

Introduction to Operating System Concepts

Operating systems concepts, Operating systems services, Types of operating systems, Structure of operating System, Introduction to System call, Types of system calls.

Introduction

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. Operating System is computer software that manages the hardware components. It is responsible for managing the system resources and providing a smooth working environment for the users.

The management includes the following – process management, processor management, memory management, storage management, user management, protection and security.



Operating systems concepts

- An operating system is a program that acts as an intermediary between the user of a computer and the computer hardware.
- The purpose of an Operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
 - Execute user programs and make solving user problems easier
 - Make the computer system convenient to use
 - Use the computer hardware in an efficient manner

Computer System Structure

- Computer system can be divided into four components:
 - Hardware – provides basic computing resources
 - CPU, memory, I/O devices
 - Operating system
 - Controls and coordinates use of hardware among various applications and users

- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
 - Word processors, compilers, web browsers, database systems, video games
- Users
 - People, machines, other computers

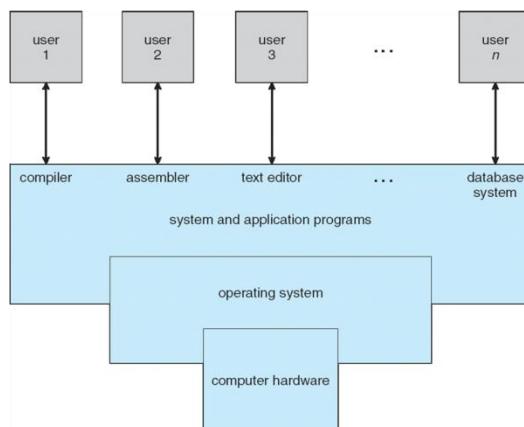


Fig: Components of a computer System

Definition:

- OS is a resource allocator
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
- OS is a control program
 - Controls execution of programs to prevent errors and improper use of the computer

A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel)

Operating System Services

- An Operating System provides services to both the users and to the programs.
- ✓ It provides programs, an environment to execute.
- ✓ It provides users, services to execute the programs in a convenient manner.
- The specific services provided, of course, differ from one operating system to another, but we can identify common classes.
- These operating system services are provided for the convenience of the programmer, to make the programming task easier.
- Following are few common services provided by operating systems.
 - ✓ User Interface
 - ✓ Program execution
 - ✓ I/O operations
 - ✓ File System manipulation
 - ✓ Communications
 - ✓ Error Detection
 - ✓ Resource Allocation

- ✓ Accounting
- ✓ Protection

The Figure below shows one view of the various operating-system services and how they interrelate.

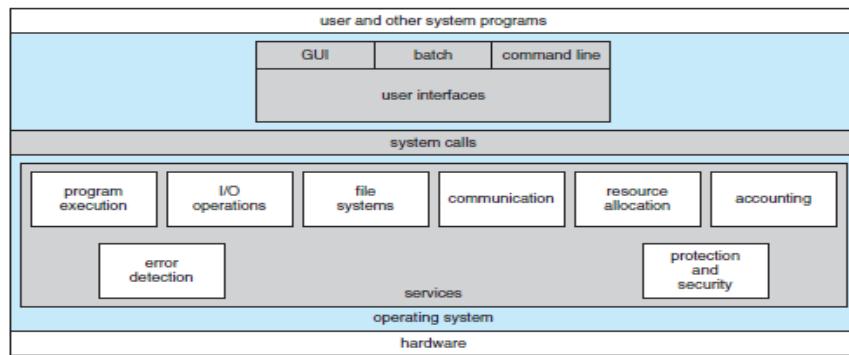


Figure A view of operating system services.

One set of operating system services provides functions that are helpful to the user.

User Interface: Almost all operating systems have a user interface (UI). This interface can take several forms. One is a command-line interface (CLI), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a batch interface, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a graphical user interface (GUI) is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text. Some systems provide two or all three of these variations.

Program execution: The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

I/O operations: A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

File System manipulation: Programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include permissions management to allow or deny access to files or directories based on file ownership. Many operating systems provide a variety of file systems, sometimes to allow personal choice and sometimes to provide specific features or performance characteristics.

Communications: There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via shared memory, in which two or more processes read and write to a shared section of memory, or message passing, in which packets of information in predefined formats are moved between processes by the operating system.

Error Detection: The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use

of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct it.

Another set of operating system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

Resource Allocation: When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code. For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the jobs that must be executed, the number of registers available, and other factors. There may also be routines to allocate printers, USB storage drives, and other peripheral devices.

Accounting: We want to keep track of which users use how much and what kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

Protection and security: The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself or herself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and to recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it.

Types of Operating Systems

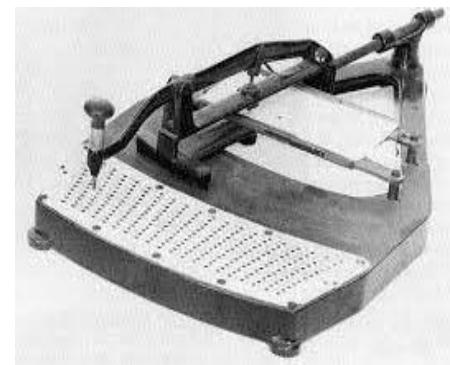
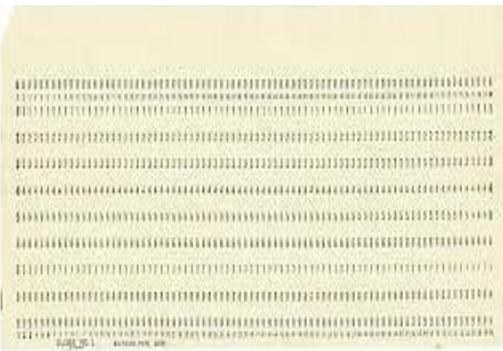
- 1) Simple Batch Systems
- 2) Multi-programmed Batched Systems
- 3) Time-sharing Systems
- 4) Personal-Computer Systems
- 5) Parallel Systems
- 6) Distributed Systems
- 7) Real Time operating Systems

Simple Batch Systems

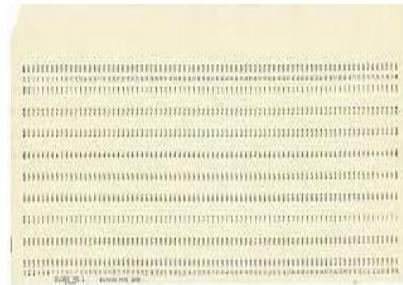
- Early computers were enormously large machines (physically).



- The common input devices were card readers and tape drives.



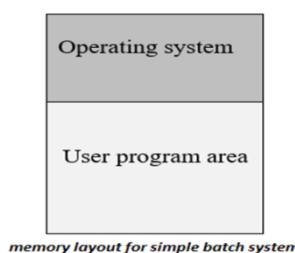
- The common output devices were line printers, tape drives and card punches.



