2-1 A modern computer system

For a computer to start running for instance, when it is powered up or rebooted it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, it is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running. On UNIX, the first system process is “init,” and it starts many other daemons. Once this phase is complete, the system is fully booted, and the system waits for some event to occur.

2.2. I/O Interrupts

The occurrence of an event is usually signalled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**). When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Interrupts are an important part of a computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. The interrupt architecture must also save the address of the interrupted instruction. Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number. More recent architectures store the return address on the system stack. If the interrupt routine needs to modify the processor state for instance, by modifying register values it must explicitly save the current state and then restore that state before returning. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

2.3. Storage Structure

The CPU can load instructions only from memory, so any programs to run must be stored there. General-purpose computers run most of their programs from rewritable memory, called main memory (also called **random-access memory**, or **RAM**). Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**. Computers use other forms of memory as well. We have already mentioned read-only memory, ROM) and electrically erasable programmable read-only memory, EEPROM). Because ROM cannot be changed, only static programs, such as the bootstrap program described earlier, are stored there. The immutability of ROM is of use in game cartridges. EEPROM can be changed but cannot-be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution. A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory. Notice that the memory unit sees only a stream of memory addresses. It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, or some other means) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a memory address is generated by a program. We are interested only in the sequence of memory addresses generated by the running program. Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.

2. Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

Thus, most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently. The most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data. Most programs (system and application) are stored on a disk until they are loaded into memory. Many programs then use the disk as both the source and the destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system, as we discuss in Chapter 10.In a larger sense, however, the storage structure that we have described consisting of registers, main memory, and magnetic disks is only one of many possible storage systems. Others include cache memory, CD-ROM, magnetic tapes, and so on. Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, cost, size, and volatility. The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases. This trade-off is reasonable; if a given storage system were both faster and less expensive than another other properties being the same then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top four levels of memory in Figure 1.4 may be constructed using semiconductor memory. In addition to differing in speed and cost, the various storage systems are either volatile or non-volatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **non-volatile storage** for safe keeping. In the hierarchy shown in Figure 2.2, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are non-volatile.

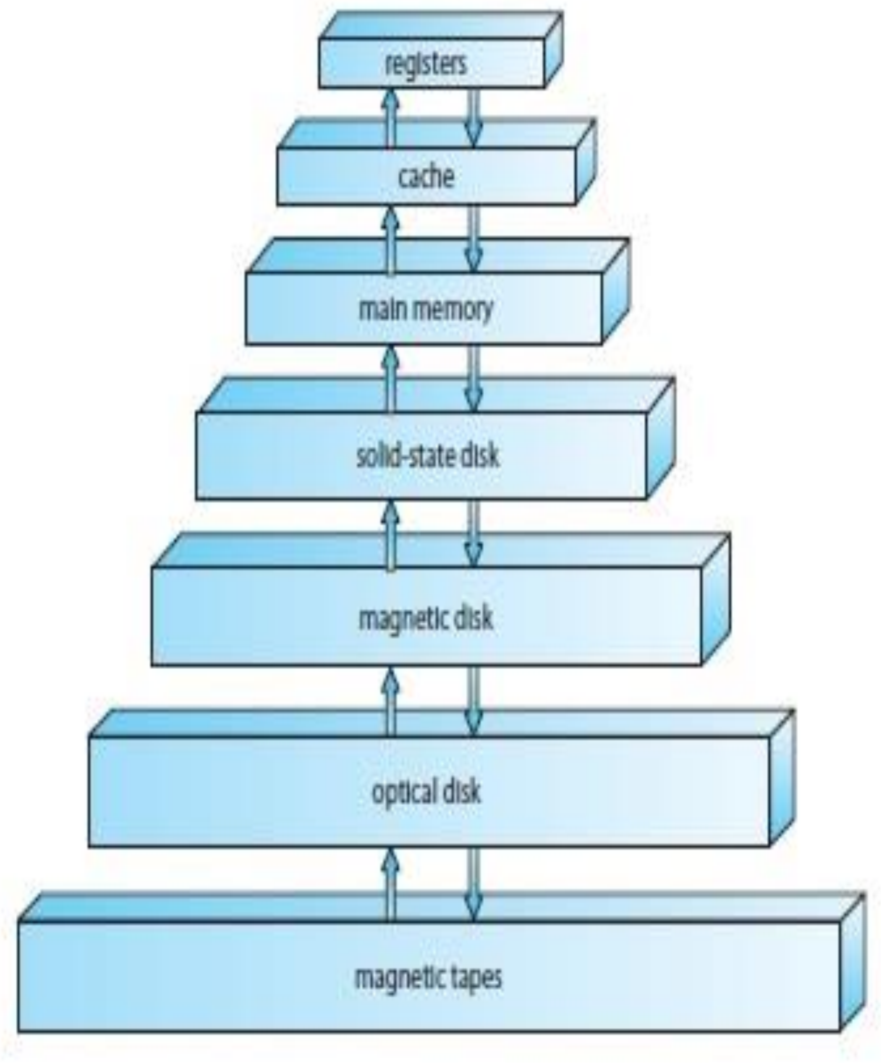


Figure 2-2 Storage-device hierarchy.

Solid-state disks have several variants but in general are faster than magnetic disks and are non-volatile. One type of solid-state disk stores data in a large DRAM array during normal operation but also contains a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, this solid-state disk's controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into RAM. Another form of solid-state disk is flash memory, which is popular in cameras and **personal digital assistants (PDAs)**, in robots, and increasingly for storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of non-volatile storage is **NVRAM**, which is DRAM with battery backup power. This memory can be as fast as DRAM and (as long as the battery lasts) is non-volatile. The design of a complete memory system must balance all the factors just discussed: it must use only as much expensive memory as necessary while providing as much in-expensive, non-volatile memory as possible. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

2.4.Hardware Protection

To improve system utilization, the O.S began to share system resources among several programs simultaneously. Multi programming put several programs in memory at the same time. This sharing created both improved utilization and increased problems. When the system was run without sharing an error in a program could cause problems for only the one program that was running. With sharing many process could be affected by a bug in one program.

2.4.1. Dual-Mode Operation

To ensure proper operation we must protect the O.S and all programs and their data from any malfunctioning program. Protection is needed for any shared resource. The approach taken is to H/W support to allow as differentiating among various modes of executions. Therefore we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit called mode bit is added to H/W to indicate the current mode; monitor (0) or user (1). With the mode bit we are able to distinguish between an execution that is done on behalf of the O.S, and one that is done on behalf of the user. The dual mode of operation provides us with the means for protecting the O.S from errant users and errant users from one another. The H/W allows privileged instructions to be executed in only monitor mode.

2.4.2. I/O Protection

To prevent a user from performing illegal I/O we define all I/O instructions to be privileged instructions. Thus user cannot issue I/O instructions directly they must do it through the O.S. For I/O protection to be complete we must be sure that a user program can never gain control of the Computer in monitor mode.

2.4.3. Memory Protection

To ensure correct operation we must protect the interrupt vector from modification by a user program. Also we must protect the interrupt service routines in the O.S from modification. What we need to separate each program's memory space is an ability to determine the range of legal addresses that the program may access, and to protect the memory outside that space. We can provide this protection by using two registers usually a base and a limit as illustrated in figure 2.3.

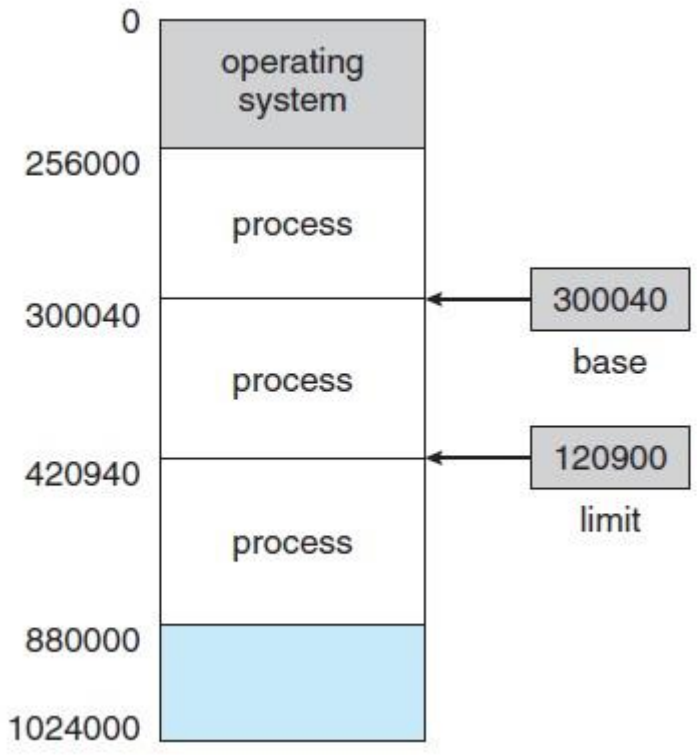


Figure 2-3 A base and a limit register define a logical address space

The base register holds the smallest legal physical memory address; the limit register contains the size of the range. For example if the base register holds 300040 and limit register is 120900 then the program can legally access all addresses from 300040 through 420940 inclusive.

The CPU H/W comparing every address generated in user mode with registers accomplishes this protection. Any attempt by a program executing in user mode to access monitor memory or other user's memory or other users memory results in a trap to the monitor which treats the attempt as a fatal error (figure 2.4).This scheme prevents the user program from modifying the code or data structures of either the O.S or other users.

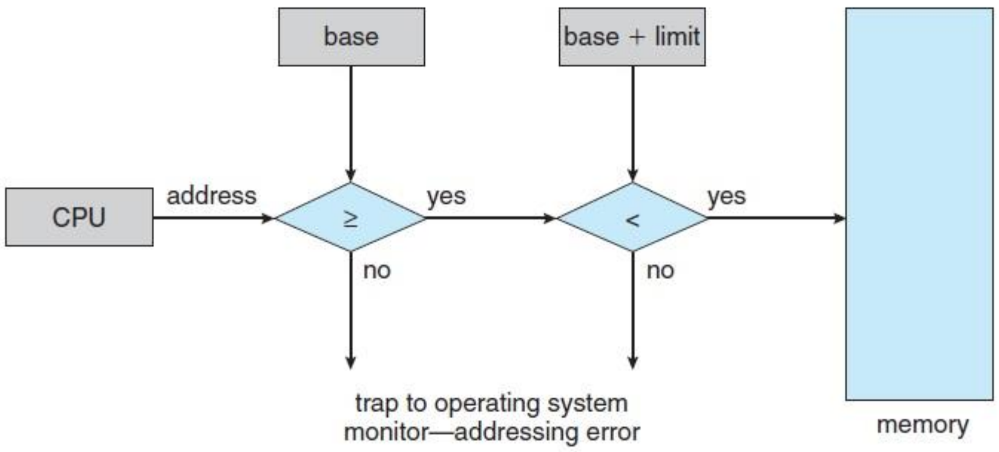


Figure 2-4 Hardware address protection with base and limit registers

The base and limit registers can be loaded by only the O.S which uses a special privileged instruction. Since privileged instructions can be executed in only monitor mode therefore only O.S can load the base and limit registers. This scheme allows the monitor to change the value of the registers but prevents user programs from changing the registers contents.

2.4.4. CPU Protection

The third piece of the protection is ensuring that the O.S maintains control, we must prevent a user program from an infinite loop, and never returning control to the O.S. To achieve this goal we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (1/60 second) or variable (from 1 millisecond to 1 second). To control the timer the O.S sets the counter, according to fixed-rate clock. Every time that the clock ticks the counter is decremented. When the counter reaches (0) on interrupt occurs, and control transfers automatically to the O.S, which may treat the interrupt as a fatal error or may give the program more time.

2.5. System Calls

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions. Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design. One approach is for the program to ask the user for the names. In an interactive system, this approach will require a sequence of system calls, first to write a prompting message on the screen and then to read from the keyboard the characters that define the two files. On mouse-based and icon-based systems, a menu of file names is usually displayed in a window. The user can then use the mouse to select the source name, and a window can be opened for the destination name to be specified. This sequence requires many I/O system calls.

Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call. Possible error conditions for each operation can require additional system calls. When the program tries to open the input file, for example, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console (another sequence of system calls) and then terminate abnormally (another system call). If the input file exists, then we must create a new output file. We may find that there is already an output file with the same name. This situation may cause the program to abort (a system call), or we may delete the existing file (another system call) and create a new one (yet another system call). Another option, in an interactive system, is to ask the user (via a sequence of system calls to output the prompting message and to read the response from the terminal) whether to replace the existing file or to abort the program.

When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call). Each read and write must return status information regarding various possible error conditions. On input, the program may find that the end of the file has been reached or that there was a hardware failure in the read (such as a parity error). The write operation may encounter various errors; depending on the output device (for example, no more disk space).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call).

Chapter Three

3.1. Process Management

A process can be thought of as a program in execution. A process will need certain resources such as CPU time, memory, files, and I/O devices to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

Early C/S allowed only one program to be executed at a time. This program had complete control of the system and had access to all of the system resources. Today C/S allows multiple programs to be loaded into memory and to be executed concurrently. The more complex the O.S the more it is expected to do on behalf of its users. A system therefore consists of a collection of processes:

O.S processes executing system code and user processes executing user code. By switching the CPU between processes the O.S can make the C/S more productive.

3.2. Process Concept

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes jobs, whereas a time-shared system has user programs, or tasks. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package. And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes. The terms job and process are used almost interchangeably in this text.

Although we personally prefer the term process, much of operating-system theory and terminology was developed during a time when the major activity of operating systems was job processing. It would be misleading to avoid the use of commonly accepted terms that include the word job (such as job scheduling) simply because process has superseded job. The execution of a process must progress in a sequential fashion. A process is more than the program code: sometimes known as the Text section, the value of program counter and the contents of the processor's registers.

3.3. Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

• **New**. The process is being created.

• **Running**. Instructions are being executed.

• **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

• **Ready**. The process is waiting to be assigned to a processor.

• **Terminated**. The process has finished execution.

These names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems also more finely delineate process states. It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting,* however. The state diagram corresponding to these states is presented in Figure 3.1.

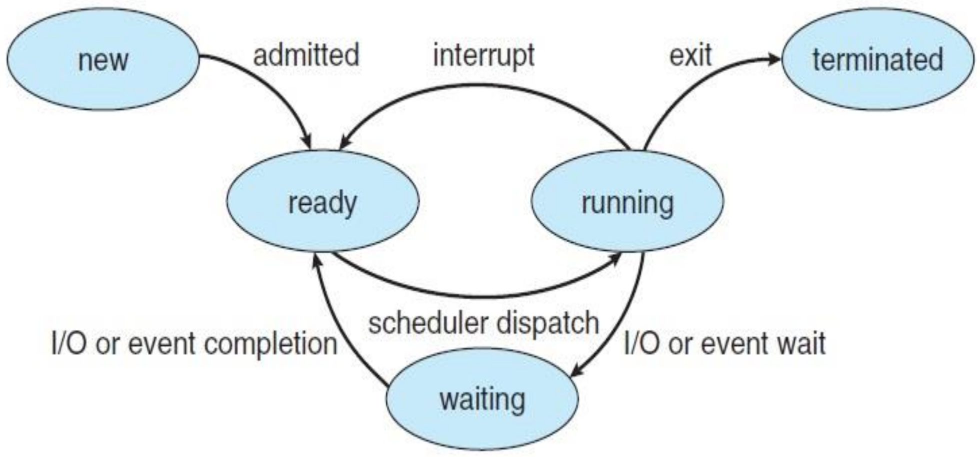


Figure 3-1 Diagram of process state

3.4. Process Control Block

A Process Control Block (PCB) also called a task control block represents each process in the O.S. A PCB is shown in the following figure 3.2. It contains many pieces of information associated with a specific process including these:

- Process state: It may be new, ready, running, waiting, halted and so on.

- Program counter: The address of the next instruction to be executed for this process.
- CPU registers: They include accumulators, index registers, stack pointer, and any general-purpose registers plus and condition-code information.
- CPU scheduling Information: It includes a process priority, pointers to scheduling queues.
- Memory information management: It may include the value of the base and limit registers, the page tables... etc.
- Accounting information: It includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- I/O status information: It includes the list of I/O devices allocated to this process, a list of open files, and so on.

The PCB simply serves as the repository for any information that may vary from process to process.

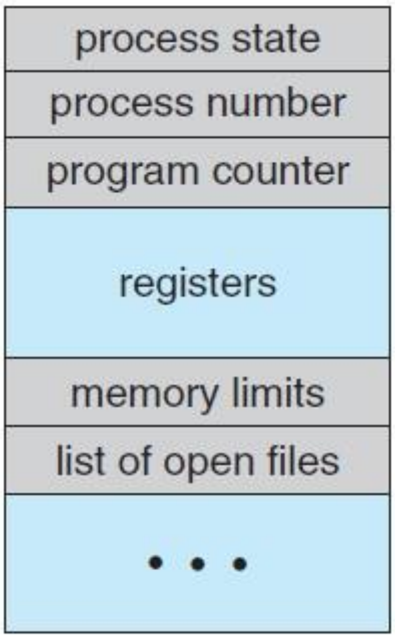


Figure 3-2 Process control block (PCB)

3.5. Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

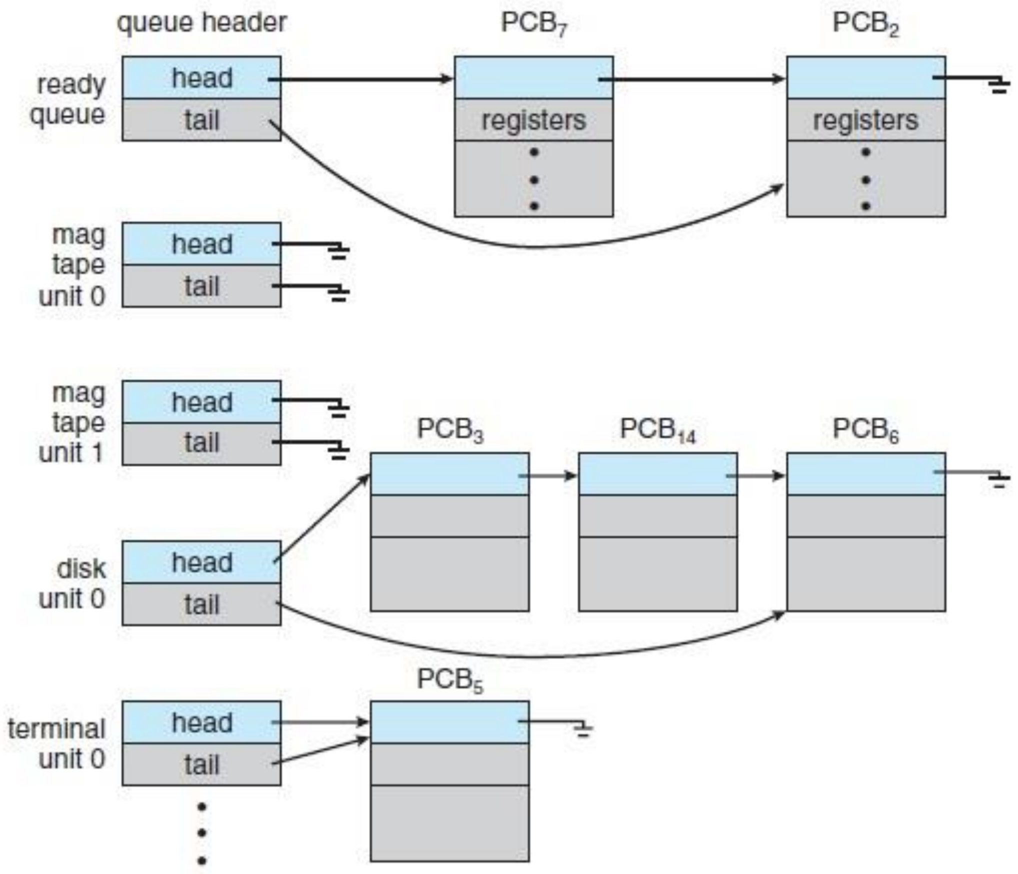


Figure 3-3 the ready queue and various I/O device queues

3.6. Scheduling Queues

- As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.

- The processes that are residing in memory and are ready and waiting to execute are kept on a list called the Ready queue.
- The queue is generally stored as a linked list the queue header contains pointers to the first and last PCB's in that list.
- Each PCB has a pointer field that points to the next process in the queue. Each device has its own device queue (Figure 3.4).