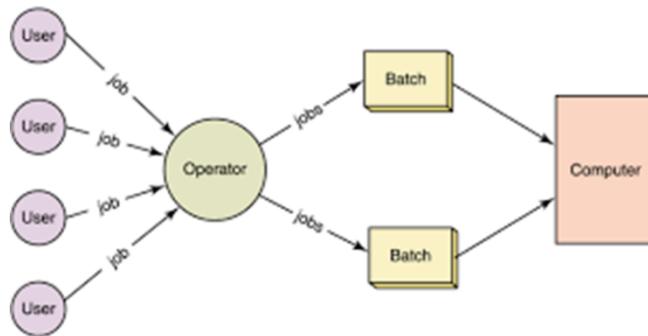
The users of batch operating system did not interact with the computer directly. The user prepared a job – which consisted of the program, the data and some control information about the nature of the job and submitted it to the computer operator. The job will be usually in the form of punch cards. Ex: Payroll System, Bank Statements etc. At some later time the output appeared.

The main task of OS was to transfer control automatically from one job to the next.

The OS was always resident in the memory.

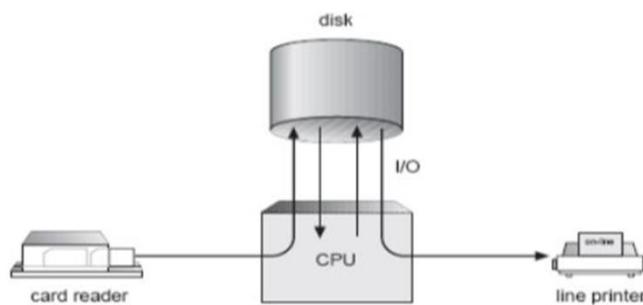


- To speed up processing jobs with similar needs were batched together and were run through the computer as a group.



Disadvantages:

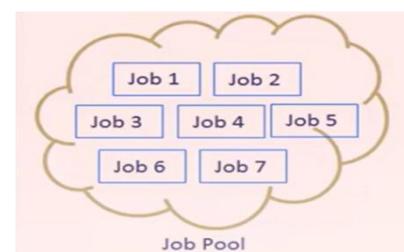
- ✓ Lack of interaction between the user and job.
- ✓ CPU is often idle, because the speeds of the mechanical I/O devices are slower than CPU.

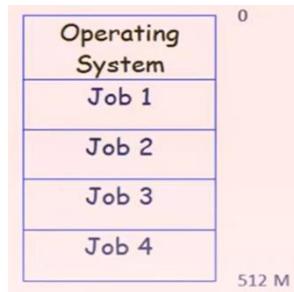


Simultaneous peripheral operation on-line (Spooling)

Multi-programmed Batched Systems

- Spooling provides an important data structure: *job pool*.
- Job Scheduling
- The most important aspect of Job scheduling is the ability to Multiprogram.
- CPU Scheduling

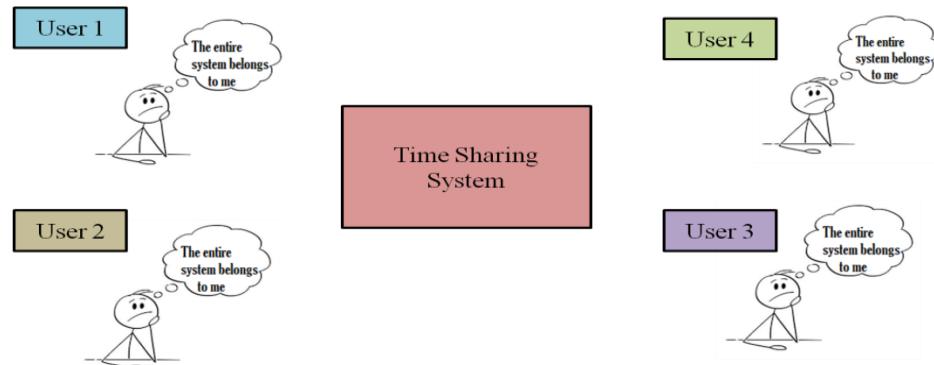




Memory Layout for Multiprogrammed System

Time Sharing Systems

Also referred as multitasking. It is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them but the switches occur so frequently that the users may interact with each program while it is running. An interactive or hands-on computer system provides on-line communication between the user and the system.



Personal Computer Systems

A computer system dedicated to a single user. First appeared in 1970's. They are micro computers that are considerably smaller and less expensive than mainframe systems.



Parallel Systems

- Multiprocessor system sharing the computer bus, memory and peripheral devices.
- Also referred to as tightly coupled systems.

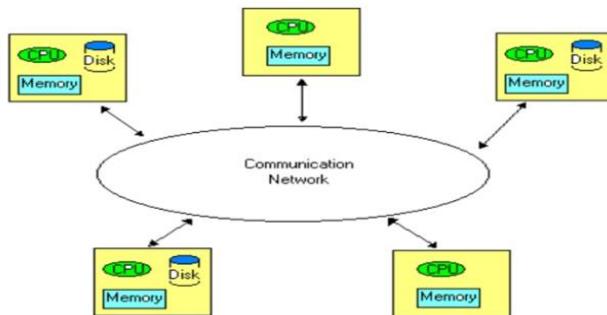
Advantages:

- Increased Throughput
- Save money
- Increase reliability.

Distributed Systems

- In contrast to the tightly coupled systems the processors do not share memory.
- Instead each processor has its own local memory.
- The processors communicate through telephone lines or high-speed buses.
- Usually referred as loosely coupled systems

Architecture of Distributed OS



Reasons for building Distributed Systems (Advantages):

- ✓ **Resource Sharing:** With resource sharing facility user at one site may be able to use the resources available at another.
- ✓ **Computation Speedup:** Computation can be distributed among various sites to run that computation concurrently.
- ✓ **Reliability:** If one site fails in a distributed system, the remaining sites can potentially continue operating.
- ✓ **Communication:** Users may initiate file transfers or communicates with one another via electronic mail.

Real Time Systems

- Real-time operating systems (RTOS) are used in environments where a large number of events, mostly external to the computer system, must be accepted and processed in a short time or within certain deadlines.
- A real –time operating system has well-defined fixed time constraints.
- This system is time-bound and has a fixed deadline.
- The processing in this type of system must occur within the specified constraints. Otherwise, this will lead to system failure.
- Examples of the real-time operating systems: Airline traffic control systems, Command Control Systems, Medical imaging systems, Home-appliance controllers, Robots etc.

The real-time operating systems can be of 2 types –

- *Hard real-time*
- *Soft real-time*

Hard Real-Time operating system:

- These operating systems guarantee that critical tasks be completed within a range of time.
- Example: Robot in a war field.

Soft real-time operating system:

- This operating system provides some relaxation in the time limit.
- Example: Multimedia, under sea exploration, planetary rovers etc..

Structure of Operating system

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

Simple Structure (Monolithic Structure)

- Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. **MS-DOS** is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not carefully divided into modules. Figure below shows its structure.

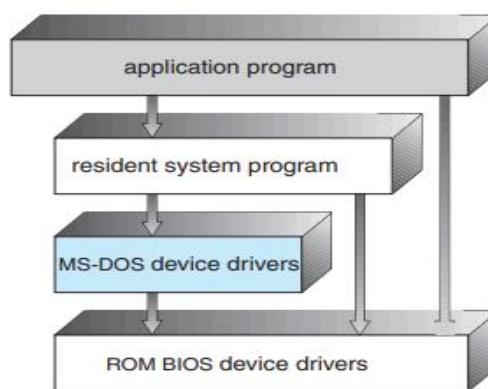


Figure MS-DOS layer structure.

- In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.
- Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure below.
- Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level.

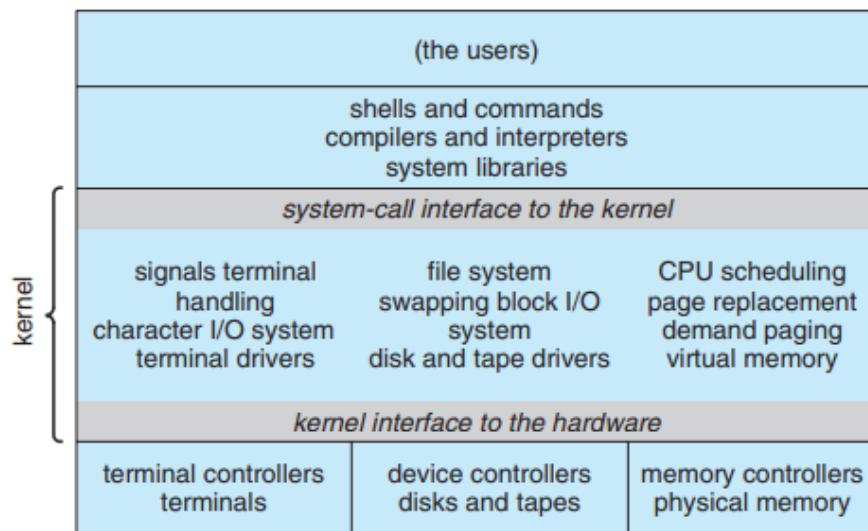


Figure Traditional UNIX system structure.

- This **monolithic structure** was difficult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel.

Layered Approach

- With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS and UNIX systems. The operating system can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems.
- A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure below.

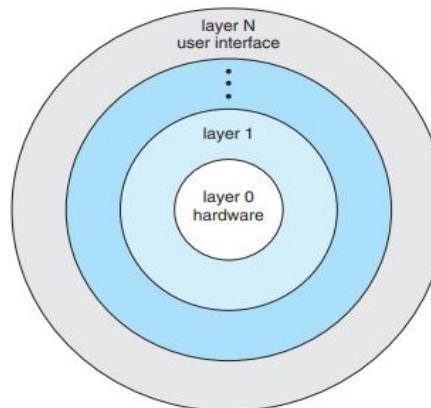


Figure A layered operating system.

- An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.
- The main advantage of the layered approach is **simplicity of construction and debugging**. The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified.
- The major difficulty with the layered approach involves appropriately **defining the various layers**. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

Microkernels

- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through **message passing**.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

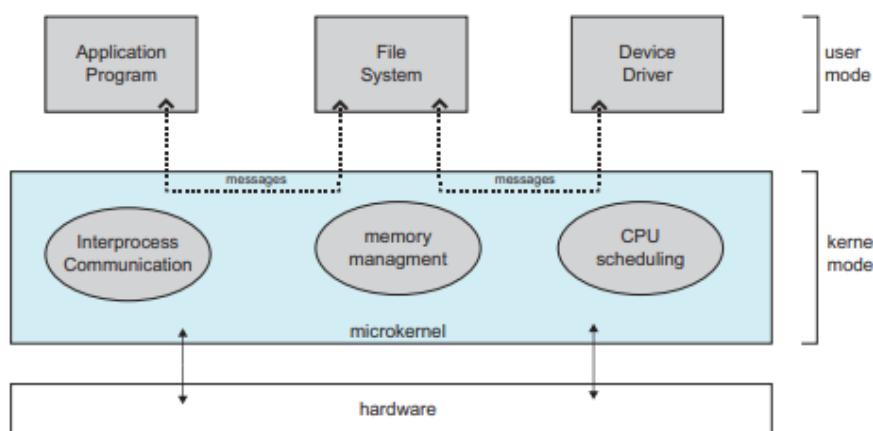


Figure Architecture of a typical microkernel.

- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.

When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

- Unfortunately, the performance of microkernels can suffer due to increased system-function overhead.

Modules

- Perhaps the best current methodology for operating-system design involves using loadable kernel modules. Here, the kernel has a set of core components and links in additional services via modules, either at boot time or during run time. This type of design is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X, as well as Windows.
- The idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running. Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.
- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module. The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

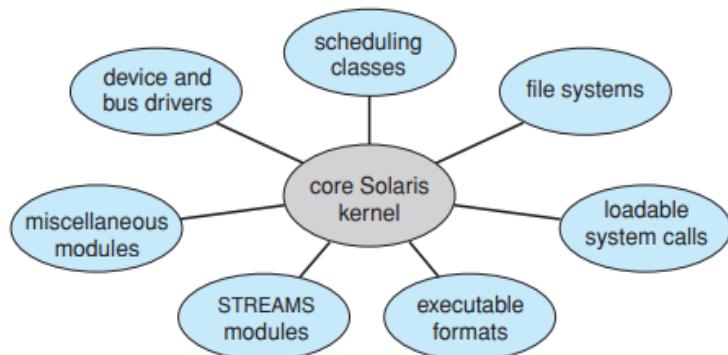


Figure Solaris loadable modules.

- The Solaris operating system structure, shown in Figure above, is organized around a core kernel with seven types of loadable kernel modules: **1. Scheduling classes 2. File systems 3. Loadable system calls 4. Executable formats 5. STREAMS modules 6. Miscellaneous 7. Device and bus driver**

Hybrid Systems

- In practice, very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, both Linux and Solaris are monolithic, because having the operating system in a single address space provides very efficient performance. However, they are also

modular, so that new functionality can be dynamically added to the kernel. Windows is largely monolithic as well (again primarily for performance reasons), but it retains some behavior typical of microkernel systems, including providing support for separate subsystems (known as operating-system personalities) that run as user-mode processes. Windows systems also provide support for dynamically loadable kernel modules.

- Operating-System Structures three hybrid systems: **the Apple Mac OS X operating system and the two most prominent mobile operating systems—iOS and Android.**

Introduction to system call

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.
- 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). This system-call sequence is shown in Figure below.

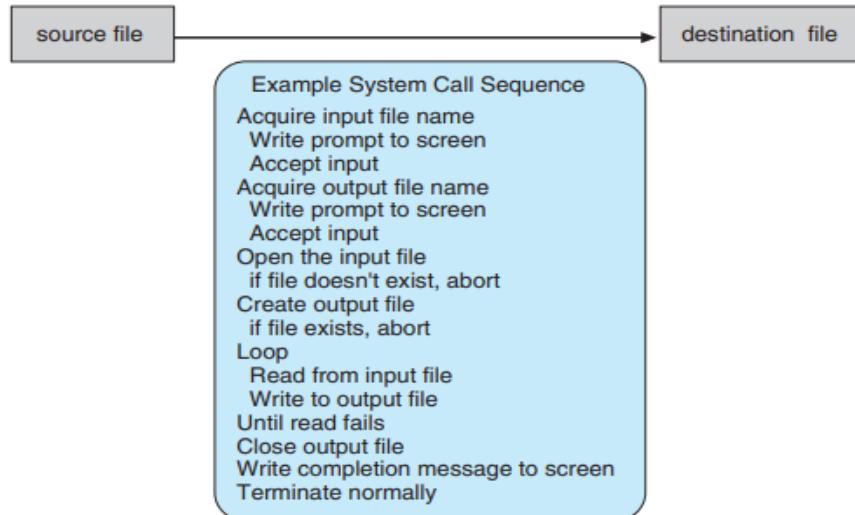


Figure Example of how system calls are used.