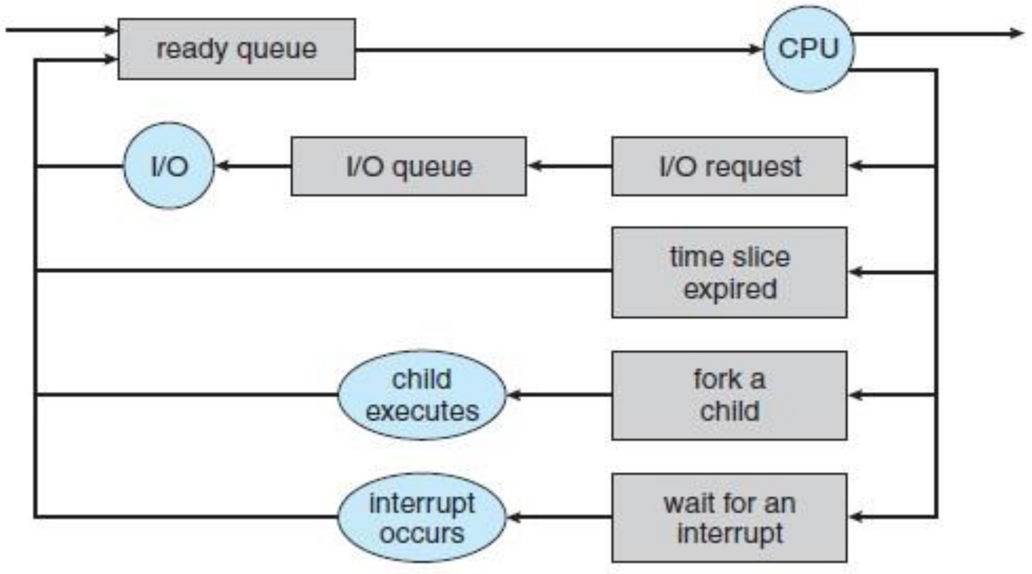


Figure 3-4 Queuing-diagram representation of process scheduling

- There are also other queues in the system; such as a list of processes waiting for a particular I/O device is called a device queue.
- A new process is initially put in the ready queue waits until it is selected for execution (or dispatched) and is given the CPU.
- Once the process is allocated the CPU and is executing one of several events could occur:

a. The process could issue an I/O request and then be placed in an I/O queue.

b. The process could create a new sub-process and wait for its termination.

c. The process could be removed forcibly from the CPU as a result of an interrupt and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

3.7. Scheduling levels

A process migrates between the various scheduling queues throughout its lifetime. The O.S must select processes from these queues in some fashion. The appropriate scheduler carries out the selection process. There are three levels (terms) of scheduling:

3.7.1. Long-term scheduler

The long-term scheduler or (Job scheduler) selects processes from the job pool on the disk and loads them into memory for execution. The long-term scheduler execute much less frequently there may be minutes between the creation of new processes in the system. The L.T.S control represents the degree of multi programming (The number of processes in memory). If the degree of multi programming is stable, than the average rate of processes creation must be equal to the average departure rate of processes leaving the system.

It is important that the L.T.S .make a careful selection. In general most processes can be described as either I/O bound or CPU bound.

- An I/O bound process is one that spends more of its time doing I/O than it spends doing computations.
- A CPU-bound process is one that generates I/O requests infrequently, using more of its time doing computation than an I/O-bound process.

The L.T.S selects a good process mix of I/O-bound and CPU-bound processes.

3.7.2. The short-term scheduler (or CPU Scheduler)

It is selects from among the processes that are ready to execute and allocates the CPU to one of them. The S.T.S must select a new process for the CPU quite frequently. Often the S.T.S must be very fast. If it takes 10 milliseconds to decide to executes a process for 100 milliseconds then 10/(100+10) = 9% of the CPU used (wasted) for scheduling the work. If all processes are I/O bound the ready queue will almost be empty and the S.T.S will have little to do. If all processes are CPU-bound the waiting queue will almost be empty. The system with the best performance will have a combination of CPU bound and I/0-bound processes.

3.7.3. The medium-term scheduler

Some O.S such as time-sharing systems may introduce an additional intermediate level of scheduling. The key behind the M.T.S is that sometimes it can be advantageous to remove processes from memory and thus to reduce the degree of multiprogramming. The process can be swapped out and swapped in later by the M.T.S swapping may be necessary to improve the process mix. The figure 3.5 shows the M.T.S.

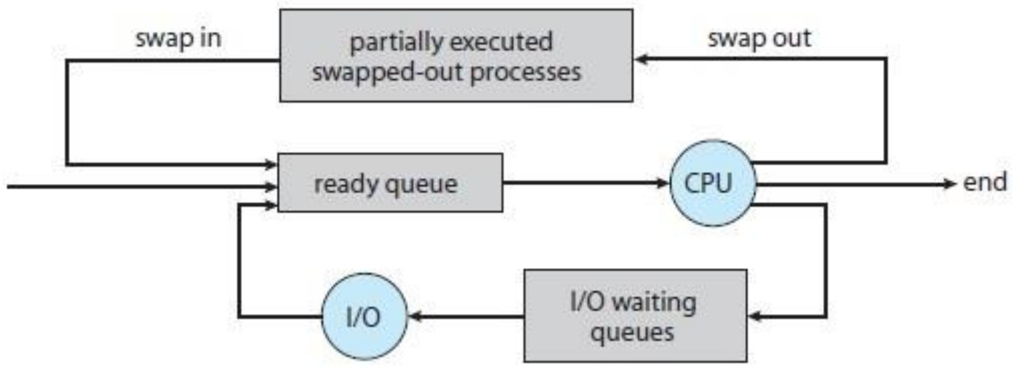


Figure 3-5 Addition of medium-term scheduling to the queuing diagram

3.8. Context Switch

- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch.
- Context-switch time is pure overhead because the system does no useful work while switching.
- The more complex the O.S the more work must be done during a context switch.

3.9. Operations on Processes

- O.S that mange processes must be able to perform certain operation on and with processes. These include: create, destroy, suspend, resume, change a process priority, block a process, wake up a process dispatch a process, enable a process to communicate with another process.

- Creating a process involves many operations including: name a process, insert it in the ready queue, determine the process initial priority, create the PCB and allocate the process's initial resources.
- A process may create a new process. If it does the creating process is called the parent process and the created process is called the child process.

When a process creates a new process two possibilities exist in terms of execution:

a. The parent continues to execute concurrently with its children.

b. The parent waits until some or all of its children have terminated.

- A process terminates when it finishes executing its last statement and asks the O.S to delete it by using the existing system call.
- A parent may terminate the execution of one of its children for a variety at reasons such as:

a. The child has exceeded its usage of some of the resources it has been allocated.

b. The task assigned to the child is no longer required.

c. The parent is finished and the O.S does not allow a child to continue if its parent terminated.

3.10. Cooperating processes

The concurrent processes executing in the O.S may be either independent processes or cooperating processes. A process is independent if it cannot affect or be not affected by the other processes executing in the system. Any process that does not share or any data (temporary or persistent) with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system or any process that share data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

1. Information sharing. 2. Computation speedup.

3. Modularity: Dividing the system functions into separate processes.

4. Convenience, Many tasks to work on at one time, a user may be editing, printing and compiling in parallel.

To illustrate the concept of cooperating processes let us consider the producer-consumer problem as an example of cooperating processes. A produce process produces information that is consumed by a consumer process. For example a print program produces characters that are consumed by the printer driver. To allow producer and consumer to run concurrently we must have a buffer of item that can be filled by the producer and emptied by the consumer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized. The consumer must wait until an item is produced (the buffer is empty) and the producer must wait if the buffer is full.

In the bounded-buffer and be one solution for the producer and consumer processes share the following variables:

```
var n;
type item = ......;
var buffer: array [0 .. n-l] of item;
in out: 0 .. n-1
```

Within, out initialized to the value 0. The shared buffer is implemented as a circular array with two logical pointers: in and out.

in points to the next free position in the buffer; out points to the first full position in the buffer.

The buffer is empty when in=out; the buffer is full when in+1 mod n=out.

The producer process has a local variable nextp in which the new item to be produced is stored.

```
Repeat
……
produce an item in nextp
……
while in+1 mod n=out do no-op;
```

```
buffer [in] :=nextp;
in:=in+l mod n;
until false;
```

The consumer process has a local variable nextc in which the item to be consumed is stored;

```
repeat do-nothing instruction
while in=out do no-op;
nextc:= buffer [out],
out:=out+1 mod n;
……
consume the item in nextc; until false;
```

3.11. Thread structure

A thread sometimes called light weight process (LWP) is a basic unit of CPU utilization and consists of a program counter, a register set, and a stack space. It shares with peer threads its code section, data section and O.S resources such as open files and signals collectively known as a task. A traditional or heavy weight process is equal to a task with one thread.

Threads can be in one of several states ready, blocked, running, or terminated. Threads can create child threads if one thread is blocked another thread can run. Unlike processes threads are not independent of one another, because all threads can access in the task.

3.12. Interrupt Processing

An interrupt is an event that alters the sequence in which a processor executes instructions. The H/W of C/S generates the interrupt. When an interrupt occurs the following actions will be taken:

a. The O.S gains control.

b. The O.S saves the state of interrupted process in its PCB.

c. The O.S analyzes the interrupt and passes control to the appropriate routine to handle the interrupt.

d. The interrupt handler routine processes the interrupt.(IHR)

e. The state of the interrupted process (or some other next process) is restored.

f. The interrupted process (or some other next process) executes.

An interrupt may be specifically initiated by a running process (in which case it is often called a trap and said to be synchronous with the operation of the process). Or it may be caused by some event that may or may not be related to the running process. It is said to be asynchronous with the operation of the process.

3.13. Interrupt Classes (types)

There are six interrupt classes. These are:

3.13.1. SVC (Supervisor call) interrupts

A running process that executes the SVC instruction such as initiates these:

- I/O request. - Obtaining more storage. - Communicating with user operator.

3.13.2. I/O interrupts:

There are initiated by the I/O H/W. such as:

- An I/O operation completes.

- An I/O error occurs.

- When a device is made ready.

3.13.3. External interrupts:

These are caused by various events including: the expiration of a quantum on an interrupting clock.

- Pressing of the console's interrupt key by the operator.

- Receipt of a signal from another processor.

3.13.4. Restart interrupts:

These occur when the operator:

- Presses the console's restart bottom.

- When a restart signal processor instruction arrives from another processor on a multi-processor system.

3.13.5. Program checks interrupt:

These are caused by many problems such as:

- Divide by zero.
- Arithmetic overflow.
- Data is in the wrong format.
- Attempt to execute invalid operation code.
- Attempt to reference a memory location beyond the limits of main memory.
- Attempt to execute a privileged instruction.
- Attempt to reference a protected resource.

Chapter Four

4.1. CPU Scheduling

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

4.2. CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 4.1).

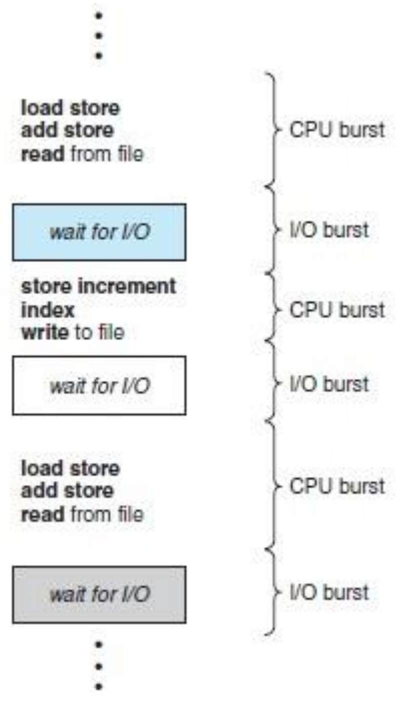


Figure 4-1 Alternating sequence of CPU and I/O bursts

4.3. CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler**, or CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process. Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

4.4. Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling. Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

4.5. Dispatcher

Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

• Switching context.

• Switching to user mode.

• Jumping to the proper location in the user program to restart that program.

The dispatcher should be as fast as possible, since it is invoked during every

process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

4.6. Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

• **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

• **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

• **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

• **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

• **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

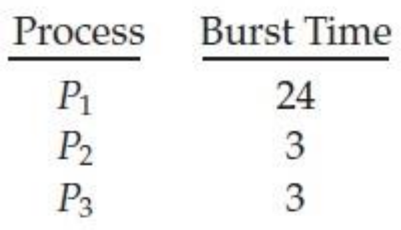
It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

4.7. Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms. In this section, we describe several of them.

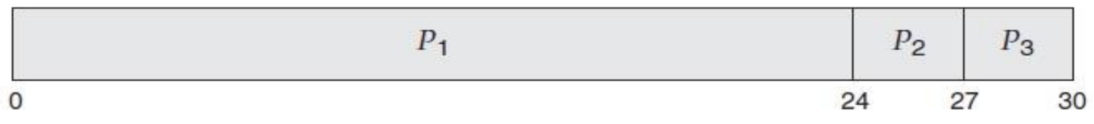
4.7.1. First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

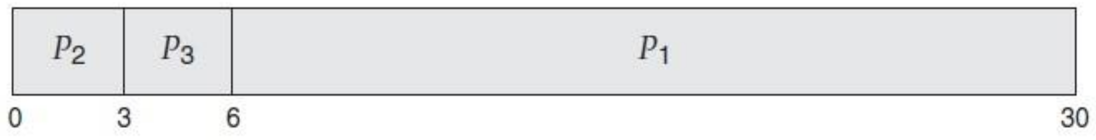


Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order *P*1, *P*2, *P*3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process *P*1, 24 milliseconds for process *P*2, and 27 milliseconds for process *P*3. Thus, the average waiting time is (0+ 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order *P*2, *P*3, *P*1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU-bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

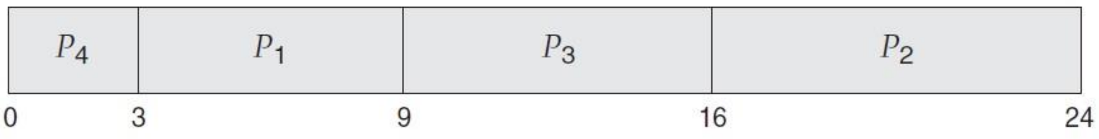
Note also that the FCFS scheduling algorithm is nonpreemptive. Once, the CPU has been allocated to a process, that process keeps the CPU until its releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

4.7.2. Shortest-Job-First Scheduling

A different approach to CPU scheduling is the **shortest-job-first (SJF)** scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the ***shortest-next-CPU-burst*** algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
$P1$	6
$P2$	8
$P3$	7
$P4$	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



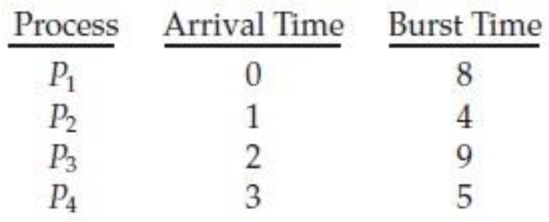
The waiting time is 3 milliseconds for process $P1$, 16milliseconds for process $P2$, 9milliseconds for process $P3$, and 0milliseconds for process $P4$. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases. The real difficulty with the SJF algorithm knows the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. In this situation, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response but too low a value will cause a time-limit-exceeded error and require resubmission. SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

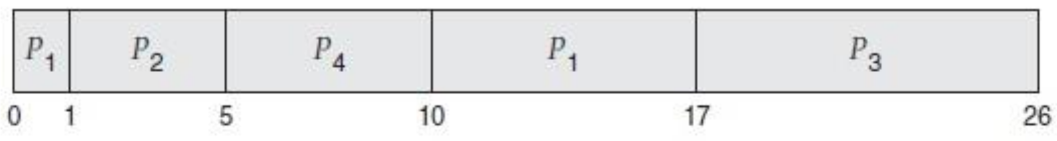
The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:



Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process $P1$ is started at time 0, since it is the only process in the queue. Process $P2$ arrives at time 1. The remaining time for process $P1$ (7 milliseconds) is larger than the time required by process $P2$ (4 milliseconds), so process $P1$ is preempted, and process $P2$ is scheduled. The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

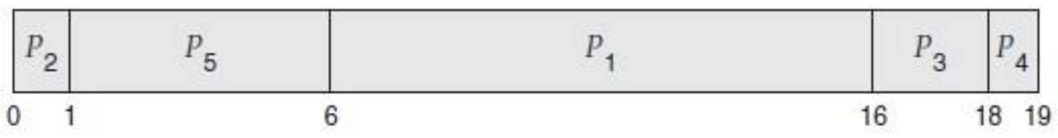
4.7.3. Priority Scheduling

The SJF algorithm is a special case of the general **priority-scheduling** algorithm. Apriority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority. As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P1$, $P2$, · · ·, $P5$, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors. Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Generally, one of two things will happen. Either the process will eventually be run (at 2 A.M. Sunday, when the system is finally lightly loaded), or the computer system will eventually crash and lose all unfinished low-priority processes.

A solution to the problem of in definite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

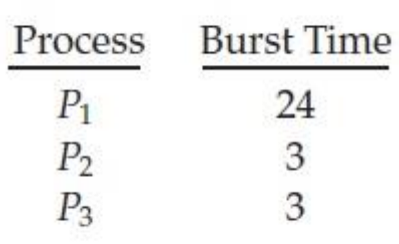
4.7.4. Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

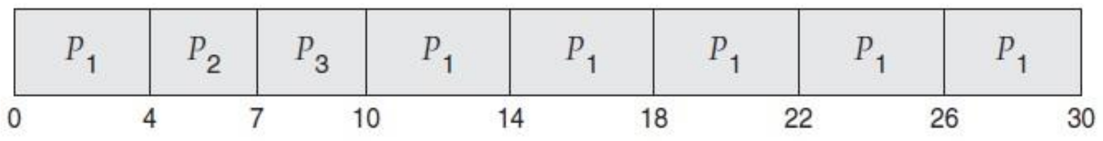
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue. The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:



Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process *P*1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P*2. Process *P*2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process *P*3. Once each process has received 1 time quantum, the CPU is returned to process *P*1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. *P*1 waits for 6 milliseconds (10 - 4), *P*2 waits for 4 milliseconds, and *P*3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds. The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 4.2).

Thus, we want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

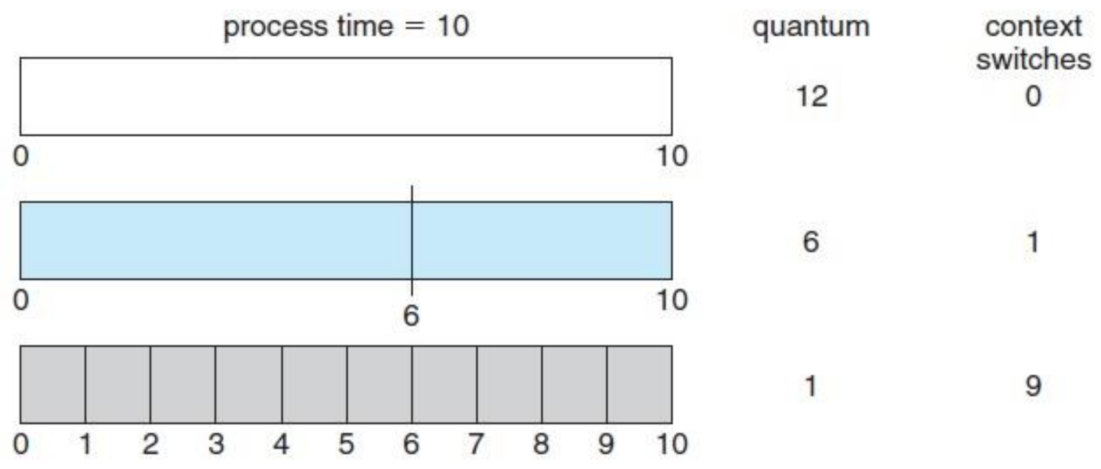


Figure 4-2 How a smaller time quantum increases context switches

Turnaround time also depends on the size of the time quantum. As we can see from Figure 4.2, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

4.7.5. Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground** (interactive) processes and **background** (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes. A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues (Figure 6.6). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue. Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes

2. Interactive processes

3. Interactive editing processes

4. Batch processes

5. Student processes

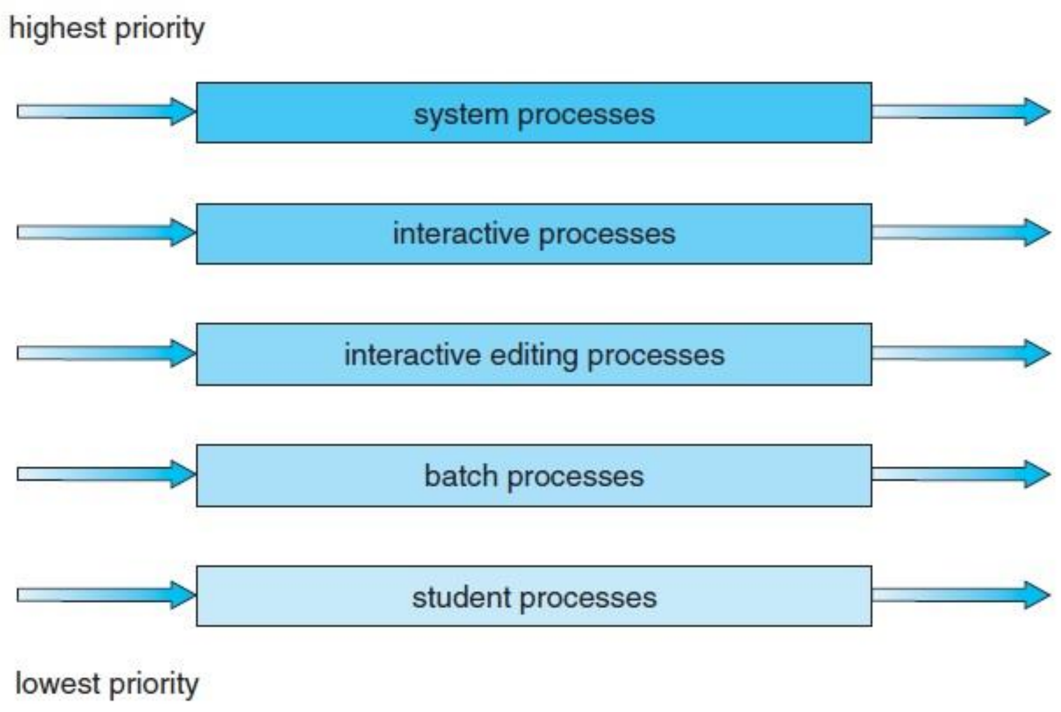


Figure 4-3 Multilevel queue scheduling

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

4.7.6. Multilevel Feedback Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature. This setup has the advantage of low scheduling overhead, but it is inflexible.