Even simple programs may make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. Most programmers never see this level of detail, however. Typically, application developers design programs according to an application programming interface (API). The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. Three of the most common APIs available to application programmers are the Windows API for Windows systems, the POSIX API for POSIX-based systems (which include virtually all versions of UNIX, Linux, and Mac OS X), and the Java API for programs that run on the Java virtual machine.

Types of system calls

System calls can be grouped roughly into six major categories: **process control, file manipulation, device manipulation, information maintenance, communications, and protection.**

- Process control
 - end, abort
 - load, execute
 - create process, terminate process
 - get process attributes, set process attributes
 - wait for time
 - wait event, signal event
 - allocate and free memory
- File management
 - create file, delete file
 - open, close
 - read, write, reposition
 - get file attributes, set file attributes
- Device management
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
- Information maintenance
 - get time or date, set time or date
 - get system data, set system data
 - get process, file, or device attributes
 - set process, file, or device attributes
- Communications
 - create, delete communication connection
 - send, receive messages
 - transfer status information
 - attach or detach remote devices

Figure Types of system calls.

Process Control

- A running program needs to be able to halt its execution either normally (**end()**) or abnormally (**abort()**). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs—to determine the cause of the problem.

- A process or job executing one program may want to **load()** and **execute()** another program.
- If control returns to the existing program when the new program terminates, we must save the memory image of the existing program; thus, we have effectively created a mechanism for one program to call another program. If both programs continue concurrently, we have created a new job or process to be multiprogrammed. Often, there is a system call specifically for this purpose (**create process()** or **submit job()**).
- If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (**get process attributes()** and **set process attributes()**).
- We may also want to terminate a job or process that we created (**terminate process()**) if we find that it is incorrect or is no longer needed.
- Having created new jobs or processes, we may need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (**wait time()**). More probably, we will want to wait for a specific event to occur (**wait event()**). The jobs or processes should then signal when that event has occurred (**signal event()**).

File Management

- We first need to be able to **create()** and **delete()** files. Either system call requires the name of the file and perhaps some of the file's attributes.
- Once the file is created, we need to **open()** it and to use it. We may also **read()**, **write()**, or **reposition()** (rewind or skip to the end of the file, for example).
- Finally, we need to **close()** the file, indicating that we are no longer using it.
- We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system.
- In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, **get file attributes()** and **set file attributes()**, are required for this function.
- Some operating systems provide many more calls, such as calls for **file move()** and **copy()**.

Device Management

- The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first **request()** a device, to ensure exclusive use of it. After we are finished with the device, we **release()** it.
- Once the device has been requested (and allocated to us), we can **read()**, **write()**, and (possibly) **reposition()** the device, just as we can with files.

Information Maintenance

- Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the **current time()** and **date()**. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

- Another set of system calls is helpful in debugging a program. Many systems provide system calls to **dump()** memory. This provision is useful for debugging.
- In addition, the operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (**get process attributes()** and **set process attributes()**).

Communication

There are two common models of interprocess communication: the **message-passing model** and the **shared-memory model**. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network. Each computer in a network has a **host name** by which it is commonly known. A host also has a network identifier, such as an **IP address**. Similarly, each process has a **process name**, and this name is translated into an identifier by which the operating system can refer to the process.

- The **get hostid()** and **get processid()** system calls do this translation. The identifiers are then passed to the general purpose **open()** and **close()** calls provided by the file system or to specific **open connection()** and **close connection()** system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an **accept connection()** call.
- Most processes that will be receiving connections are special-purpose **daemons**, which are system programs provided for that purpose. They execute a **wait for connection()** call and are awakened when a connection is made. The source of the communication, known as **the client**, and the receiving daemon, known as a **server**, then exchange messages by using **read message()** and **write message()** system calls. The **close connection()** call terminates the communication.
- In the shared-memory model, processes use **shared memory create()** and **shared memory attach()** system calls to create and gain access to regions of memory owned by other processes.

Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

- Typically, system calls providing protection include **set permission()** and **get permission()**, which manipulate the permission settings of resources such as files and disks. The **allow user()** and **deny user()** system calls specify whether particular users can—or cannot—be allowed access to certain resources.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Process Management – Process concept, The process, Process State Diagram, Process control block, Process Scheduling- Scheduling Queues, Schedulers, Operations on Processes, Inter-process Communication, Threading Issues, Scheduling-Basic Concepts, Scheduling Criteria, Scheduling Algorithms.

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.

A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.

Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.

The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Process Concept

A batch system executes jobs, whereas a time-shared 105 106 Chapter 3 Processes 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 Process

A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section**. It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables. A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure below.

We emphasize that a program by itself is not a process. A program is a **passive entity**, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an **active entity**, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

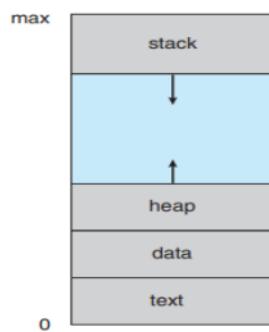


Figure Process in memory.

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.

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 below:

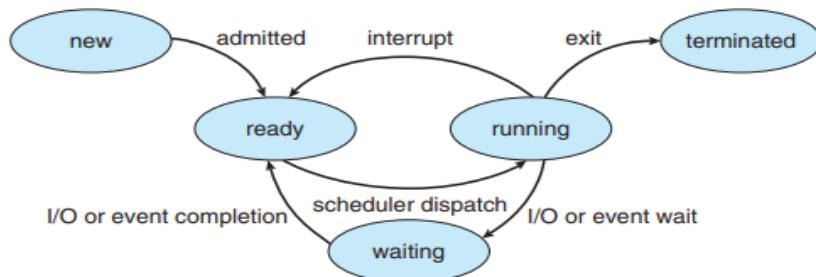


Figure Diagram of process state.

Process Control Block

Each process is represented in the operating system by a process control block (PCB)—also called a task control block. A PCB is shown in Figure below. It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on. In brief, the PCB simply serves as the repository for any information that may vary from process to process

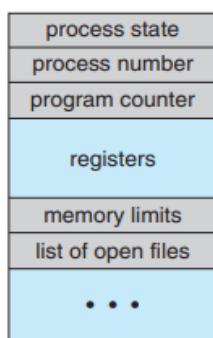


Figure 2 Process control block (PCB).

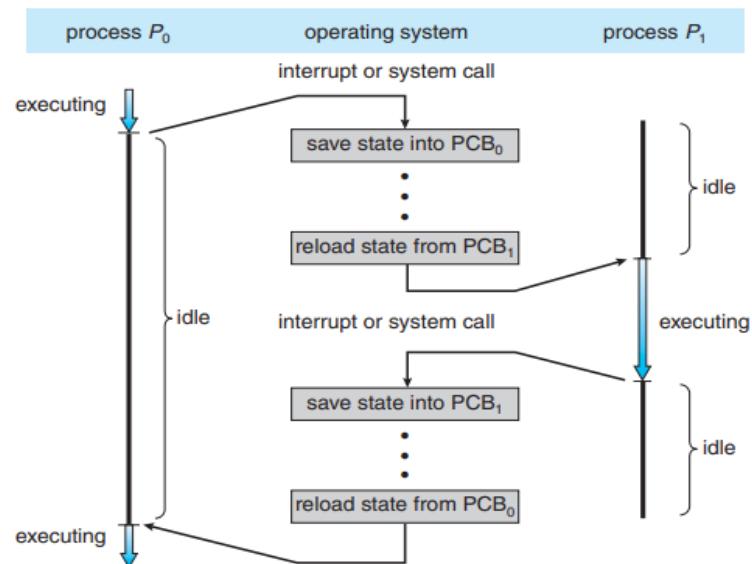


Figure 3 Diagram showing CPU switch from process to process.