The **multilevel feedback queue** scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 4.4). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

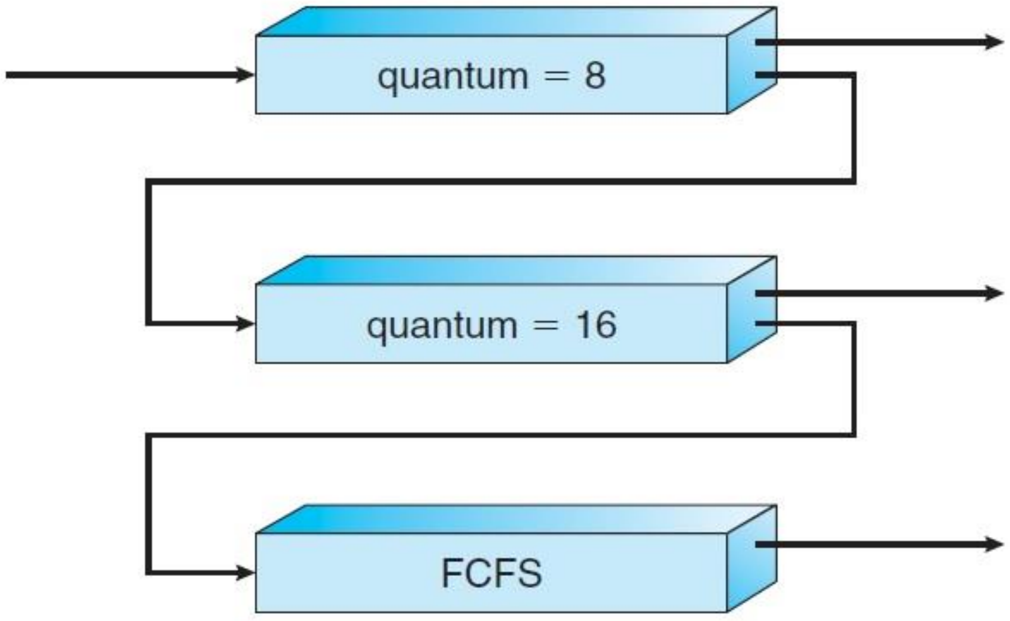


Figure 4-4 Multilevel feedback queues.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

• The number of queues

• The scheduling algorithm for each queue

• The method used to determine when to upgrade a process to a higher priority queue

• The method used to determine when to demote a process to a lower priority queue

• The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

4.8. Algorithm Evaluation

How do we select a CPU-scheduling algorithm for a particular system? there are many scheduling algorithms, each with its own parameters. As a result, selecting an algorithm can be difficult. The first problem is defining the criteria to be used in selecting an algorithm. criteria are often defined in terms of CPU utilization, response time, or throughput. To select an algorithm, we must first define the relative importance of these elements. Our criteria may include several measures, such as these:

• Maximizing CPU utilization under the constraint that the maximum response time is 1 second

• Maximizing throughput such that turnaround time is (on average) linearly proportional to total execution time

Chapter Five

5.1. Process Synchronization

5.1.1. Background

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages.

We've already seen that processes can execute concurrently or in parallel. The role of process scheduling and has been described how the CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true)

        /* produce an item and put in nextProduced

        while (count == BUFFER_SIZE)

                ; // do nothing

buffer [in] = nextProduced;
```

```
        in = (in + 1) % BUFFER_SIZE;
        count++; }
```

Consumer

```
while (1)
        { while (count == 0)
                ; // do nothing
                nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
count--;
/* consume the item in nextConsumed }
```

5.1.2. Race Condition

If there are several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in operating systems as different parts of the system manipulate resources. Furthermore, as we have emphasized in earlier chapters, the growing importance of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads which are quite possibly sharing data are running in parallel on different processing cores. Clearly we want any changes that result from such activities not to interfere with one another.

count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

count-- could be implemented as

register2 = count

register2 = register2 - 1

count = register2

Consider this execution interleaving with “count = 5” initially:

S0: producer execute register1 = count {register1 = 5}

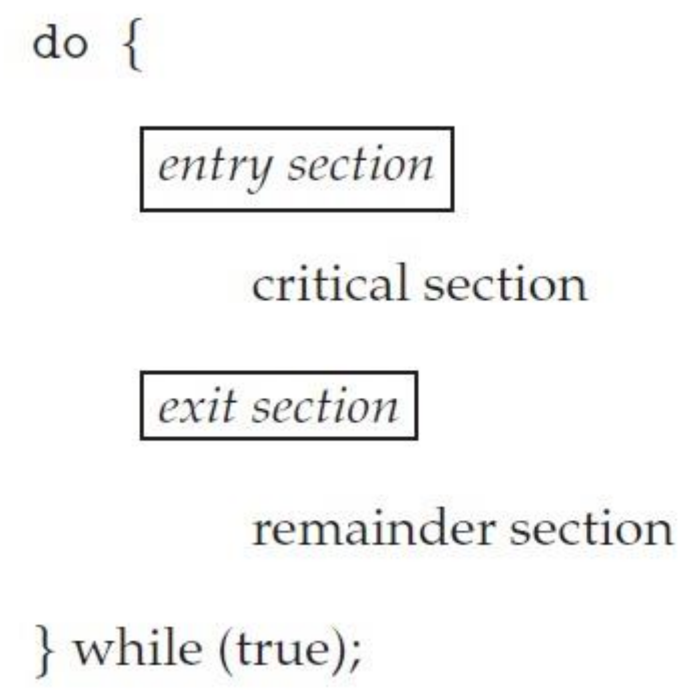
S1: producer execute register1 = register1 + 1 {register1 = 6}

S2: consumer execute register2 = count {register2 = 5}

S3: consumer execute register2 = register2 - 1 {register2 = 4}

S4: producer execute count = register1 {count = 6 }

S5: consumer execute count = register2 {count = 4}



```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Figure 5-1 General structure of a typical process Pi

5.1.3. The Critical-Section Problem

We begin our consideration of process synchronization by discussing the so called critical-section problem. Consider a system consisting of *n* processes {*P*0, *P*1, ..., *Pn*−1}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process *Pi* is shown in Figure 5.1. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Chapter Six

6.1. Deadlock

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**. Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”

Although some applications can identify programs that may deadlock, operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs. Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and an emphasis on long-lived file and database servers rather than batch systems.

6.1.1. System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances.

If a process requests an instance of a resource type, the allocation of any instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly. For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. In other words, a process cannot request three printers if the system has only two.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).

3. Release. The process releases the resource.

For each use of a kernel-managed resource by a process or thread, the operating system checks to make sure that the process has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example files). However, other types of events may result in deadlocks.

To illustrate a deadlocked state, consider a system with three CD RW drives. Suppose each of three processes holds one of these CDRW drives. If each process now requests another drive, the three processes will be in a deadlocked state. Each is waiting for the event “CD RW is released,” which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.

Deadlocks may also involve different resource types. For example, consider a system with one printer and one DVD drive. Suppose that process *Pi* is holding the DVD and process *Pj* is holding the printer. If *Pi* requests the printer and *Pj* requests the DVD drive, a deadlock occurs.

Developers of multithreaded applications must remain aware of the possibility of deadlocks. The locking tools presented in Chapter 5 are designed to avoid race conditions. However, in using these tools, developers must pay careful attention to how locks are acquired and released. Otherwise, deadlock can occur.

6.2. Deadlock Characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting. Before we discuss the various methods for dealing with the deadlock problem, we look more closely at features that characterize deadlocks.

6.2.1. Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait. A set {*P*0, *P*1, ..., *Pn*} of waiting processes must exist such that *P*0 is waiting for a resource held by *P*1, *P*1 is waiting for a resource held by *P*2, ..., *Pn*−1 is waiting for a resource held by *Pn*, and *Pn* is waiting for a resource held by *P*0. We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

6.2.2. Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices *V* is partitioned into two different types of nodes: $P = \{P1, P2, ..., Pn\}$, the set consisting of all the active processes in the system, and $R = \{R1, R2, ..., Rm\}$, the set consisting of all resource types in the system.

A directed edge from process *Pi* to resource type *Rj* is denoted by $Pi \rightarrow Rj$; it signifies that process *Pi* has requested an instance of resource type *Rj* and is currently waiting for that resource. A directed edge from resource type *Rj* to process *Pi* is denoted by $Rj\rightarrow Pi$; it signifies that an instance of resource type *Rj* has been allocated to process *Pi*. A directed edge $Pi \rightarrow Rj$ is called a **request edge**; a directed edge $Rj \rightarrow Pi$ is called an **assignment edge**.

Pictorially, we represent each process *Pi* as a circle and each resource type *Rj* as a rectangle. Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle *Rj* , whereas an assignment edge must also designate one of the dots in the rectangle.

When process *Pi* requests an instance of resource type *Rj*, a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted. The resource-allocation graph shown in Figure 6.1 depicts the following situation.

• The sets *P*, *R*, and *E*:

◦ $P = \{P1, P2, P3\}$

◦ $R = \{R1, R2, R3, R4\}$

◦ $E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$

• Resource instances:

◦ One instance of resource type *R*1

◦ Two instances of resource type *R*2

◦ One instance of resource type *R*3

◦ Three instances of resource type *R*4

• Process states:

◦ Process *P*1 is holding an instance of resource type *R*2 and is waiting for an instance of resource type *R*1.

◦ Process *P*2 is holding an instance of *R*1 and an instance of *R*2 and is waiting for an instance of *R*3.

◦ Process *P*3 is holding an instance of *R*3.

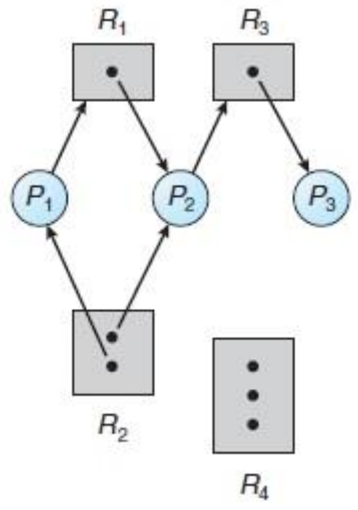


Figure 6-1 Resource-allocation graph

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock. If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock. To illustrate this concept, we return to the resource-allocation graph depicted in Figure 6.1. Suppose that process *P*3 requests an instance of resource type *R*2. Since no resource instance is currently available, we add a request edge *P*3→ *R*2 to the graph (Figure 6.2). At this point, two minimal cycles exist in the system:

*P*1 → *R*1 → *P*2 → *R*3 → *P*3 → *R*2 → *P*1

*P*2 → *R*3 → *P*3 → *R*2 → *P*2

Processes *P*1, *P*2, and *P*3 are deadlocked. Process *P*2 is waiting for the resource *R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process *P*2 to release resource *R*1.

Now consider the resource-allocation graph in Figure 6.3. In this example, we also have a cycle:

*P*1 → *R*1 → *P*3 → *R*2 → *P*1

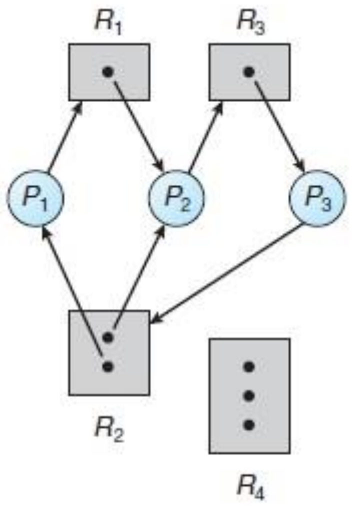


Figure 6-2 Resource-allocation graph with a deadlock

However, there is no deadlock. Observe that process *P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

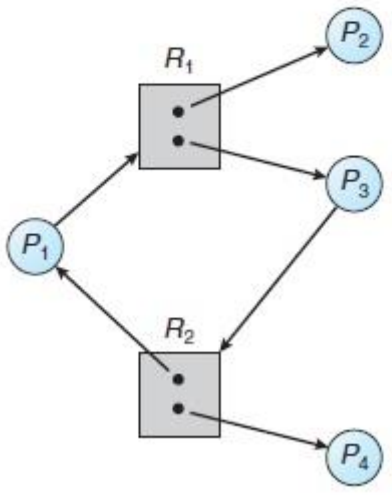


Figure 6-3 Resource-allocation graph with a cycle but no deadlock

6.3. Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

• We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.

• We can allow the system to enter a deadlocked state, detect it, and recover.

• We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including Linux and Windows. It is then up to the application developer to write programs that handle deadlocks.

To ensure that deadlocks never occur, the system can use either deadlock prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made. **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise. In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock (if a deadlock has indeed occurred).

In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

Although this method may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems, as mentioned earlier. Expense is one important consideration. Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per year), the extra expense of the other methods may not seem worthwhile. In addition, methods used to recover from other conditions may be put to use to recover from deadlock. In some circumstances, a system is in a frozen state but not in a deadlocked state. We see this situation, for example, with a real-time process running at the highest priority (or any process running on a nonpreemptive scheduler) and never returning control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

6.4. Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

6.4.1. Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be non-sharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file. A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

6.4.2. Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources. One protocol that we can use requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.

An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end.

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates. Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period.

In the example given, for instance, we can release the DVD drive and disk file, and then request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.

Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

6.4.3. No Preemption

The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted. In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as mutex locks and semaphores.

6.4.4. Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration. To illustrate, we let $R = \{R1, R2... Rm\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially request any number of instances of a resource type say, Ri. After that, the process can request instances of resource type Rj if and only if $F(Rj) > F(Ri)$. For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that a process requesting an instance of resource type Rj must have released any resources Ri such that $F(Ri) \geq F(Rj)$. Note also that if several instances of the same resource type are needed, a ***single*** request for all of them must be issued.

If these two protocols are used, then the circular-wait condition cannot hold. We can demonstrate this fact by assuming that a circular wait exists (proof by contradiction). Let the set of processes involved in the circular wait be $\{P0, P1, ..., Pn\}$, where Pi is waiting for a resource Ri, which is held by process $Pi+1$. (Modulo arithmetic is used on the indexes, so that Pn is waiting for a resource Rn held by $P0$.) Then, since process $Pi+1$ is holding resource Ri while requesting resource $Ri+1$, we must have $F(Ri) < F(Ri+1)$ for all i. But this condition means that $F(R0) < F(R1) < ... < F(Rn) < F(R0)$. By transitivity, $F(R0) < F(R0)$, which is impossible. Therefore, there can be no circular wait. We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. All requests for synchronization objects must be made in increasing order.

Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering. Also note that the function *F* should be defined according to the normal order of usage of the resources in a system. For example, because the tape drive is usually needed before the printer, it would be reasonable to define *F*(tape drive)<*F*(printer).

Although ensuring that resources are acquired in the proper order is the responsibility of application developers, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible. It is also important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a function that transfers funds between two accounts.

6.5. Deadlock Avoidance

Deadlock-prevention algorithms, as discussed in the previous section, prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur. Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested. For example, in a system with one tape drive and one printer, the system might need to know that process *P* will request first the tape drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the tape drive. With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

The various algorithms that use this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections, we explore two deadlock-avoidance algorithms.

6.5.1. Safe State

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**. A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if, for each *Pi*, the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with *j* <*i*. In this situation, if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished. When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, *Pi*+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe.*

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however (Figure 6.4). An unsafe state ***may*** lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs. The behaviour of the processes controls unsafe states.

To illustrate, we consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2. Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives. Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives. (Thus, there are three free tape drives.)

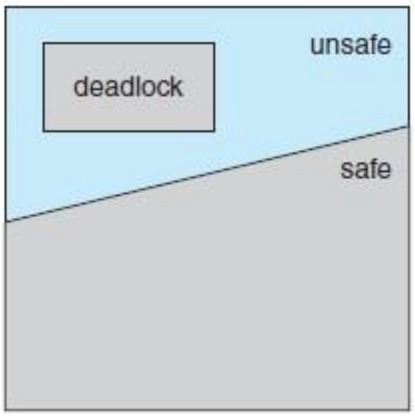


Figure 6-4 Safe, unsafe, and deadlocked state spaces

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition. Process *P*1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *P*0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P*2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

A system can go from a safe state to an unsafe state. Suppose that, at time *t*1, process *P*2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process *P*1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process *P*0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process *P*2 may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process *P*2 for one more tape drive. If we had made *P*2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state. In this scheme, if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

6.5.2. Resource-Allocation-Graph Algorithm

If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph that has been defined previously for deadlock avoidance. In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**. A claim edge $Pi \rightarrow Rj$ indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process *Pi* requests resource *Rj*, the claim edge $Pi \rightarrow Rj$ is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi*, the assignment edge $Rj \rightarrow Pi$ is reconverted to a claim edge $Pi \rightarrow Rj$.

Note that the resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $Pi \rightarrow Rj$ to be added to the graph only if all the edges associated with process *Pi* are claim edges.

Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge $Pi \rightarrow Rj$ to an assignment edge $Rj \rightarrow Pi$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where *n* is the number of processes in the system.

If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process *Pi* will have to wait for its requests to be satisfied. To illustrate this algorithm, we consider the resource-allocation graph of Figure 6.5. Suppose that *P*2 requests *R*2. Although *R*2 is currently free, we cannot allocate it to *P*2, since this action will create a cycle in the graph (Figure 6.6). A cycle, as mentioned, indicates that the system is in an unsafe state. If *P*1 requests *R*2, and *P*2 requests *R*1, then a deadlock will occur.

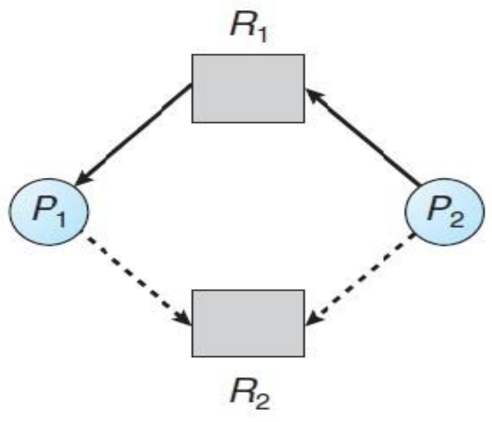


Figure 6-5 Resource-allocation graph for deadlock avoidance

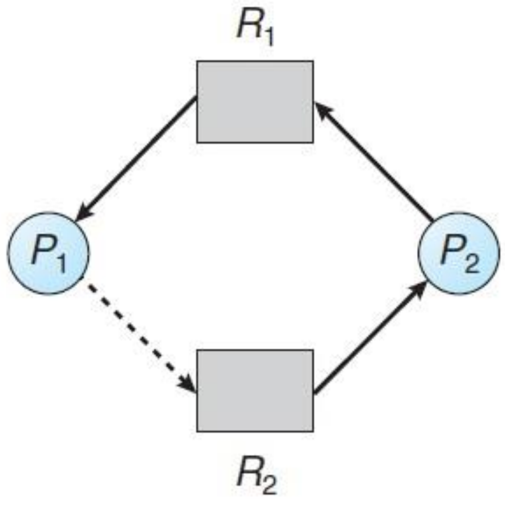


Figure 6-6 An unsafe state in a resource-allocation graph

6.5.3. Banker's Algorithm

The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm.** The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources. Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:

• **Available**. A vector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] = *k*, then *k* instances of resource type *Rj* are available.