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.

Scheduling Queues

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.

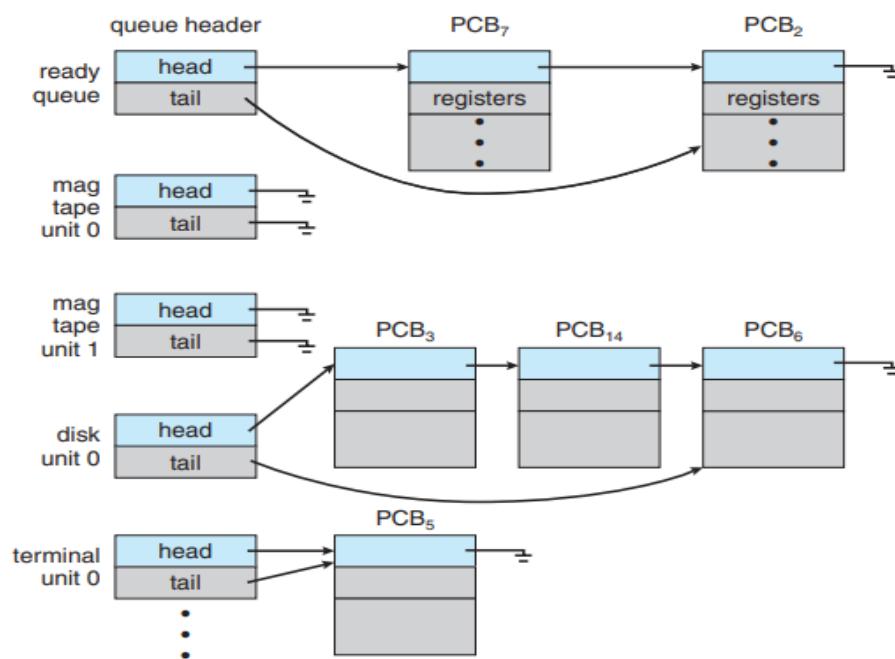


Figure The ready queue and various I/O device queues.

A common representation of process scheduling is a **queueing diagram**, such as that in Figure below. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing, one of several events could occur:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.

- 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.

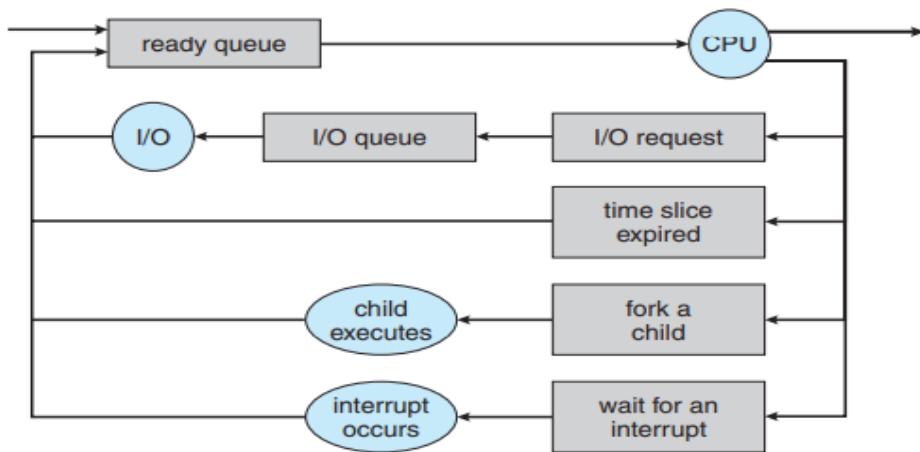


Figure Queueing-diagram representation of process scheduling.

Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler, or job scheduler**, selects processes from this pool and loads them into memory for execution. The **short-term scheduler, or CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The primary distinction between these two schedulers lies in frequency of execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then $10/(100 + 10) = 9$ percent of the CPU is being used (wasted) simply for scheduling the work.

The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the **degree of multiprogramming** is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

It is important that the long-term scheduler 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**, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in Figure below. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

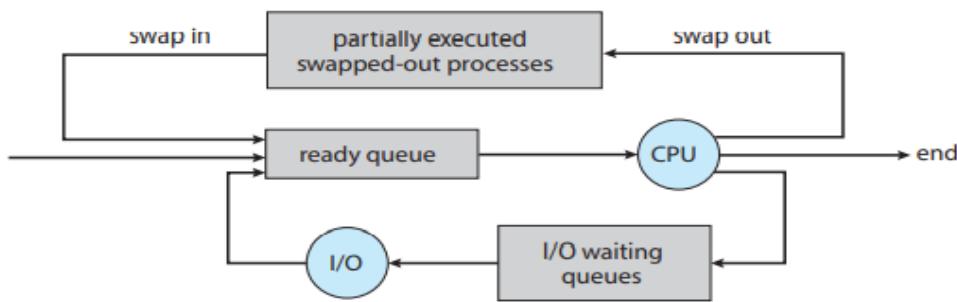


Figure Addition of medium-term scheduling to the queueing diagram.

Context Switch

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information. Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. As we will see in Chapter 8, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system.

Operations on Process

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination. In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifier (or **pid**), which is typically an integer number. The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.

The following figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid. (We use the term process rather loosely, as Linux prefers the term task instead.) The **init** process (which always has a pid of 1) serves as the root parent process for all user processes. Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server, and the like. In the figure, we see two children of init—**kthreadd** and **sshd**. The **kthreadd** process is responsible for creating additional processes that perform tasks on behalf of the kernel (in this situation, **khelper** and **pdflush**). The **sshd** process is responsible for managing clients that connect to the system by using ssh (which is short for secure shell). The **login** process is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the **bash** shell, which has been assigned pid 8416. Using the **bash** command-line interface, this user has created the process **ps** as well as the **emacs** editor.

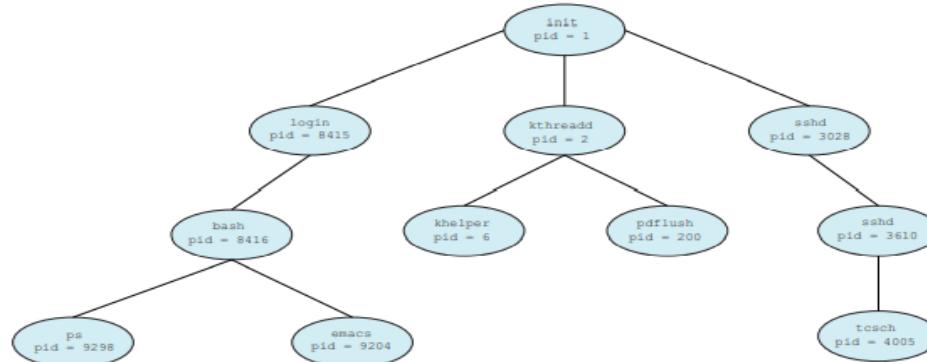


Figure A tree of processes on a typical Linux system.