• **Max**. An *n* × *m* matrix defines the maximum demand of each process. If ***Max***[*i*][*j*]=*k*, then process *Pi* may request at most *k* instances of resource type *Rj* .

• **Allocation**. An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] =*k,* then process *Pi* is currently allocated *k* instances of resource type *Rj* .

• **Need**. An *n* × *m* matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] =*k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task. Note that

Need[*i*][*j*] =***Max***[*i*][*j*]− ***Allocation***[*i*][*j*].

These data structures vary over time in both size and value. To simplify the presentation of the banker's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, ..., n$.

For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

We can treat each row in the matrices ***Allocation*** and ***Need*** as vectors and refer to them as ***Allocation****i* and ***Need****i* **.** The vector ***Allocation****i* specifies the resources currently allocated to process *Pi*; the vector ***Need****i* specifies the additional resources that process *Pi* may still request to complete its task.

6.5.3.1. Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize

Work = ***Available*** and ***Finish***[*i*] = ***false*** for $i = 0, 1, ..., n - 1$.

2. Find an index *i* such that both

a. ***Finish***[*i*] == ***false***

b. ***Need****i* ≤ ***Work***

If no such *i* exists, go to step 4.

3. ***Work*** = ***Work*** + ***Allocation****i*

Finish[*i*] = ***true***

Go to step 2.

4. If ***Finish***[*i*] == ***true*** for all *i*, then the system is in a safe state.

This algorithm may require an order of $m \times n2$ operations to determine whether a state is safe.

6.5.3.2. Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let ***Request****i* be the request vector for process *Pi* .

If ***Request****i* [*j*] == *k*, then

process *Pi* wants *k* instances of resource type *Rj*. When a request for resources is made by process *Pi*, the following actions are taken:

1. If ***Request****i* **≤*Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

Available = ***Available***–***Request****i* ;

Allocation*i* = ***Allocation****i* + ***Request****i* ;

Need*i* = ***Need****i* –***Request****i* ;

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for ***Requesti ,*** and the old resource-allocation state is restored.

6.5.3.3. An Illustrative Example

To illustrate the use of the banker's algorithm, consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B,* and *C.* Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances. Suppose that, at time *T*0, the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

The content of the matrix ***Need*** is defined to be ***Max*** − ***Allocation*** and is as follows:

	Need
	A B C
*P*0	7 4 3
*P*1	1 2 2
*P*2	6 0 0
*P*3	0 1 1
*P*4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence <*P*1,*P*3, *P*4, *P*2, *P*0> satisfies the safety criteria. Suppose now that process *P*1 requests one additional instance of resource type *A* and two instances of resource type *C,* so *Request*1 = (1,0,2). To decide whether this request can be immediately granted, we first check that *Request*1 ≤ ***Available***—that is, that (1,0,2) ≤ (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	***Need***	***Available***
	A B C	*A B C*	*A B C*
*P*0	0 1 0	7 4 3	2 3 0
*P*1	3 0 2	0 2 0	
*P*2	3 0 2	6 0 0	
*P*3	2 1 1	0 1 1	
*P*4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.

You should be able to see, however, that when the system is in this state, a request for (3,3,0) by *P*4 cannot be granted, since the resources are not available.

Furthermore, a request for (0,2,0) by *P*0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

We leave it as a programming exercise for students to implement the banker's algorithm.

6.6. Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

• An algorithm that examines the state of the system to determine whether

a deadlock has occurred

• An algorithm to recover from the deadlock

In the following discussion, we elaborate on these two requirements as they pertain to systems with only a single instance of each resource type, as well as to systems with several instances of each resource type. At this point, however, we note that a detection-and-recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

6.6.1. Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. More precisely, an edge from *Pi* to *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs. An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj* for some resource *Rq* . In Figure 6.7, we present a resource-allocation graph and the corresponding wait-for graph. As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to ***maintain*** the wait-for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of *n*2 operations, where *n* is the number of vertices in the graph.

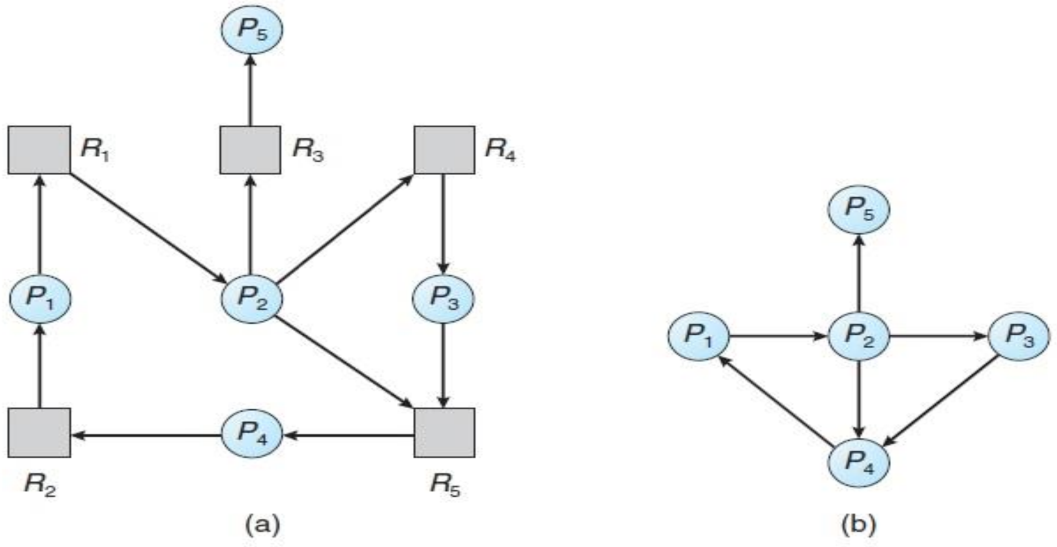


Figure 6-7 (a) Resource-allocation graph. (b) Corresponding wait-for graph

6.6.2. Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm:

• **Available**. A vector of length *m* indicates the number of available resources

of each type.

• **Allocation**. An *n* × *m* matrix defines the number of resources of each type currently allocated to each process.

• **Request**. An *n* × *m* matrix indicates the current request of each process.

If ***Request***[*i*][*j*] equals *k*, then process *Pi* is requesting *k* more instances of

resource type *Rj* .

The≤relation between two vectors is defined as in Banker's algorithm. To simplify

notation, we again treat the rows in the matrices ***Allocation*** and ***Request*** as

vectors; we refer to them as ***Allocationi*** and ***Requesti*** **.** The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm.

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available.*** For *i* = 0, 1, ..., *n–1*, if ***Allocationi*** ≠ 0, then ***Finish***[*i*] = ***false.*** Otherwise, ***Finish***[*i*] = ***true.***

2. Find an index *i* such that both

a. ***Finish***[*i*] == ***false***

b. ***Requesti*** ≤ ***Work***

If no such *i* exists, go to step 4.

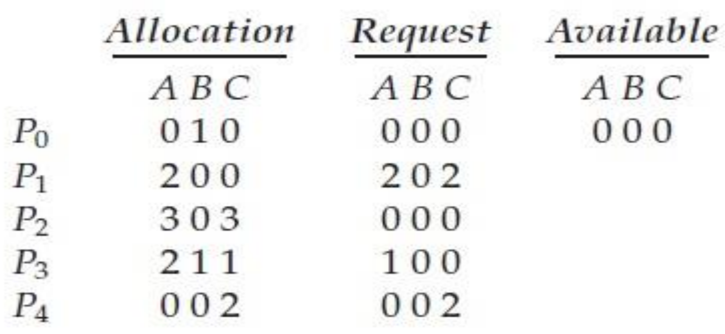
3. ***Work*** = ***Work*** + ***Allocationi***

Finish[*i*] = ***true***

Go to step 2.

4. If ***Finish***[*i*] ==***false*** for some *i*, 0≤*i*<*n*, then the system is in a deadlocked state. Moreover, if ***Finish***[*i*] == ***false,*** then process *Pi* is deadlocked.

This algorithm requires an order of $m \times n2$ operations to detect whether the system is in a deadlocked state. You may wonder why we reclaim the resources of process *Pi* (in step 3) as soon as we determine that ***Requesti*** ≤ ***Work*** (in step 2b). We know that *Pi* is currently ***not*** involved in a deadlock (since ***Requesti*** ≤ ***Work***). Thus, we take an optimistic attitude and assume that *Pi* will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked. To illustrate this algorithm, we consider a system with five processes *P*0 through *P*4 and three resource types *A*, *B*, and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has six instances. Suppose that, at time *T*0, we have the following resource-allocation state:



	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*P*0, *P*2, *P*3, *P*1, *P*4> results in ***Finish***[*i*] == *true* for all *i*.

Suppose now that process *P*2 makes one additional request for an instance of type *C*. The ***Request*** matrix is modified as follows:

	Request
	A B C
*P*0	0 0 0
*P*1	2 0 2
*P*2	0 0 1
*P*3	1 0 0
*P*4	0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process *P*0, the number of available resources is not sufficient to fulfil the requests of the other processes. Thus, a deadlock exists, consisting of processes *P*1, *P*2, *P*3, and *P*4.

6.7. Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system **recover** from the deadlock automatically. There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to preempt some resources from one or more of the deadlocked processes.

6.7.1. Process Termination

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes**. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated**. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term ***minimum cost*** is not a precise one. Many factors may affect which process is chosen, including:

1. What the priority of the process is

2. How long the process has computed and how much longer the process will compute before completing its designated task

3. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)

4. How many more resources the process needs in order to complete

5. How many processes will need to be terminated?

6. Whether the process is interactive or batch

Chapter Seven

7.1. Memory Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be at least partially in main memory during execution.

To improve both the utilization of the CPU and the speed of its response to users, a general-purpose computer must keep several processes in memory. Many memory-management schemes exist, reflecting various approaches, and the effectiveness of each algorithm depends on the situation. Selection of a memory-management scheme for a system depends on many factors, especially on the hardware design of the system. Most algorithms require hardware support.

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses. A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

We begin our discussion by covering several issues that are pertinent to managing memory: basic hardware, the binding of symbolic memory addresses to actual physical addresses, and the distinction between logical and physical addresses. We conclude the section with a discussion of dynamic linking and shared libraries.

7.1.1. Basic Hardware

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. There are machine instructions that take memory addresses as arguments, but none that take disk addresses. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick. The same cannot be said of main memory, which is accessed via a transaction on the memory bus. Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. This situation is intolerable because of the frequency of memory accesses. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access. To manage a cache built into the CPU, the hardware automatically speeds up memory access without any operating-system control.

Not only are we concerned with the relative speed of accessing physical memory, but we also must ensure correct operation. For proper system operation we must protect the operating system from access by user processes. On multiuser systems, we must additionally protect user processes from one another. This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). Hardware implements this production in several different ways, as we show throughout the chapter. Here, we outline one possible implementation.

We first need to make sure that each process has a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as illustrated in Figure 7.1. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

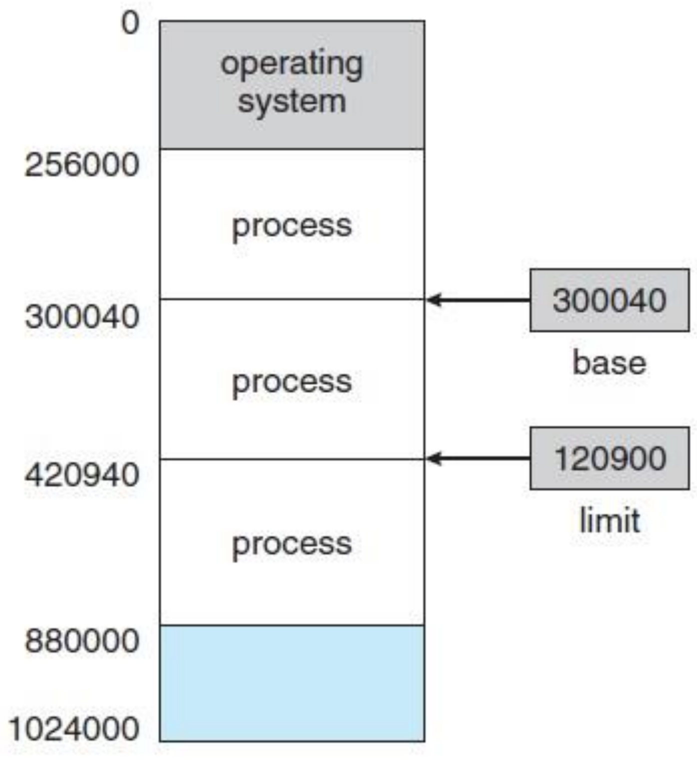


Figure 7-1 A base and a limit register define a logical address space

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (Figure 7.2). This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents. The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory. This provision allows the operating system to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services. Consider, for example, that an operating system for a multiprocessing system must execute context switches, storing the state of one process from the registers into main memory before loading the next process's context from main memory into the registers.

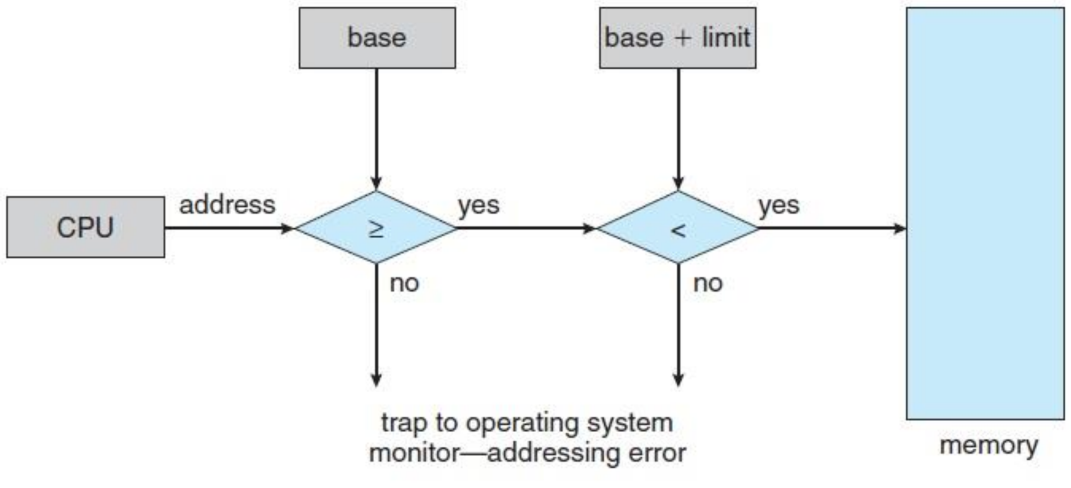


Figure 7-2 Hardware address protection with base and limit registers

7.1.2. Address Binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. The normal single-tasking procedure is to select one of the processes in the input queue and to load that process into memory. As the process is executed, it accesses instructions and data from memory. Eventually, the process terminates, and its memory space is declared available.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer may start at 00000, the first address of the user process need not be 00000. You will see later how a user program actually places a process in physical memory.

In most cases, a user program goes through several steps some of which may be optional before being executed (Figure 7.3). Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic (such as the variable count). A compiler typically **binds** these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader in turn binds the relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

• **Compile time**. If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.

• **Load time**. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**. In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

• **Execution time**. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

A major portion of this chapter is devoted to showing how these various bindings can be implemented effectively in a computer system and to discussing appropriate hardware support.

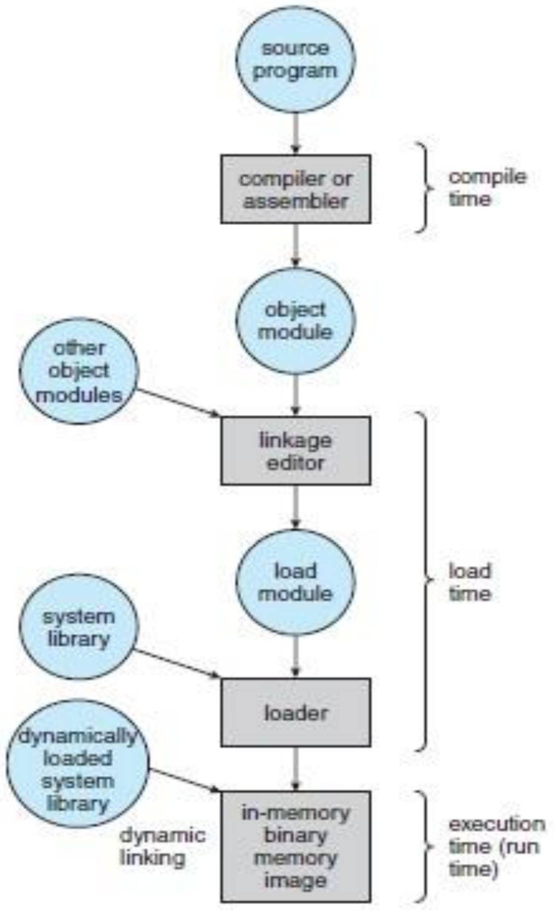


Figure 7-3 Multistep processing of a user program

7.1.3. Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit that is, the one loaded into the **memory address register** of the memory is commonly referred to as a **physical address**. The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a **virtual address**. We use ***logical address*** and ***virtual address*** interchangeably in this text. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**. The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 7.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location14000; an access to location346 is mapped to location 14346.

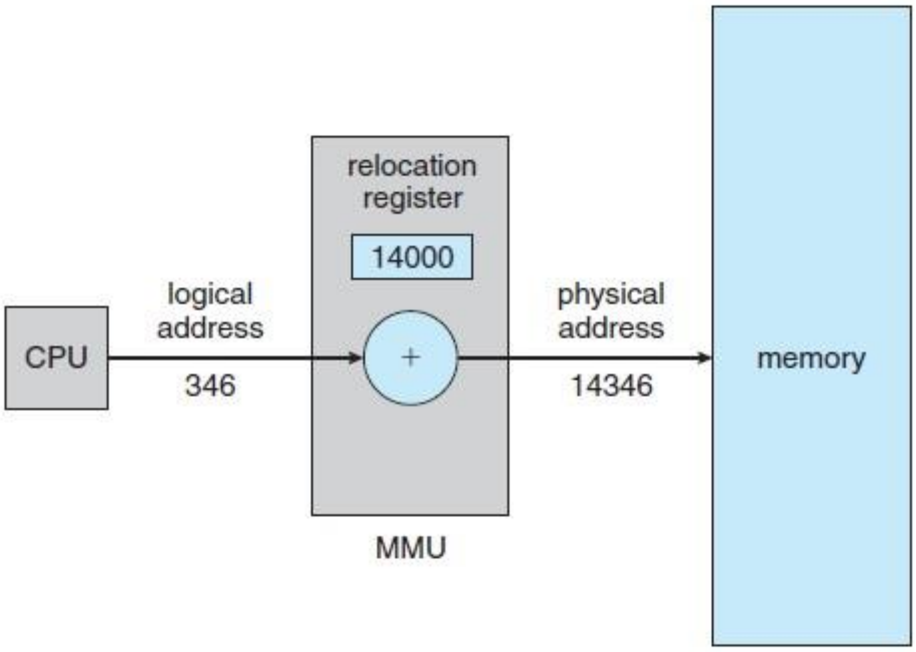


Figure 7-4 Dynamic relocation using a relocation register

The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses all as the number 346. Only when it is used as a memory address (in an indirect load or store, perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. This form of execution-time binding was discussed in Section 8.1.2. The final location of a referenced memory address is not determined until the reference is made.

We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range *R* + 0 to *R* + *max* for a base value *R*). The user program generates only logical addresses and thinks that the process runs in locations 0 to *max*. However, these logical addresses must be mapped to physical addresses before they are used. The concept of a logical address space that is bound to a separate physical address space is central to proper memory management.

7.2. Swapping

A process must be in memory to be executed. A process, however, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continued execution (Figure 8.5). Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

7.2.1. Standard Swapping

Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

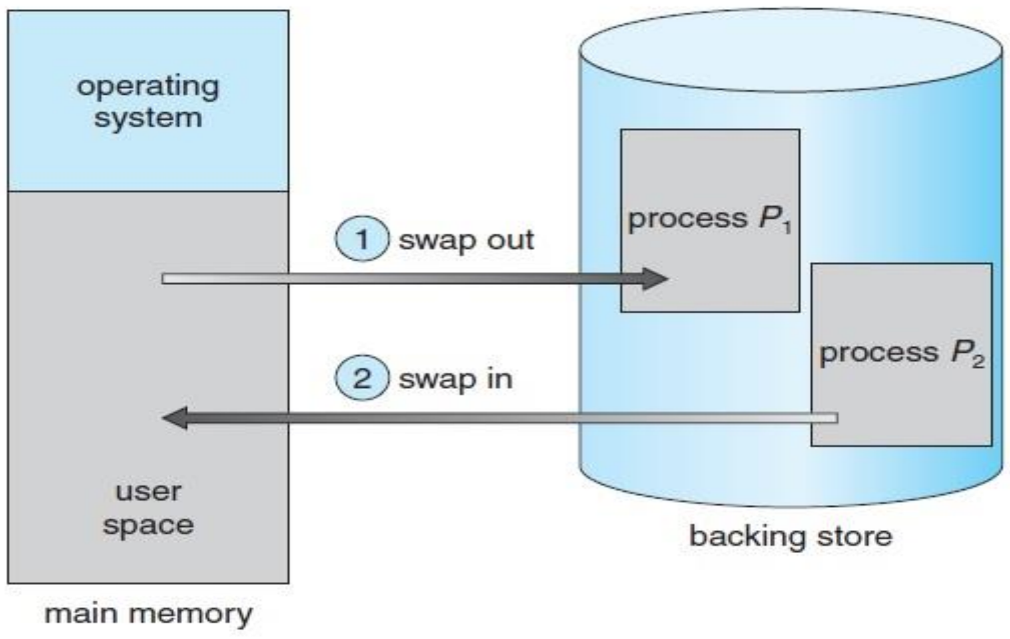


Figure 7-5 Swapping of two processes using a disk as a backing store

7.3. Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well. Thus, in this text, we discuss only the situation in which the operating system resides in low memory. The development of the other situation is similar.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

7.3.1. Memory Protection

Before discussing memory allocation further, we must discuss the issue of memory protection. We can prevent a process from accessing memory it does not own by combining two ideas previously discussed. If we have a system with a relocation register, together with a limit register, we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 7.6).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

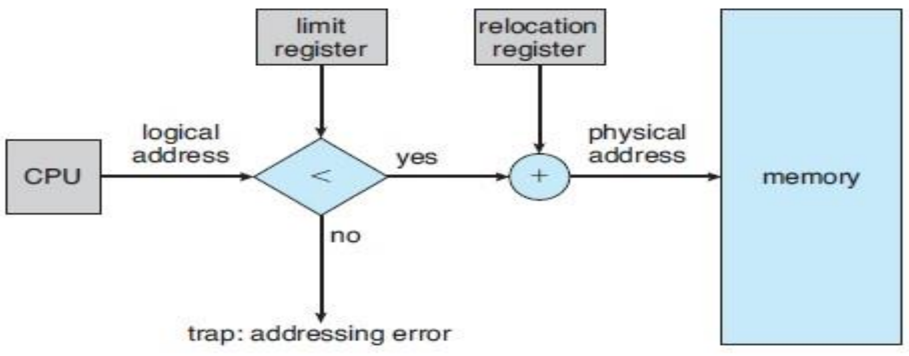


Figure 7-6 Hardware support for relocation and limit registers

7.3.2. Memory Allocation

Now we are ready to turn to memory allocation. One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this **multiple partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. This method was originally used by the IBM OS/360 operating system (called MFT)but is no longer in use. The method described next is a generalization of the fixed-partition scheme (called MVT); it is used primarily in batch environments. Many of the ideas presented here are also applicable to a time-sharing environment in which pure segmentation is used for memory management (Section 8.4).

In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**. Eventually, as you will see, memory contains a set of holes of various sizes.

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

At any given time, then, we have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, as mentioned, the memory blocks available comprise a ***set*** of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage allocation problem**, which concerns how to satisfy a request of size *n* from a list of free holes. There are many solutions to this problem. The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

• **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

• **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

• **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

7.3.3. Fragmentation

Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**. As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Whether we are using the first-fit or best-fit strategy, that’s can be affected the amount of fragmentation. (First fit is better for some systems, whereas best fit is better for others.) Another factor is which end of a free block is allocated. (Which is the leftover piece the one on the top or the one on the bottom?) No matter which algorithm is used, however, external fragmentation will be a problem.

Depending on the total amount of memory storage and the average process size, external fragmentation may be a minor or a major problem. Statistical analysis of first fit, for instance, reveals that, even with some optimization, given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18,464 bytes. Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The overhead to keep track of this hole will be substantially larger than the hole itself. The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is **internal fragmentation** unused memory that is internal to a partition.

One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever such memory is available. Two complementary techniques achieve this solution: segmentation and paging. These techniques can also be combined.

Fragmentation is a general problem in computing that can occur wherever we must manage blocks of data. We discuss the topic further in the storage management chapters.

7.4. Segmentation

As we've already seen, the user's view of memory is not the same as the actual physical memory. This is equally true of the programmer's view of memory. Indeed, dealing with memory in terms of its physical properties is inconvenient to both the operating system and the programmer. What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory? The system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

7.4.1. Basic Method

Do programmers think of memory as a linear array of bytes, some containing instructions and others containing data? Most programmers would say "no." Rather, they prefer to view memory as a collection of variable-sized segments, with no necessary ordering among the segments (Figure 7.7).

When writing a program, a programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy. She is not concerned with whether the stack is stored before or after the Sqrt() function. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack, the fifth instruction of the Sqrt(), and so on.

Segmentation is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.

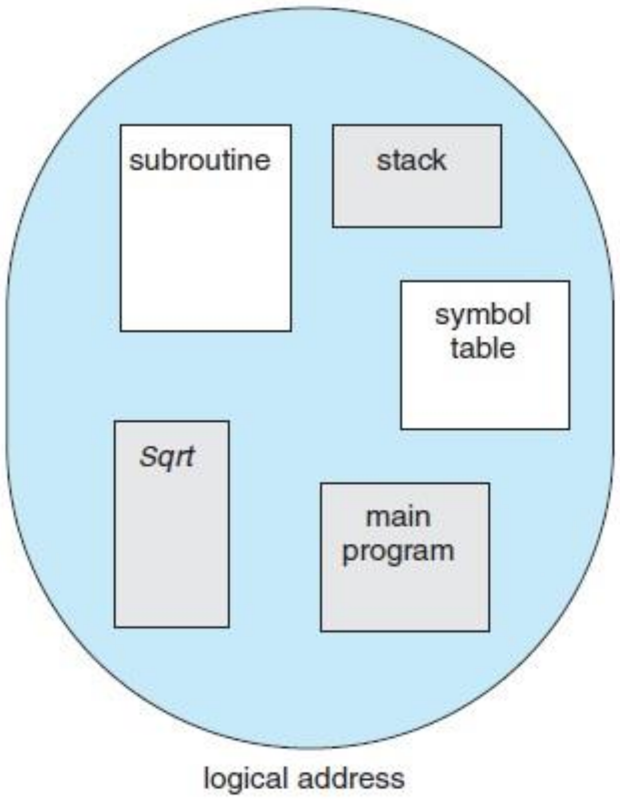


Figure 7-7 Programmer's view of a program

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program. A C compiler might create separate segments for the following:

1. The code

2. Global variables

3. The heap, from which memory is allocated

4. The stacks used by each thread

5. The standard C library

Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

7.4.2. Segmentation Hardware

Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a **segment table**. Each entry in the segment table has a **segment base** and a **segment limit**. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment.

The use of a segment table is illustrated in Figure 7.8. A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*. The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base limit register pairs.

As an example, consider the situation shown in Figure 7.9. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

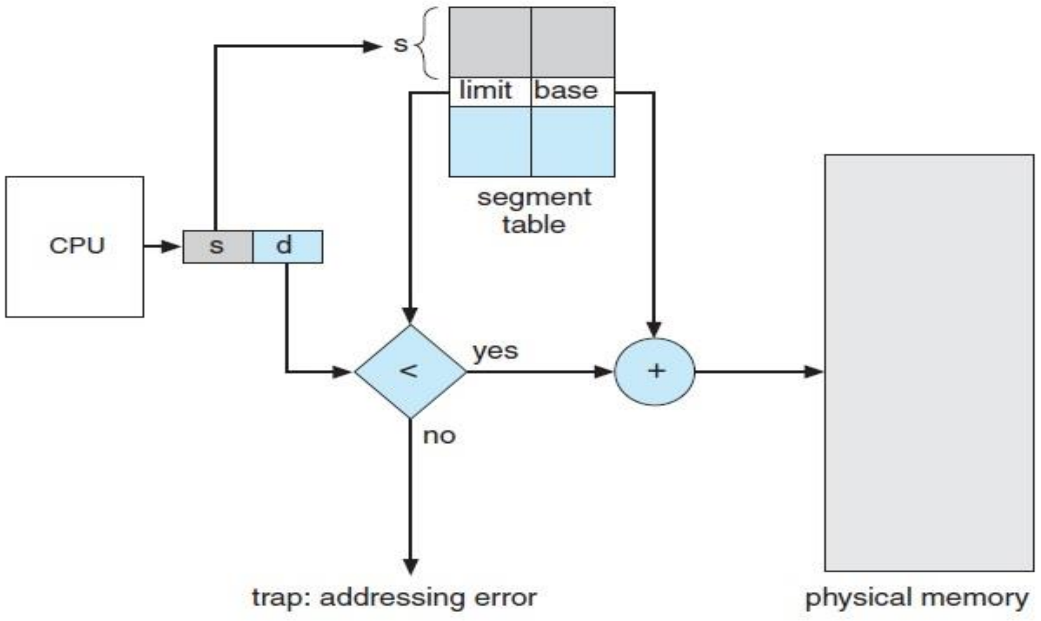
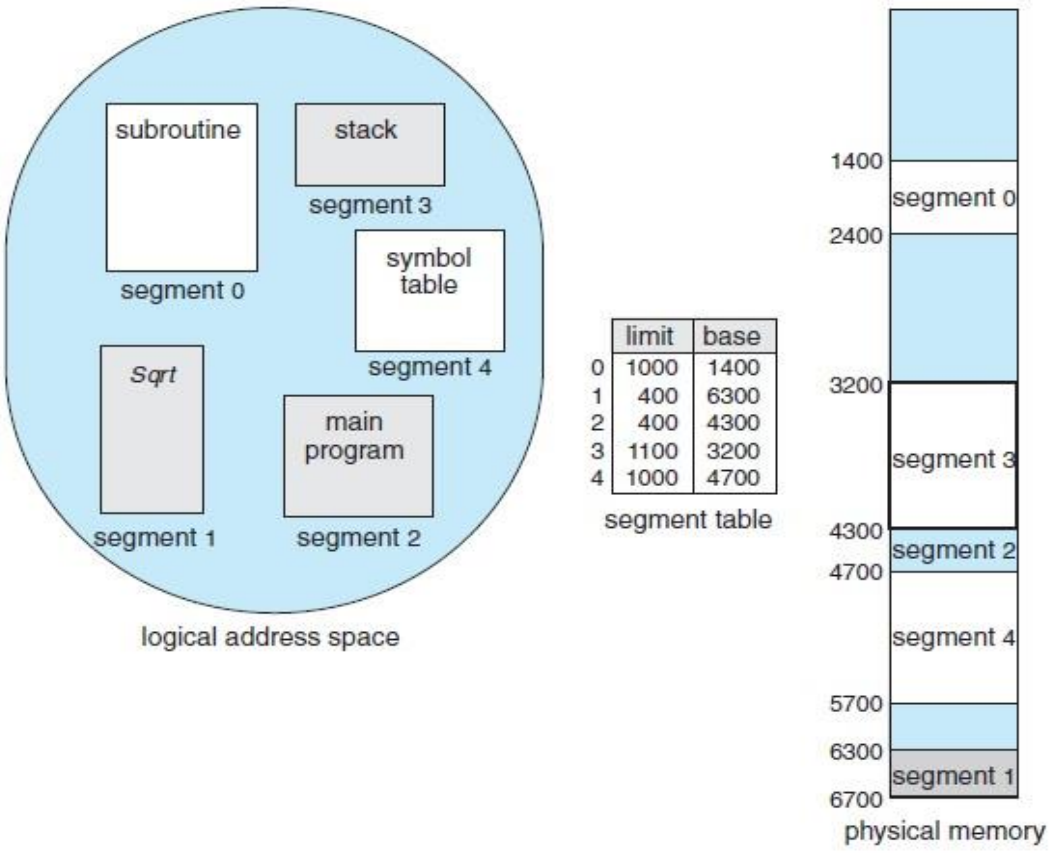


Figure 7-8 Paging hardware



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure 7-9 Example of segmentation

7.5. Paging

Segmentation permits the physical address space of a process to be noncontiguous. **Paging** is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Most memory-management schemes used before the introduction of paging suffered from this problem. The problem arises because, when code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The backing store has the same fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible. Because of its advantages over earlier methods, paging in its various forms is used in most operating systems, from those for mainframes through those for smartphones. Paging is implemented through cooperation between the operating system and the computer hardware.

7.5.1. Basic Method

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames. This rather simple idea has great functionality and wide ramifications. For example, the logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 264 bytes of physical memory.

The hardware support for paging is illustrated in Figure 7.10. Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**. The page number issued as an index into a **page table**. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 7.11.

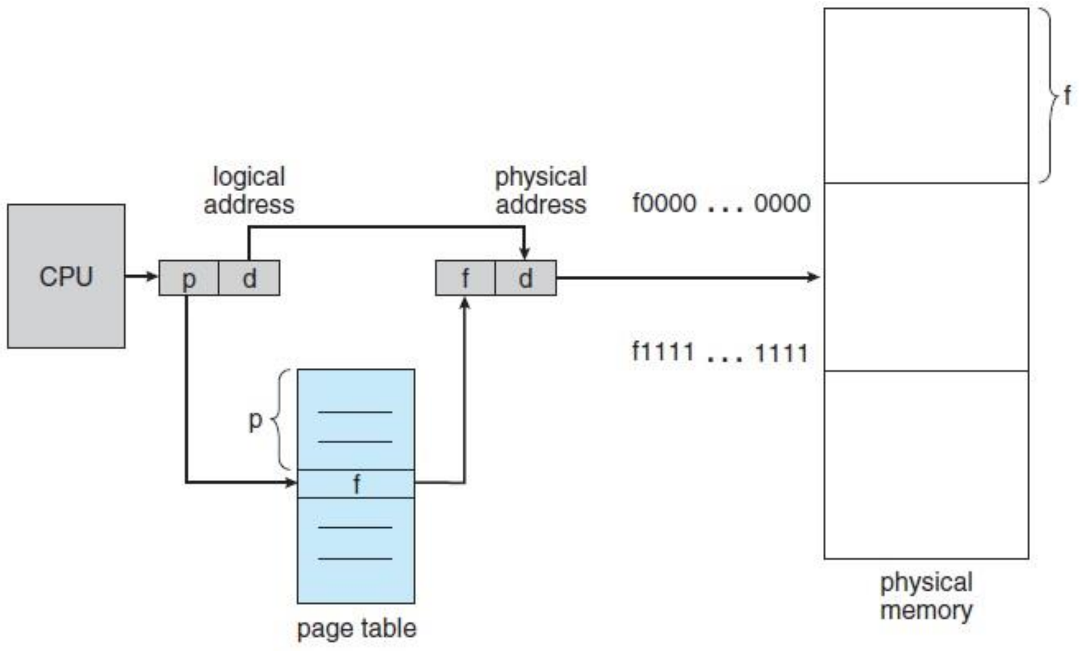


Figure 7-10 Paging hardware

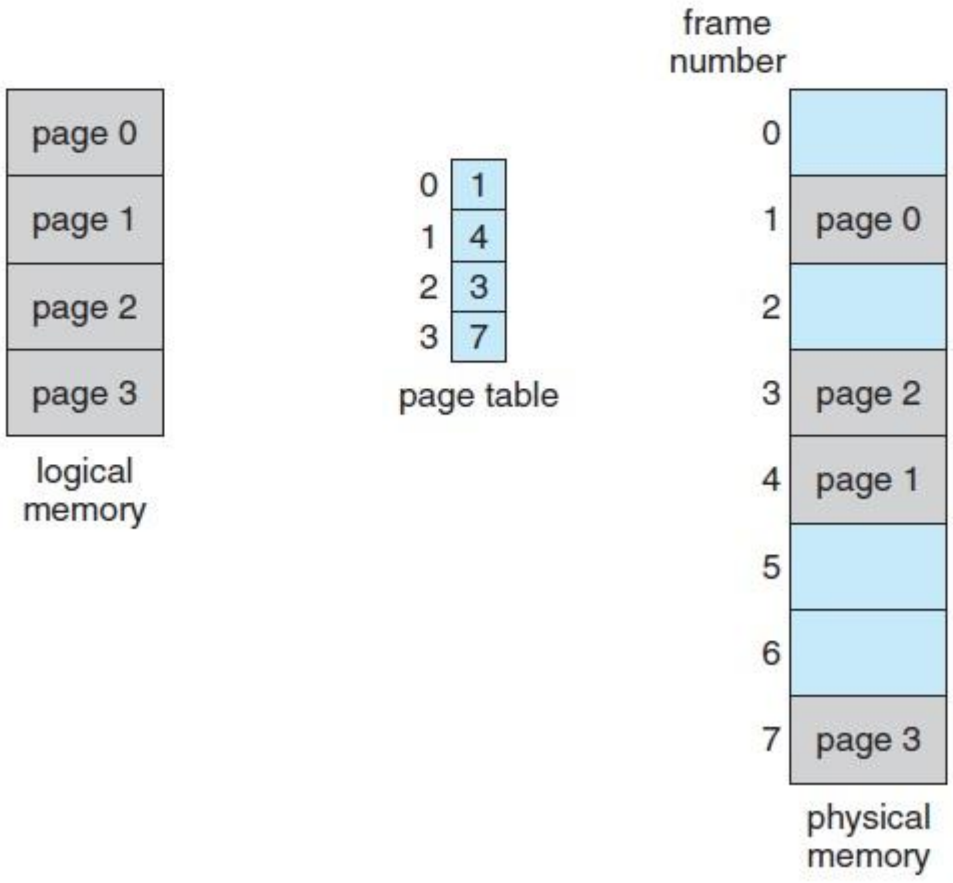
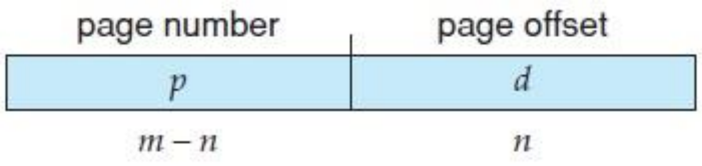


Figure 7-11 Paging model of logical and physical memory

The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2*m*, and a page size is 2*n* bytes, then the high-order *m*− *n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. Thus, the logical address is as follows:



where *p* is an index into the page table and *d* is the displacement within the page. As a concrete (although minuscule) example, consider the memory in Figure 7.12. Here, in the logical address, *n*= 2 and *m* = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages), we show how the programmer's view of memory can be mapped into physical memory. Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0]. Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3]. Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0]. Logical address 13 maps to physical address 9.

You may have noticed that paging itself is a form of dynamic relocation. Every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory. When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. Notice that frames are allocated as units. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full. For example, if page size is 2,048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes. In the worst case, a process would need n pages plus 1 byte. It would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame.