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes. In addition to supplying various physical and logical resources, the parent process may pass along initialization data (input) to the child process.

When a process creates a new process, two possibilities for execution exist:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two address-space possibilities for the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit()** system call. At that point, the process may return a status value (typically an integer) to its parent process (via the **wait()** system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, **TerminateProcess()** in Windows). Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls **wait()**, because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called **wait()**, is known as a **zombie process**.

Interprocess Communication

Processes executing concurrently in the operating system may be either **independent processes** or **cooperating** processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data 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. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.

- **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
- **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: **shared memory** and **message passing**. In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in Figure below.

Both of the models just mentioned are common in operating systems, and many systems implement both. Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory. Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

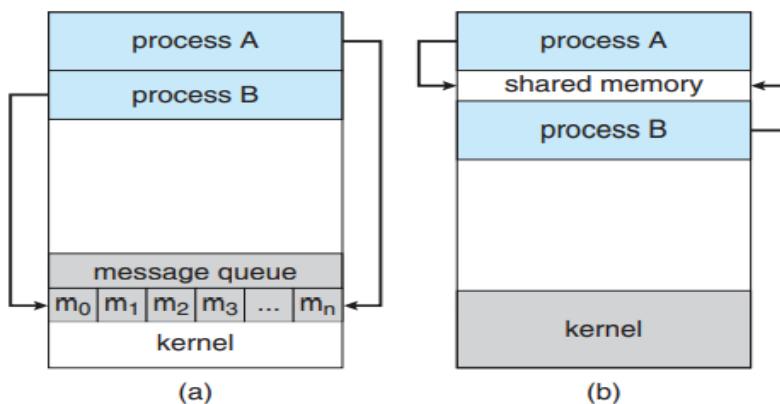


Figure Communications models. (a) Message passing. (b) Shared memory.

Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer–consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.

One solution to the producer–consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations:

- *send(message)*
- *receive(message)*

Messages sent by a process can be either fixed or variable in size. If only fixed-sized messages can be sent, the system-level implementation is straight forward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating-system design.

If processes P and Q want to communicate, they must send messages to and receive messages from each other: a **communication link** must exist between them. This link can be implemented in a variety of ways. Here are several methods for logically implementing a link and the send ()/receive () operations:

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

Issues related with Message Passing

Naming

Processes that want to communicate must have a way to refer to each other. They can use either **direct** or **indirect** communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:

- *send(P, message)*—Send a message to process P.
- *receive(Q, message)*—Receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link

This scheme exhibits symmetry in addressing; that is, both the sender process and the receiver process must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:

- **send(*P, message*)**—Send a message to process P.
- **receive(*id, message*)**—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

With indirect communication, the messages are sent to and received from **mailboxes, or ports**. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox.

The send() and receive() primitives are defined as follows:

- **send(*A, message*)**—Send a message to mailbox A.
- **receive(*A, message*)**—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if both members of the pair have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

Synchronization

Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**—also known as **synchronous** and **asynchronous**.

- **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
- **Nonblocking send.** The sending process sends the message and resumes operation.
- **Blocking receive.** The receiver blocks until a message is available.
- **Nonblocking receive.** The receiver retrieves either a valid message or a null.

Buffering

Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity.** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks. The zero-capacity case is sometimes referred to as a message system with no buffering.

Threading Issues

Overview

A thread is a basic unit of CPU utilization; it comprises a **thread ID**, a **program counter**, a **register set**, and a **stack**. It shares with other threads belonging to the same process its **code section**, **data section**, and other operating-system resources, such as **open files and signals**. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure below illustrates the difference between a traditional single-threaded process and a multithreaded process.

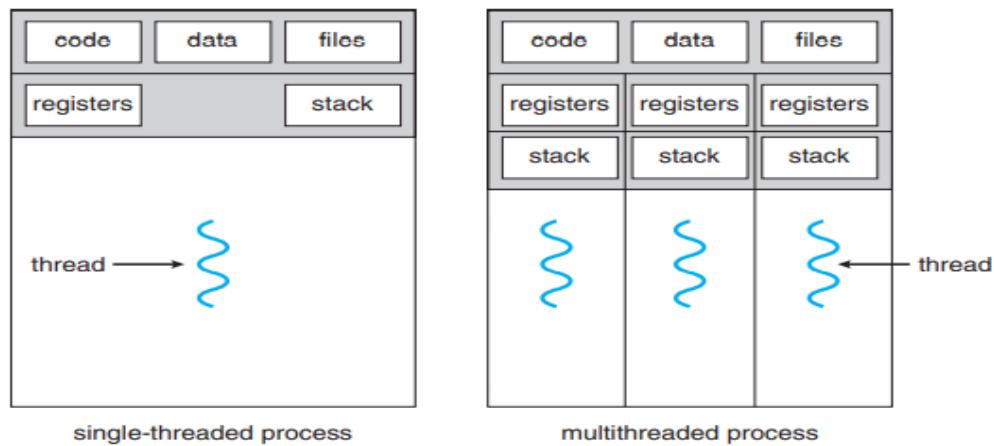


Figure Single-threaded and multithreaded processes.

Benefits

The benefits of multithreaded programming can be broken down into four major categories:

- 1. Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.
- 2. Resource sharing.** Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- 3. Economy.** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- 4. Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

Multithreading Models

Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris—support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship: **the many-to-one model, the one-to-one model, and the many-to-many model.**

Many-to-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

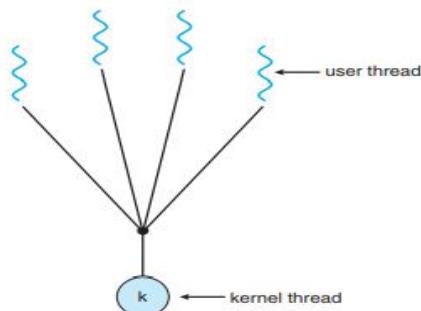


Figure Many-to-one model.

One-to-One Model

The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

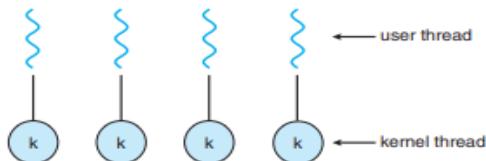


Figure One-to-one model.

Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).

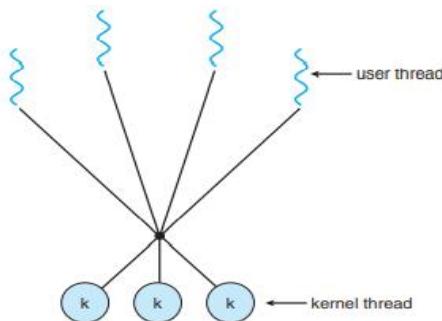


Figure Many-to-many model.

Threading Issues

The fork() and exec() System Calls

The `fork()` system call is used to create a separate, duplicate process. The semantics of the `fork()` and `exec()` system calls change in a multithreaded program.

If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call. If a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will replace the entire process—including all threads.

Which of the two versions of `fork()` to use depends on the application. If `exec()` is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to `exec()` will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec()` after forking, the separate process should duplicate all threads.

Signal Handling

A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Thread Cancellation

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page

from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation.** One thread immediately terminates the target thread.
2. **Deferred cancellation.** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

Basic Concepts

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.

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.

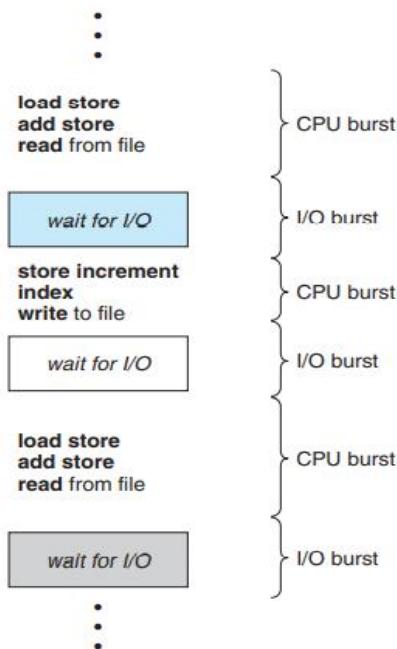


Figure Alternating sequence of CPU and I/O bursts.

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.

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.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Preemption also affects the design of the operating-system kernel.

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

Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor 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.

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.

First-Come, First-Served Scheduling

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.

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 it 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.

Shortest-Job-First Scheduling

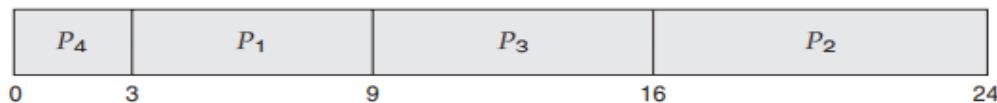
A different approach to CPU scheduling is the shortest-job-first(SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of 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
P_1	6
P_2	8
P_3	7
P_4	3

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



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds 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 a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

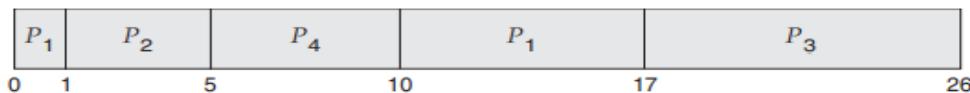
The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

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:

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
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 P_1 is started at time 0, since it is the only process in the queue. Process P_2 arrives at time 1. The remaining time for process P_1 (7 milliseconds) is larger than the time required by process P_2 (4 milliseconds), so process P_1 is preempted, and process P_2 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.

Priority Scheduling

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority 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.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P_1, P_2, \dots, P_5 , with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
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. 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 , or the computer system will eventually crash and lose all unfinished low-priority processes.

A solution to the problem of indefinite 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.

Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for time sharing 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 from 10 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.

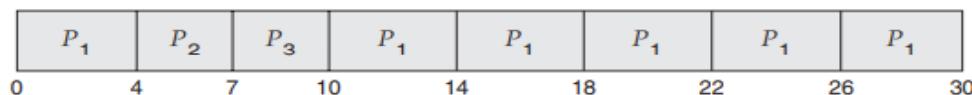
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

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:



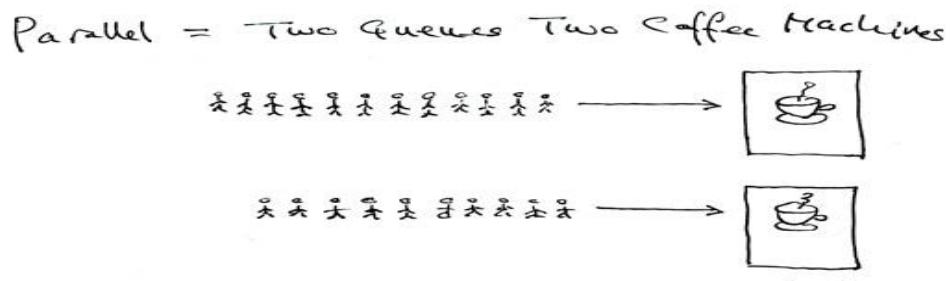
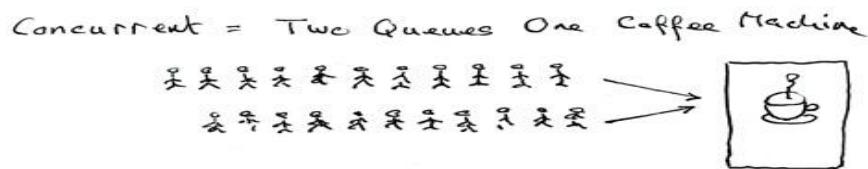
Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds ($10 - 4$), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is $17/3 = 5.66$ milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 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.

Concurrency: Process Synchronization, The Critical-Section Problem, Synchronization Hardware, Semaphores, Classic Problems of Synchronization, Monitors.

Principles of deadlock – System Model, Deadlock Characterization, Deadlock Prevention, Detection and Avoidance, Recovery from Deadlock.

Concurrency

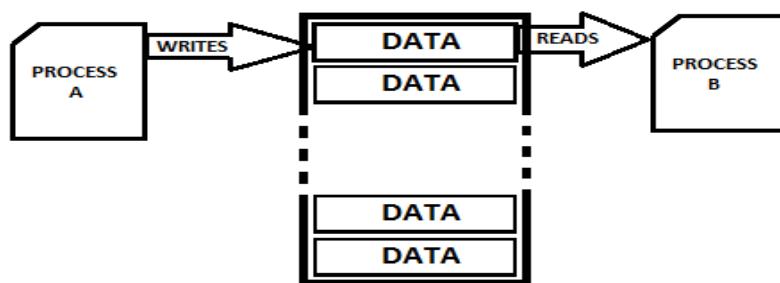
A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads. Concurrent access to shared data may result in data inconsistency, however. Various mechanisms are used to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.



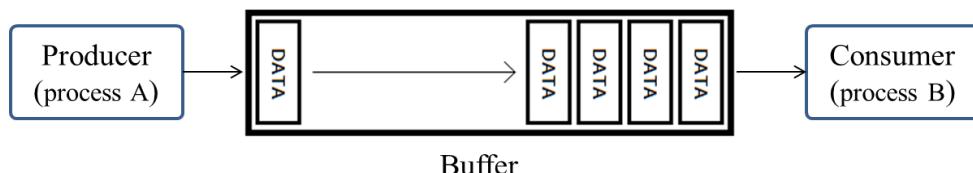
© Joe Armstrong 2013

Process Synchronization

One process changing the data in a memory location where another process is trying to read the data from the same memory location. **It is possible that the data read by the second process will be erroneous.**



PRODUCER CONSUMER PROBLEM (or Bounded-Buffer Problem)



- Problem:
 - To ensure that the Producer should not add DATA **when the Buffer is full** and the Consumer should not take data **when the Buffer is empty**.
- Providing a solution to the producer-consumer problem that fills all the buffers.
- Having an **integer counter** that keeps track of the number of full buffers.
- Initially, counter is set to 0.
- The producer produces a new buffer: **increment the counter**
- The consumer consumes a buffer: **decrement the counter**