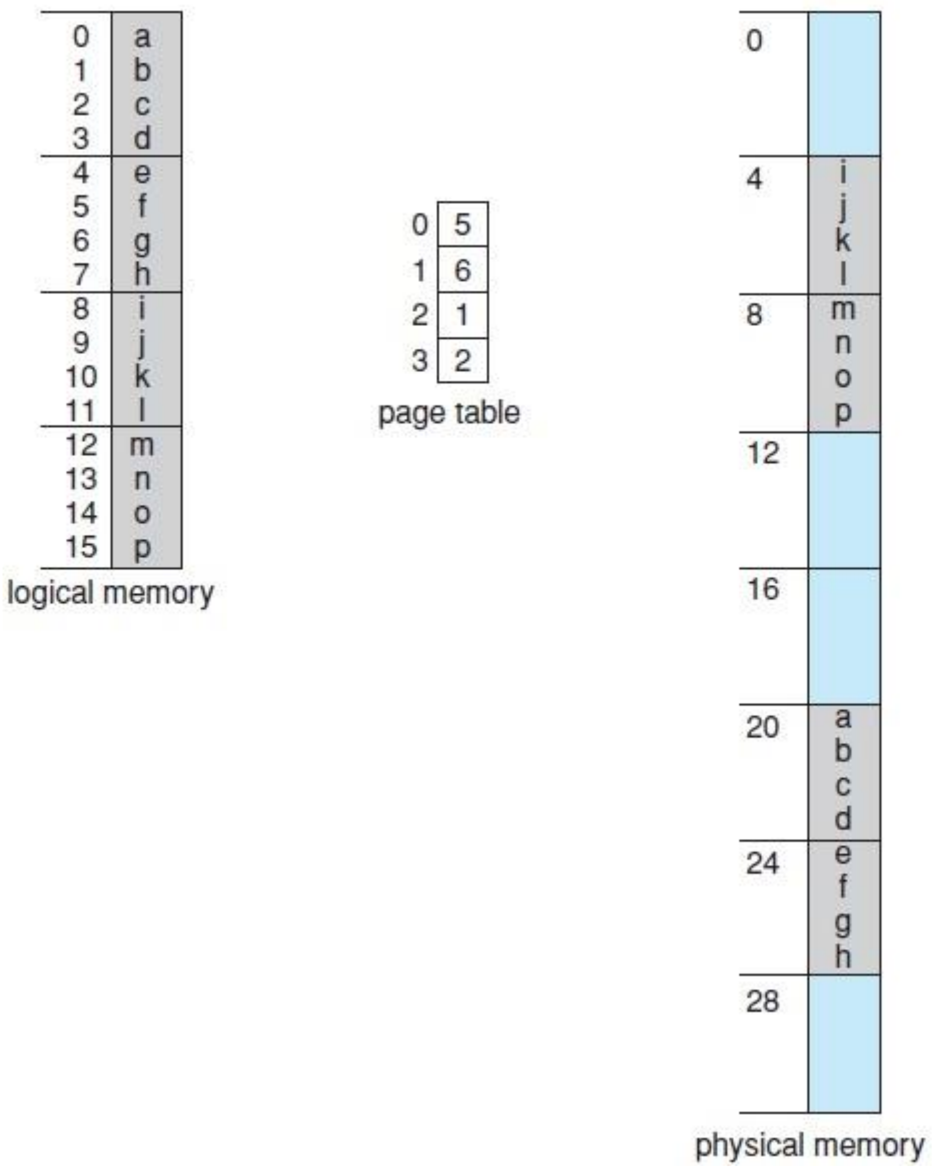


Figure 7-12 Paging example for a 32-byte memory with 4-byte pages

If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount data being transferred is larger. Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are between 4 KB and 8 KB in size and some systems support even larger page sizes. Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages. Researchers are now developing support for variable on-the-fly page size.

Frequently, on a 32-bit CPU, each page-table entry is 4 bytes long, but that size can vary as well.A32-bit entry can point to one of 232 physical page frames. If frame size is 4 KB (212), then a system with 4-byte entries can address 244 bytes (or 16 TB) of physical memory. We should note here that the size of physical memory in a paged memory system is different from the maximum logical size of a process. As we further explore paging, we introduce other information that must be kept in the page-table entries. That information reduces the number of bits available to address page frames. Thus, a system with 32-bit page-table entries may address less physical memory than the possible maximum.A32-bit CPU uses 32-bit addresses, meaning that a given process space can only be 232 bytes (4 TB). Therefore, paging lets us use physical memory that is larger than what can be addressed by the CPU's address pointer length.

When a process arrives in the system to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on (Figure 7.13).

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one program. In fact, the user program is scattered throughout physical memory, which also holds other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system. Notice that the user process by definition is unable to access memory it does not own. It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses. If a user makes a system call (to do I/O, for example), and provides an address as a parameter (a buffer, for instance), then that address must be mapped to produce the correct physical address. The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

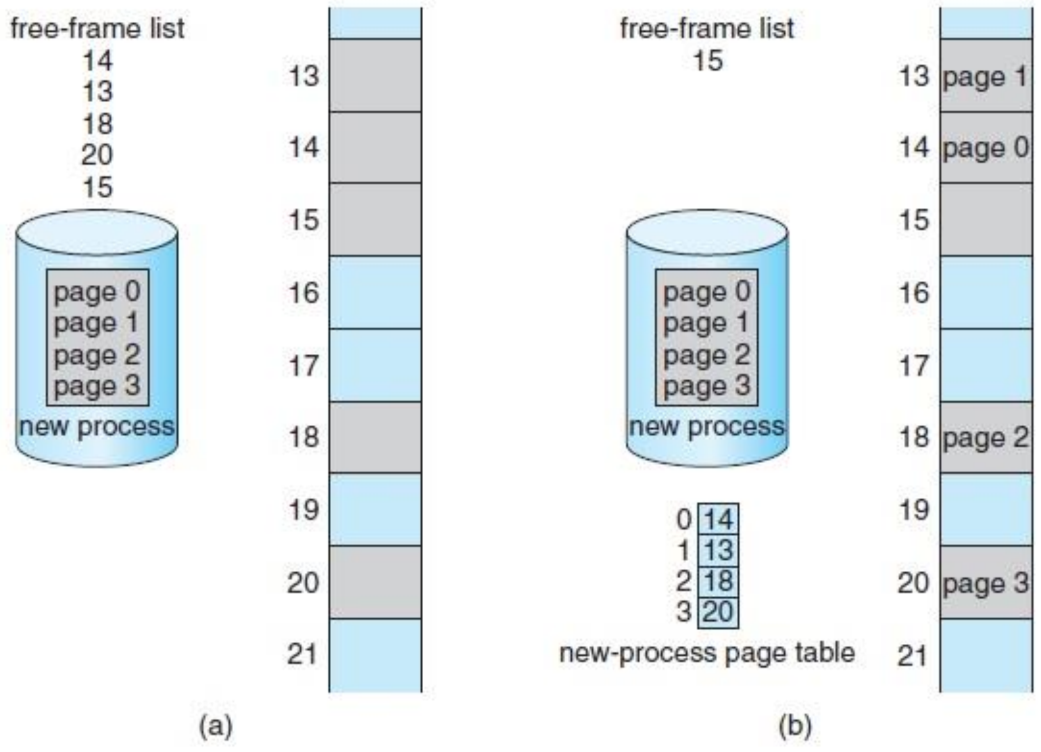


Figure 7-13 Free frames (a) before allocation and (b) after allocation

7.5.2. Hardware Support

Each operating system has its own methods for storing page tables. Some allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block. When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table. Other operating systems provide one or at most a few page tables, which decreases the overhead involved when processes are context-switched.

The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated **registers**. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers. Instructions to load or modify the page-table registers are, of course, privileged, so that only the operating system can change the memory map. The DEC PDP-11 is an example of such an architecture. The address consists of 16 bits, and the page size is 8 KB. The page table thus consists of eight entries that are kept in fast registers.

The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries). Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible. Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

The problem with this approach is the time required to access a user memory location. If we want to access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for *i*. This task requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, ***two*** memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2. This delay would be intolerable under most circumstances. We might as well resort to swapping!

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer (TLB)**. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, however, the TLB must be kept small. It is typically between 32 and 1,024 entries in size. Some CPUs implement separate instruction and data address TLBs. That cans double the number of TLB entries available, because those lookups occur in different pipeline steps. We can see in this development an example of the evolution of CPU technology: systems have evolved from having no TLBs to having multiple levels of TLBs, just as they have multiple levels of caches.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found, its frame number is immediately available and is used to access memory. As just mentioned, these steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.

If the page number is not in the TLB (known as a **TLB miss**), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory (Figure 7.14). In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) through round-robin to random. Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves. Furthermore, some TLBs allow certain entries to be **wired down**, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, the attempt is treated as a TLB miss. In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously. If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be **flushed** (or erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**. An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 100 nanoseconds to access memory, then a mapped-memory access takes 100 nanoseconds when the page number is in the TLB. If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 200 nanoseconds. (We are assuming that a page-table lookup takes only one memory access, but it can take more, as we shall see.) To find the **effective memory-access time**, we weight the case by its probability:

effective access time = $0.80 \times 100 + 0.20 \times 200 = 120$ nanoseconds

In this example, we suffer a 20-percent slowdown in average memory-access time (from 100 to 120 nanoseconds). For a 99-percent hit ratio, which is much more realistic, we have effective access time = $0.99 \times 100 + 0.01 \times 200 = 101$ nanoseconds This increased hit rate produces only a 1 percent slowdown in access time. As we noted earlier, CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above. For instance, the Intel Core i7 CPU has a 128-entry L1 instruction TLB and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU six cycles to check for the entry in the L2 512-entry TLB. Amiss in L2means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take hundreds of cycles, or interrupt to the operating system to have it do the work. A complete performance analysis of paging overhead in such a system would require miss-rate information about each TLB tier. We can see from the general information above, however, that hardware features can have a significant effect on memory performance and that operating-system improvements (such as paging) can result in and, in turn, be affected by hardware changes (such as TLBs).We will further explore the impact of the hit ratio on the TLB in Chapter 9.TLBs are a hardware feature and therefore would seem to be of little concern to operating systems and their designers. But the designer needs to understand the function and features of TLBs, which vary by hardware platform. For optimal operation, an operating-system design for a given platform must implement paging according to the platform's TLB design. Likewise, a change in the TLB design (for example, between generations of Intel CPUs) may necessitate a change in the paging implementation of the operating systems that use it.

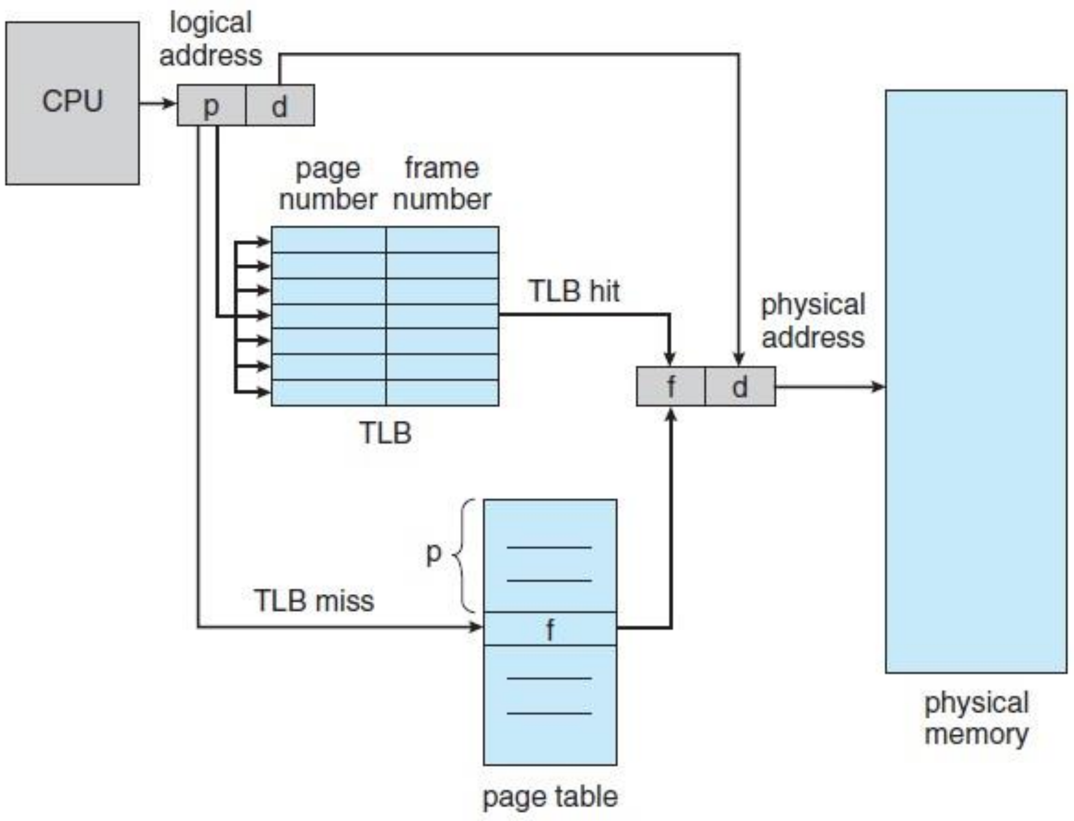


Figure 7-14 Paging hardware with TLB

7.5.3. Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

We can easily expand this approach to provide a finer level of protection. We can create hardware to provide read-only, read–write, or execute-only protection; or, by providing separate protection bits for each kind of access, we can allow any combination of these accesses. Illegal attempts will be trapped to the operating system.

One additional bit is generally attached to each entry in the page table: a **valid–invalid** bit. When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

Suppose, for example, that in a system with a 14-bit address space (0 to 16383), we have a program that should use only addresses 0 to 10468. Given a page size of 2 KB, we have the situation shown in Figure 7.15. Addresses in pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table. Any attempt to generate an address in pages 6 or 7, however, will find that the valid–invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).

Notice that this scheme has created a problem. Because the program extends only to address 10468, any reference beyond that address is illegal. However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid. Only the addresses from 12288 to 16383 are invalid. This problem is a result of the 2-KB page size and reflects the internal fragmentation of paging.

Rarely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a **page-table length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

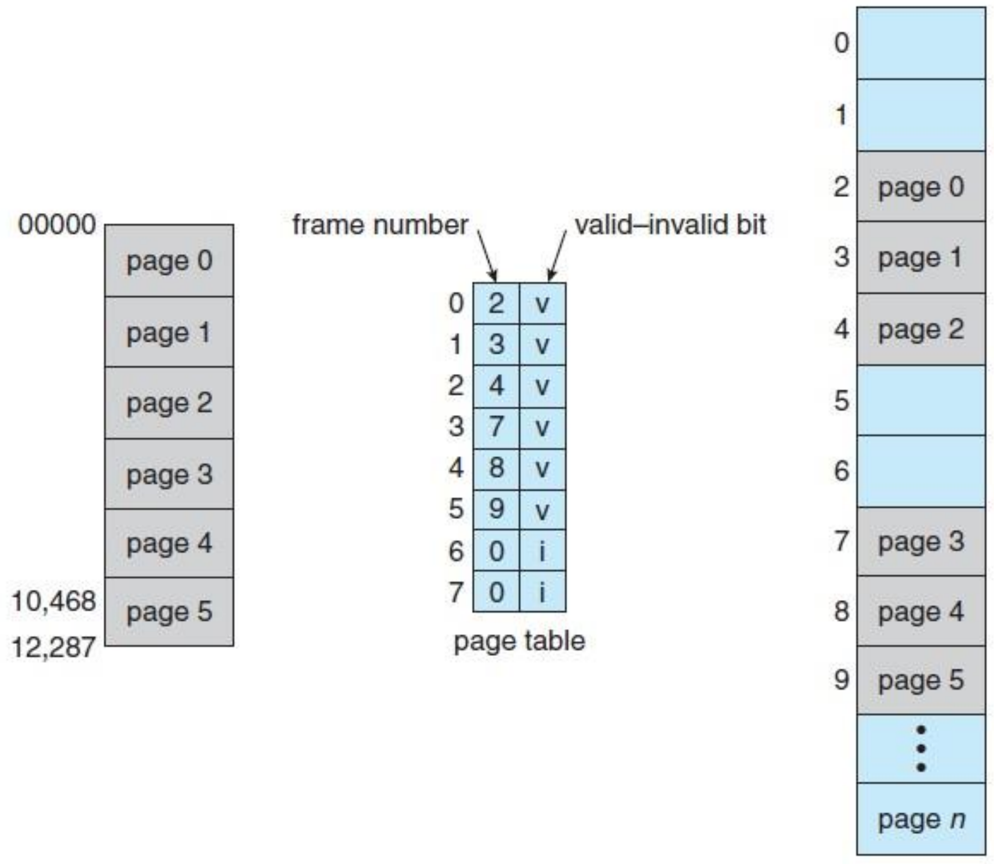


Figure 7-15 Valid (v) or invalid (i) bit in a page table

Chapter Eight

In the previous chapter we discussed various memory-management strategies used in computer systems. All these strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. However, they tend to require that an entire process be in memory before it can execute.

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory. This technique frees programmers from the concerns of memory-storage limitations. Virtual memory also allows processes to share files easily and to implement shared memory. In addition, it provides an efficient mechanism for process creation. Virtual memory is not easy to implement, however, and may substantially decrease performance if it is used carelessly. In this chapter, we discuss virtual memory in the form of demand paging and examine its complexity and cost.

8.1. Background

The memory-management algorithms outlined in Chapter 7 are necessary because of one basic requirement: The instructions being executed must be in physical memory. The first approach to meeting this requirement is to place the entire logical address space in physical memory. Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer. The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable; but it is also unfortunate, since it limits the size of a program to the size of physical memory. In fact, an examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

• Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.

• Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than

10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.

• Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Even in those cases where the entire program is needed, it may not all be needed at the same time. The ability to execute a program that is only partially in memory would confer many benefits:

• A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large ***virtual*** address space, simplifying the programming task.

• Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.

• Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and the user. **Virtual memory** involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 8.1). Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed. The **virtual address space** of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address say, address 0 and exists in contiguous memory, as shown in Figure 8.2. Recall from Chapter 7, though, that in fact physical memory may be organized in page frames and that the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.

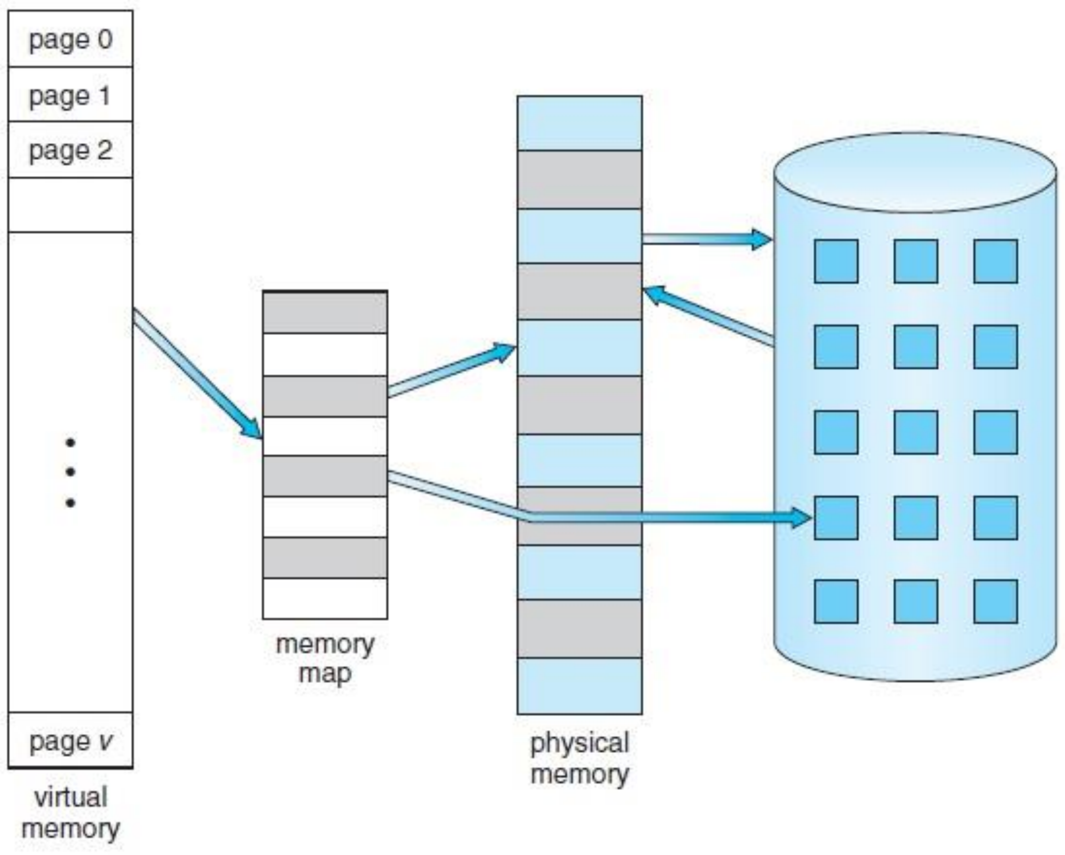


Figure 8-1 Diagram showing virtual memory that is larger than physical memory

8.2. Demand Paging

Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially ***need*** the entire program in memory. Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user. An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.

With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is similar to a paging system with swapping (Figure 8.2) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper**. A lazy swapper never swaps a page into memory unless that page will be needed. In the context of a demand-paging system, use of the term "swapper" is technically incorrect. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process. We thus use "pager," rather than "swapper," in connection with demand paging.

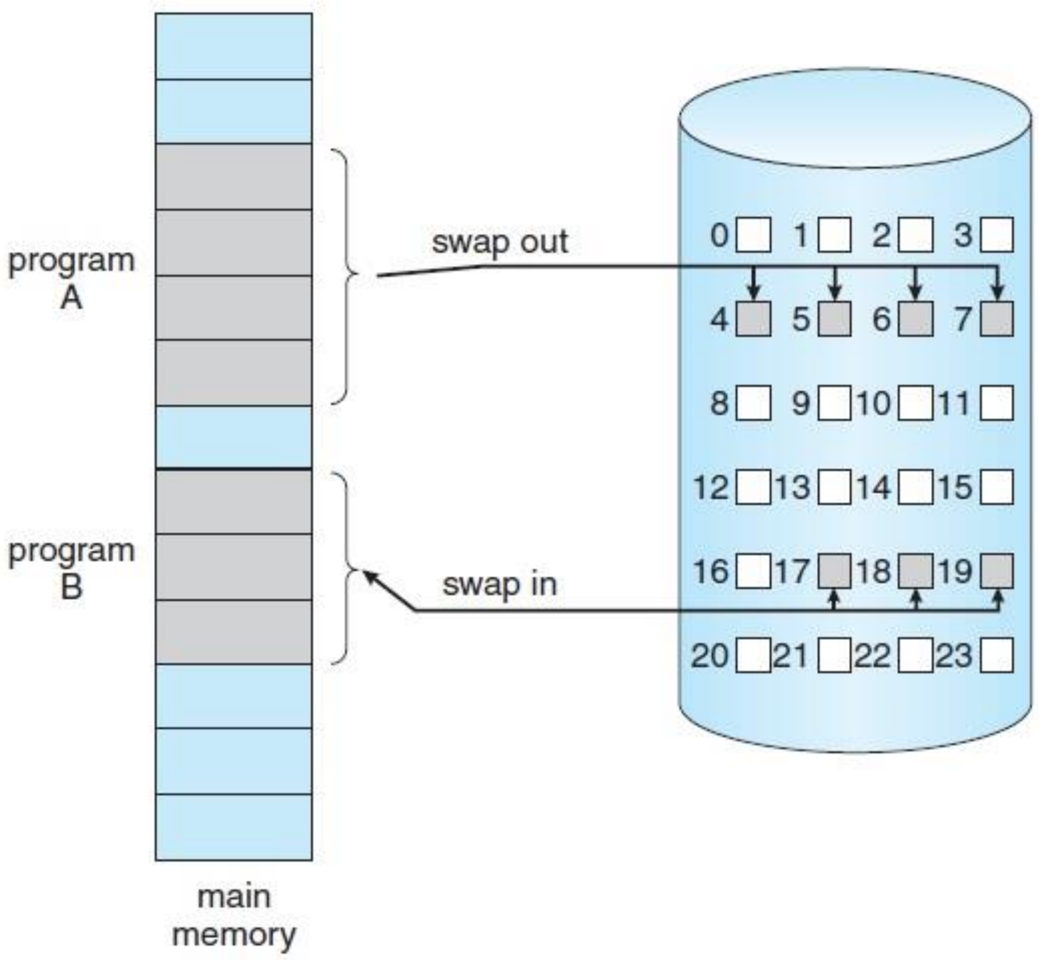


Figure 8-2 Transfer of a paged memory to contiguous disk space

8.2.1. Basic Concepts

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid invalid bit scheme that has been described before can be used for this purpose. This time, however, when this bit is set to “valid,” the associated page is both legal and in memory. If the bit is set to “invalid,” the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure 8.3.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all pages that are actually needed and only those pages, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are **memory resident**, execution proceeds normally.

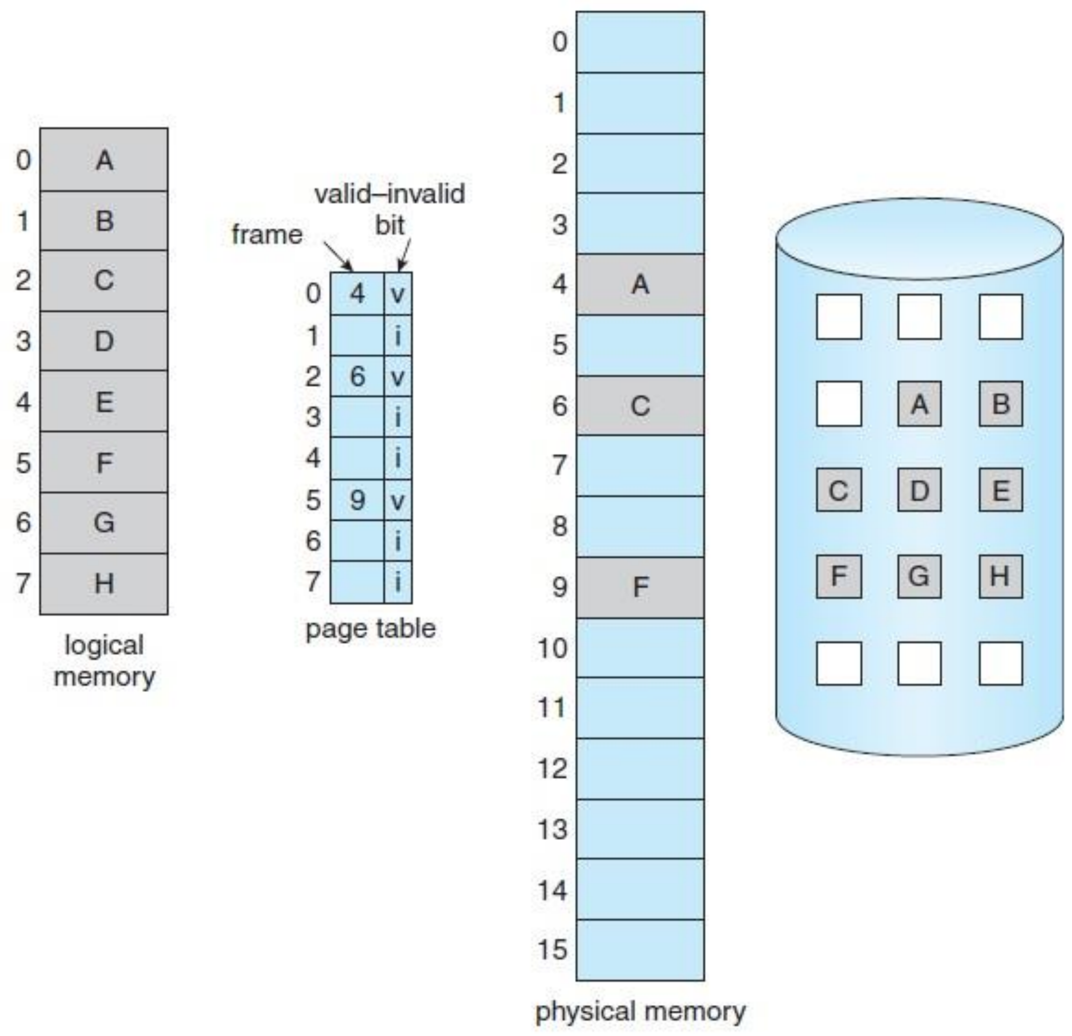


Figure 8-3 Page table when some pages are not in main memory

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**. The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system’s failure to bring the desired page into memory. The procedure for handling this page fault is straightforward (Figure 8.4):

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

2. If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one from the free-frame list, for example).

4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory. In the extreme case, we can start executing a process with ***no*** pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.

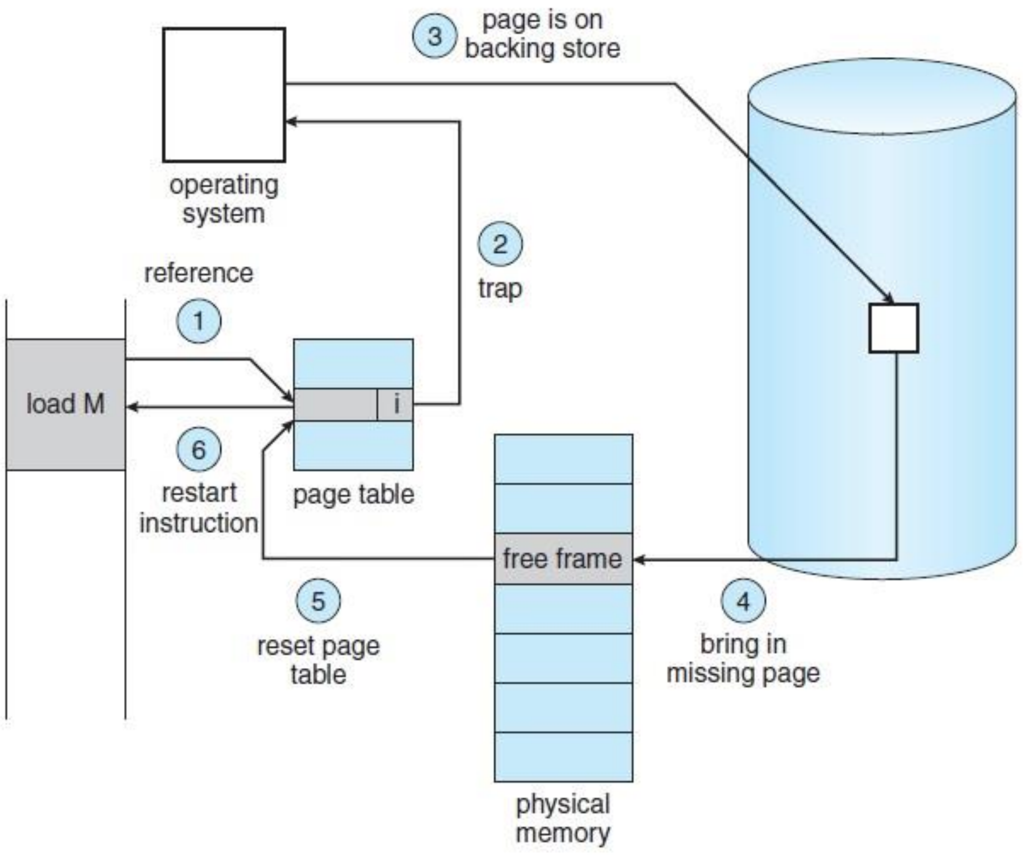


Figure 8-4 Steps in handling a page fault

Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Fortunately, analysis of running processes shows that this behaviour is exceedingly unlikely. Programs tend to have **locality of reference**, which results in reasonable performance from demand paging. The hardware to support demand paging is the same as the hardware for paging and swapping:

• **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.

• **Secondary memory**. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space**.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process in ***exactly*** the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

8.2.2. Page Replacement

In our earlier discussion of the page-fault rate, we assumed that each page faults at most once, when it is first referenced. This representation is not strictly accurate, however. If a process of ten pages actually uses only half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had forty frames, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).

If we increase our degree of multiprogramming, we are **over-allocating** memory. If we run six processes, each of which is ten pages in size but actually uses only five pages, we have higher CPU utilization and throughput, with ten frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for sixty frames when only forty are available.

Further, consider that system memory is not used only for holding program pages. Buffers for I/O also consume a considerable amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory.

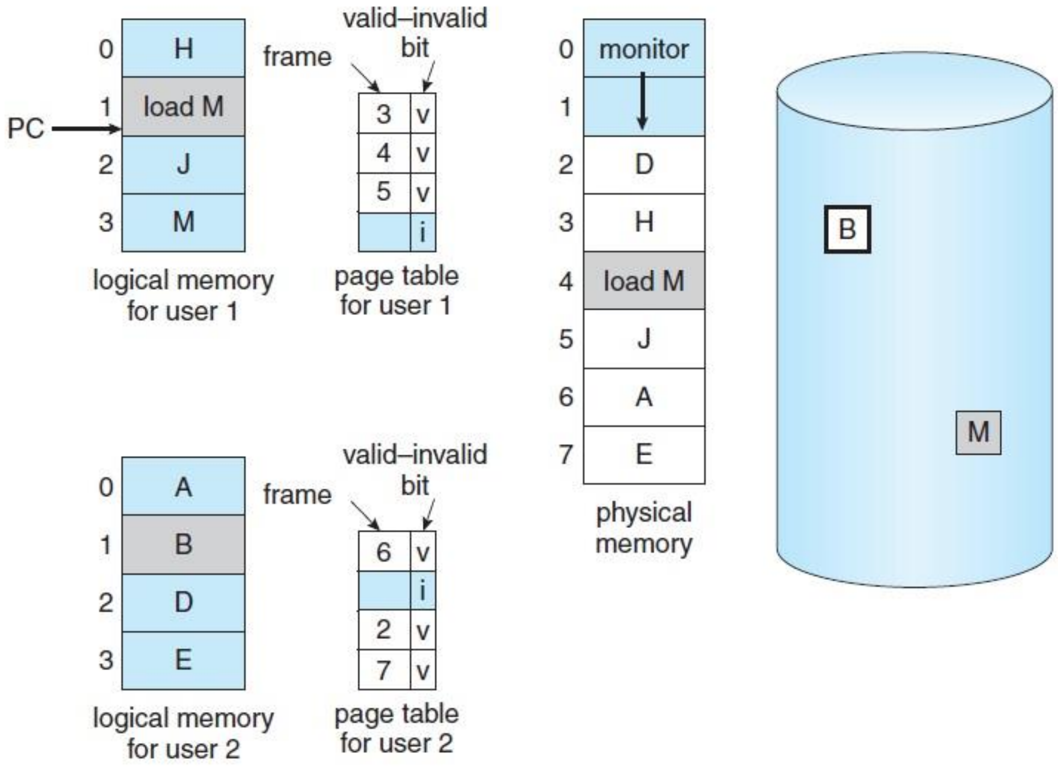


Figure 8-5 Need for page replacement

Over-allocation of memory manifests itself as follows. While a user process is executing, a page fault occurs. The operating system determines where the desired page is residing on the disk but then finds that there are ***no*** free frames on the free-frame list; all memory is in use (Figure 8.5).

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system paging should be logically transparent to the user. So this option is not the best choice. The operating system could instead swap out a process, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances, we discuss the most common solution: **page replacement**.

8.2.3. Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Figure 8.6). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.

2. Find a free frame:

a. If there is a free frame, use it.

b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.

c. Write the victim frame to the disk; change the page and frame tables

accordingly.

3. Read the desired page into the newly freed frame; change the page and frame tables.

4. Continue the user process from where the page fault occurred.

Notice that, if no frames are free, ***two*** page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. If the modify bit is not set, however, the page has ***not*** been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there. This technique also applies to read-only pages (for example, pages of binary code).

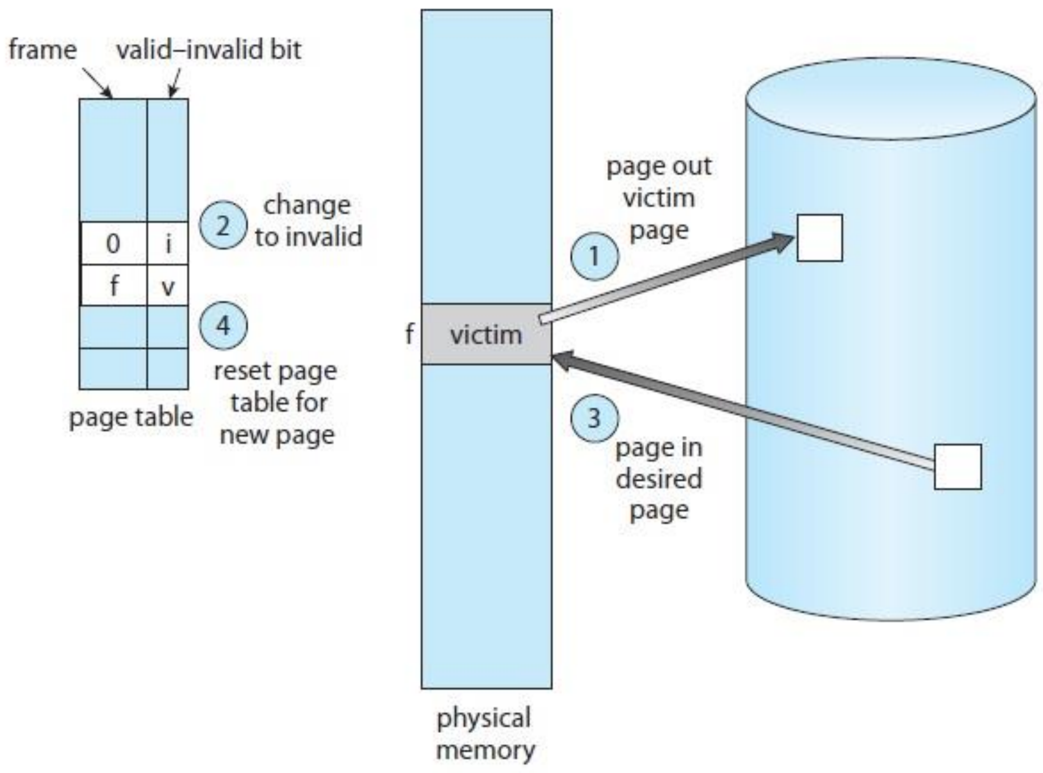


Figure 8-6 Page replacement

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With no demand paging, user addresses are mapped into physical addresses, and the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of twenty pages, we can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process.

We must solve two major problems to implement demand paging: we must develop a **frame-allocation algorithm** and a **page-replacement algorithm**. That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive. Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**.

8.2.4. FIFO Page Replacement

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue. For our example reference string, our three frames are initially empty. The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 9.12. Every time a fault occurs, we show which pages are in our three frames. There are fifteen faults altogether.

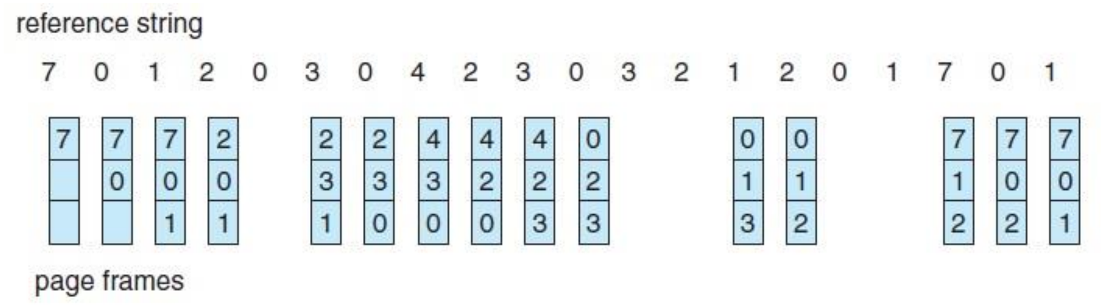


Figure 8-7 FIFO page-replacement algorithm

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. On the one hand, the page replaced may be an initialization module that was used a long time ago and is no longer needed. On the other hand, it could contain a heavily used variable that was initialized early and is in constant use.

Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

8.2.5. Optimal Page Replacement

This algorithm has the lowest page-fault rate of all algorithms. It is simply this: Replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 8.8. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because page 7 will not be used until reference 18,whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. (We encountered a similar situation with the SJF CPU-scheduling algorithm) As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst and within 4.7 percent on average.

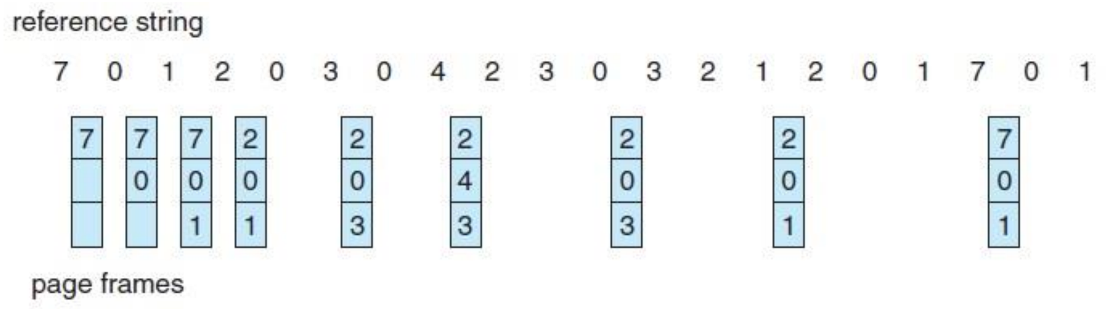


Figure 8-8 Optimal page-replacement algorithm

8.2.6. LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward versus forward in time) is that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be ***used.*** If we use the recent past as an approximation of the near future, then we can replace the page that ***has not been used*** for the longest period of time. This approach is the **least recently used (LRU) algorithm**.

LRU replacement associates with each page the time of that page’s last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let *SR* be the reverse of a reference string *S*, then the page-fault rate for the OPT algorithm on *S* is the same as the page-fault rate for the OPT algorithm on *SR*. Similarly, the page-fault rate for the LRU algorithm on *S* is the same as the page-fault rate for the LRU algorithm on *SR*.)

The result of applying LRU replacement to our example reference string is shown in Figure 8.9. The LRU algorithm produces twelve faults. Notice that the first five faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with twelve faults is much better than FIFO replacement with fifteen.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is ***how*** to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

• **Counters**. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

• **Stack**. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom (Figure 8.9). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

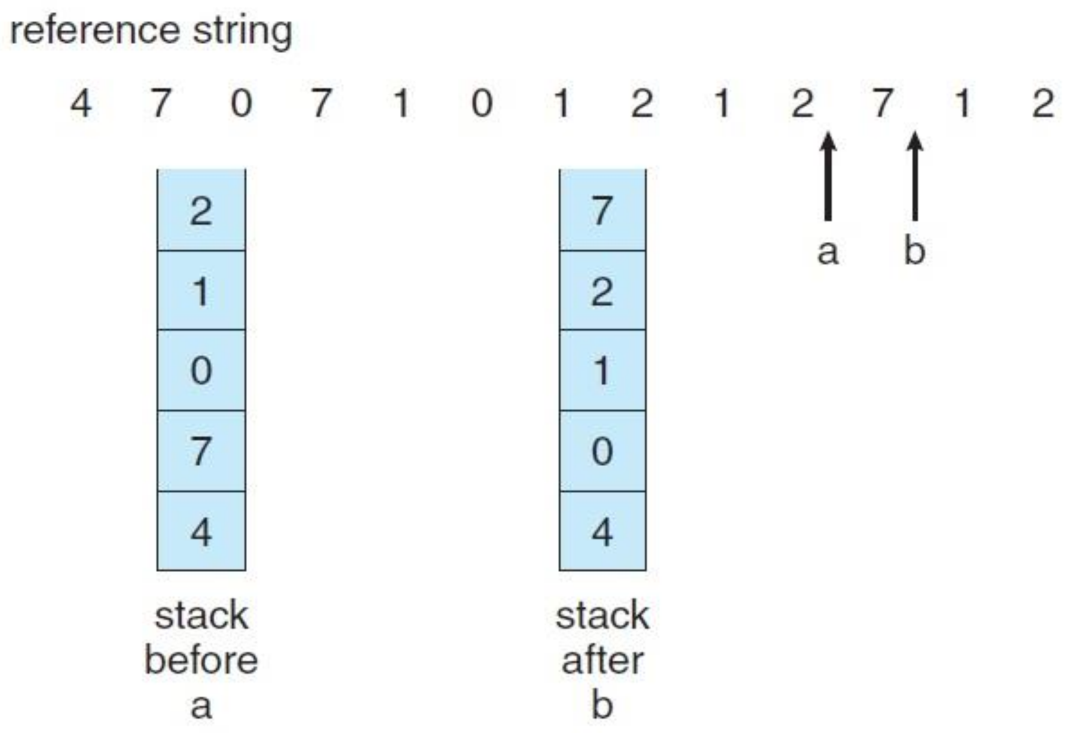


Figure 8-9 Use of a stack to record the most recent page references

8.3. Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - ✓ low CPU utilization
  - ✓ operating system thinks that it needs to increase the degree of multiprogramming
  - ✓ another process added to the system
- Thrashing = a process is busy swapping pages in and out

Chapter Nine

9.1. Storage Management

Since main memory is usually too small to accommodate all the data and programs permanently, the computer system must provide secondary storage to back up main memory. Modern computer systems use disks as the primary on-line storage medium for information (both programs and data). The file system provides the mechanism for on-line storage of and access to both data and programs residing on the disks. A file is a collection of related information defined by its creator. The files are mapped by the operating system onto physical devices. Files are normally organized into directories for ease of use.

The devices that attach to a computer vary in many aspects. Some devices transfer a character or a block of characters at a time. Some can be accessed only sequentially, others randomly. Some transfer data synchronously, others asynchronously. Some are dedicated, some shared. They can be read-only or read–write. They vary greatly in speed. In many ways, they are also the slowest major component of the computer.

Because of all this device variation, the operating system needs to provide a wide range of functionality to applications, to allow them to control all aspects of the devices. One key goal of an operating system’s I/O subsystem is to provide the simplest interface possible to the rest of the system. Because devices are a performance bottleneck, another key is to optimize I/O for maximum concurrency.

9.2. Overview of Mass-Storage Structure

In this section, we present a general overview of the physical structure of secondary and tertiary storage devices.

9.2.1. Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 10.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

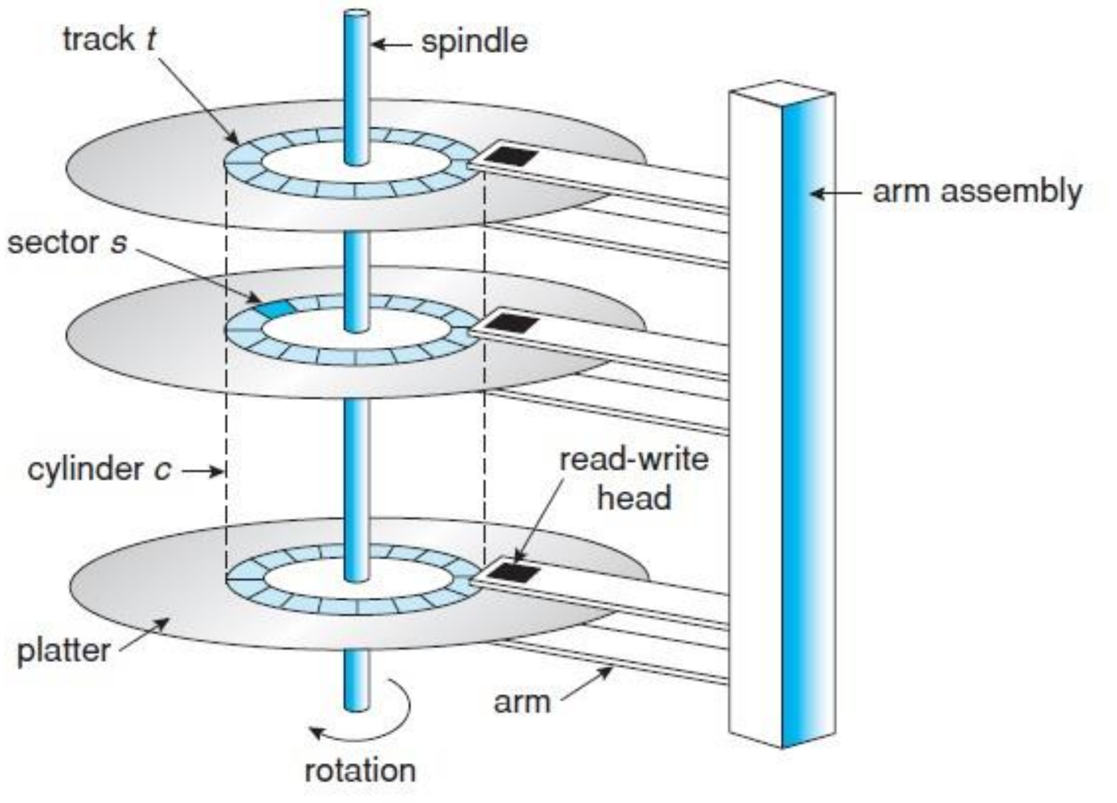


Figure 9-1 Moving-head disk mechanism

A read write head “flies” just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks that are at one arm position makes up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

9.3. Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. For magnetic disks, the access time has two major components, as mentioned in Section 10.1.1. The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which disk

I/O requests are serviced. Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. How does the operating system make this choice? Any one of several disk-scheduling algorithms can be used, and we discuss them next.

9.3.1. FCFS Scheduling

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

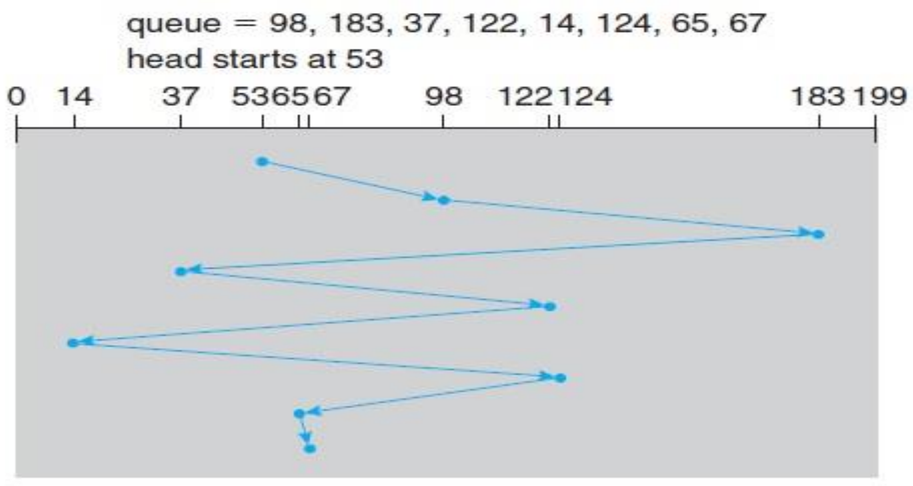


Figure 9-2 FCFS disk scheduling

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 10.4. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased substantially, and performance could be thereby improved.

9.3.2. SSTF Scheduling

It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm**. The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closest to the current head position.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183 (Figure 9.3). This scheduling method results in a total head movement of only 236 cylinders little more than one-third of the distance needed for FCFS scheduling of this request queue. Clearly, this algorithm gives a substantial improvement in performance.

SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling; and like SJF scheduling, it may cause starvation of some requests. Remember that requests may arrive at any time. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely. This scenario becomes increasingly likely as the pending-request queue grows longer.

Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal. In the example, we can do better by moving the head from

53 to 37, even though the latter is not closest, and then to 14, before turning around to service 65, 67, 98, 122, 124, and 183. This strategy reduces the total head movement to 208 cylinders.

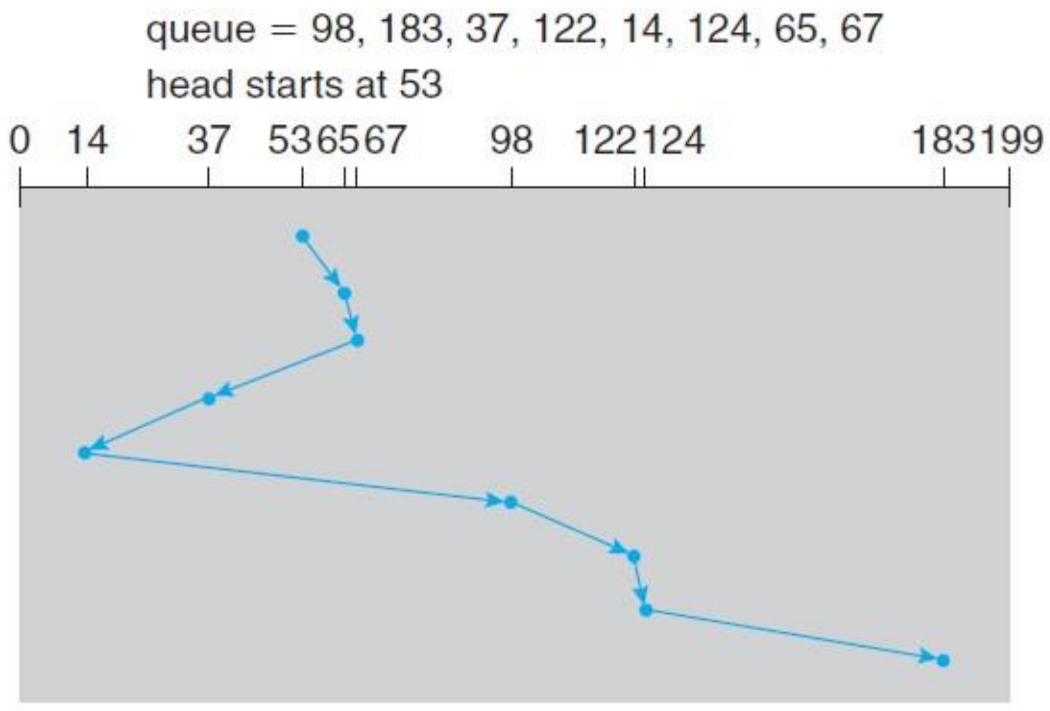


Figure 9-3 SSTF disk scheduling

9.3.3. SCAN Scheduling

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way.

Let’s return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head’s current position. Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 9.4). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Assuming a uniform distribution of requests for cylinders, consider the density of requests when the head reaches one end and reverses direction. At this point, relatively few requests are immediately in front of the head, since these cylinders have recently been serviced. The heaviest density of requests is at the other end of the disk. These requests have also waited the longest, so why not go there first? That is the idea of the next algorithm.

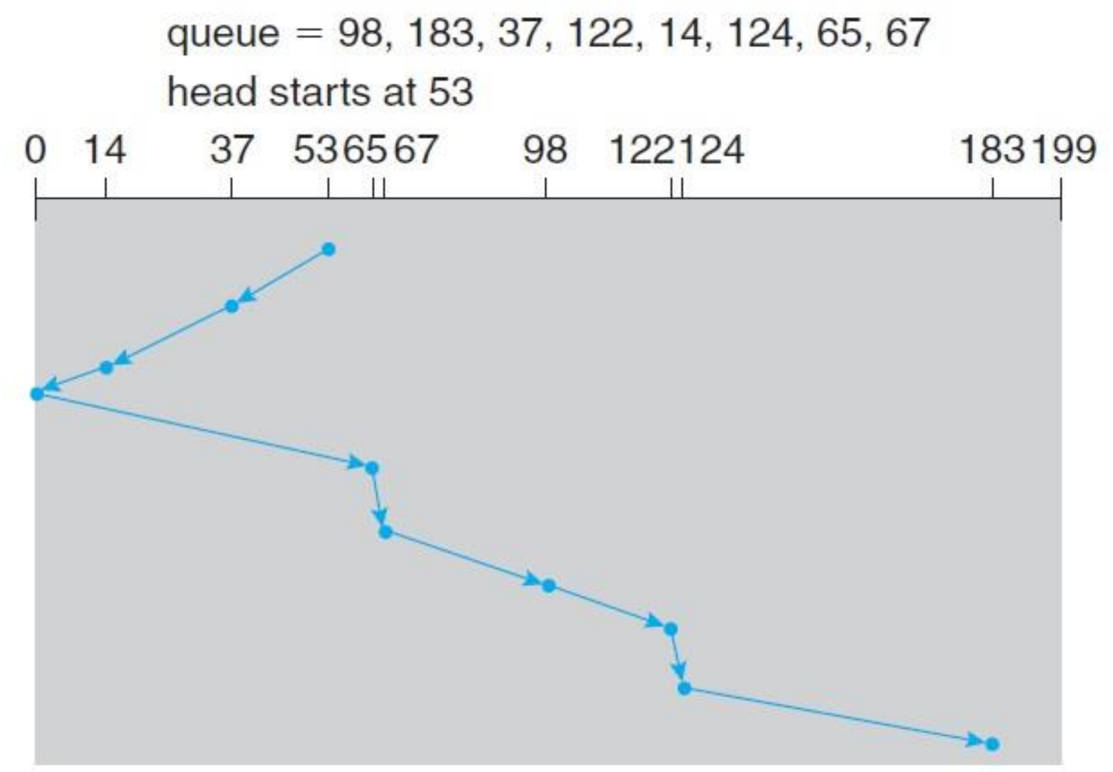


Figure 9-4 SCAN disk scheduling

9.3.4. C-SCAN Scheduling

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip (Figure 9.5). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

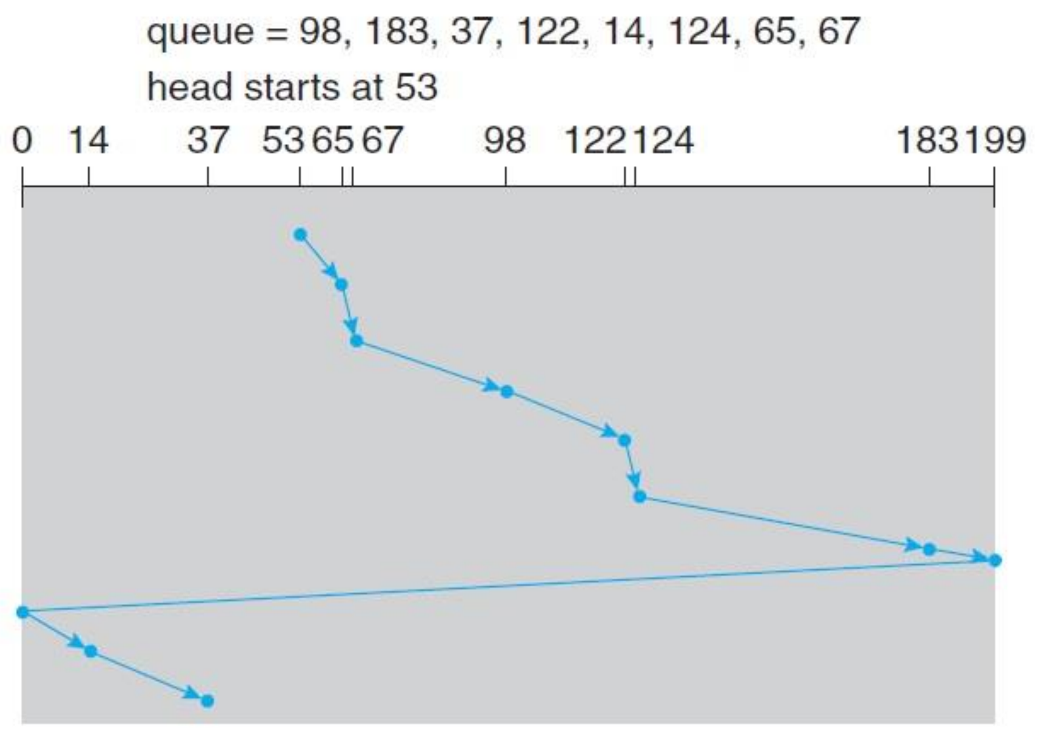


Figure 9-5 C-SCAN disk scheduling

9.3.5. LOOK Scheduling

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling**, because they ***look*** for a request before continuing to move in a given direction (Figure 9.6).

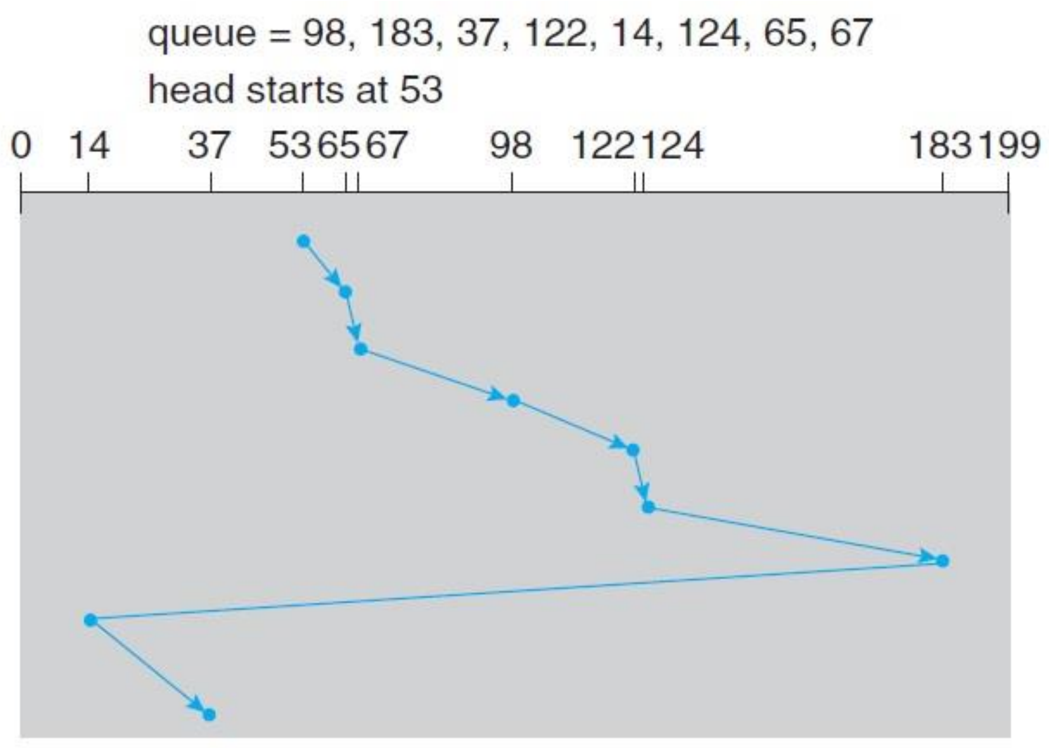


Figure 9-6 C-LOOK disk scheduling