We now return to our consideration of the bounded buffer. As we pointed out, our original solution allowed at most BUFFER SIZE – 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

The code for the producer process can be modified as follows:

```
while (true) {
    /* produce an item in next_produced */

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
    while (counter == 0)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next_consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements “counter++” and “counter-

-. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately

We can show that the value of counter may be incorrect as follows. Note that the statement “counter++” may be implemented in machine language (on a typical machine) as follows:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

where register1 is one of the local CPU registers.

Similarly, the statement “counter--” is implemented as follows:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

where again register2 is one of the local CPU registers

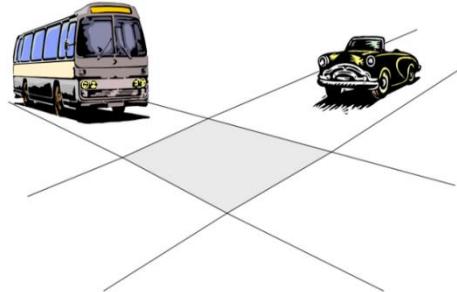
The concurrent execution of “counter++” and “counter--” is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

T ₀ :	producer	execute	register ₁ = counter	{register ₁ = 5}
T ₁ :	producer	execute	register ₁ = register ₁ + 1	{register ₁ = 6}
T ₂ :	consumer	execute	register ₂ = counter	{register ₂ = 5}
T ₃ :	consumer	execute	register ₂ = register ₂ - 1	{register ₂ = 4}
T ₄ :	producer	execute	counter = register ₁	{counter = 6}
T ₅ :	consumer	execute	counter = register ₂	{counter = 4}

Notice that we have arrived at the incorrect state “counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state “counter == 6”.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A *situation like this, where 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.

The Critical-Section Problem



Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$. 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 P_i is shown in the Figure below. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

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

Figure General structure of a typical process P_i .

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

1. **Mutual exclusion.** If process P_i 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 process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems: **preemptive** kernels and **nonpreemptive** kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Petersons Solution

It is a classic software-based solution to the critical-section problem. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered **P0** and **P1**. For convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals $1 - i$.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable **turn** indicates whose turn it is to enter its critical section. That is, if **turn == i**, then process P_i is allowed to execute in its critical section. The **flag** array is used to indicate if a process is ready to enter its critical section. For example, if $\text{flag}[i]$ is true, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure below.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = false;

    remainder section

} while (true);
```

Figure The structure of process P_i in Peterson's solution.

To enter the critical section, process P_i first sets $\text{flag}[i]$ to be true and then sets turn to the value j , thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either $\text{flag}[j] == \text{false}$ or $\text{turn} == i$. Also note that, if both processes can be executing in their critical sections at the same time, then $\text{flag}[0] == \text{flag}[1] == \text{true}$. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes —say, P_j —must have successfully executed the while statement, whereas P_i had to execute at least one additional statement (“ $\text{turn} == j$ ”). However, at that time, $\text{flag}[j] == \text{true}$ and $\text{turn} == j$, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition $\text{flag}[j] == \text{true}$ and $\text{turn} == j$; this loop is the only one possible. If P_j is not ready to enter the critical section, then $\text{flag}[j] == \text{false}$, and P_i can enter its critical section. If P_j has set $\text{flag}[j]$ to true and is also executing in its while statement, then either $\text{turn} == i$ or $\text{turn} == j$. If $\text{turn} == i$, then P_i will enter the critical section. If $\text{turn} == j$, then P_j will enter the critical section. However, once P_j exits its critical section,

it will reset flag[j] to false, allowing Pi to enter its critical section. If Pj resets flag[j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

Synchronization Hardware

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also consider the effect on a system's clock if the clock is kept updated by interrupts.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically—that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, we abstract the main concepts behind these types of instructions by describing the **test and set()** and **compare and swap()** instructions.

The test and set() instruction can be defined as shown in Figure below.

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;

    return rv;
}
```

Figure The definition of the **test_and_set()** instruction.

The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the test and set() instruction, then we can implement mutual exclusion by declaring a boolean variable lock, initialized to false. The structure of process Pi is shown in Figure below.

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

Figure Mutual-exclusion implementation with **test_and_set()**.

The **compare and swap()** instruction, in contrast to the test and set() instruction, operates on three operands; it is defined in Figure below.

```

int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}

```

Figure The definition of the `compare_and_swap()` instruction.

The operand value is set to new value only if the expression (`*value == expected`) is true. Regardless, `compare and swap()` always returns the original value of the variable `value`. Like the `test and set()` instruction, `compare and swap()` is executed atomically.

Mutual exclusion can be provided as follows: a global variable (`lock`) is declared and is initialized to 0. The first process that invokes `compare and swap()` will set `lock` to 1. It will then enter its critical section, because the original value of `lock` was equal to the expected value of 0. Subsequent calls to `compare and swap()` will not succeed, because `lock` now is not equal to the expected value of 0. When a process exits its critical section, it sets `lock` back to 0, which allows another process to enter its critical section. The structure of process `Pi` is shown in Figure below.

```

do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
} while (true);

```

Figure Mutual-exclusion implementation with the `compare_and_swap()` instruction

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

Mutex Locks

The hardware-based solutions to the critical-section problem are complicated as well as generally inaccessible to application programmers. Instead, operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.) We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The `acquire()` function acquires the lock, and the `release()` function releases the lock, as illustrated in Figure below.

```

do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);

```

Figure 5. Solution to the critical-section problem using mutex locks.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released. The definition of acquire() is as follows:

```

acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}

```

The definition of release() is as follows:

```

release() {
    available = true;
}

```

Calls to either acquire() or release() must be performed atomically. The main disadvantage of the implementation given here is that it requires busy waiting.

Semaphores

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait()** and **signal()**. The wait() operation was originally termed **P** (from the Dutch proberen, “to test”); signal() was originally called **V** (from verhogen, “to increment”). The P() operation decrements the value of the semaphore and blocks if its new value is less than zero.

The definition of wait() is as follows:

```

wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}

```

(OR)

```
P(S)
{
while (S <= 0)
/* no-operation*/
S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
    S++;
}
```

(OR)

```
V(S)
{
    S++;
}
```

The V() operation increments its value; if the resulting value becomes greater than or equal to zero.

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification ($S--$), must be executed without interruption.

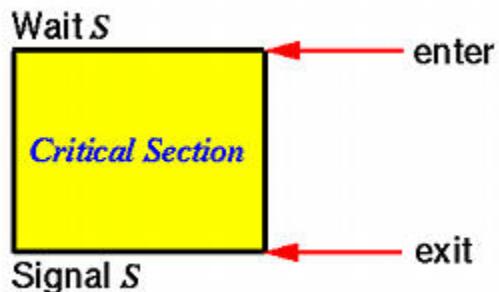
Semaphore Usage

Operating systems often distinguish between **counting** and **binary** semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

The n processes share a semaphore, mutex, initialized to 1.

```
do
{
    waiting(mutex);
    //Critical Section
    signal(mutex)
    //remainder section
}while (True);
```



Semaphore Implementation with no busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)**
 - pointer to next record in the list**
- Two operations:
 - block** – place the process invoking the operation in the appropriate waiting queue
 - wakeup** – remove one of the processes in the waiting queue and place it in the ready queue

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process. Now, the wait() semaphore operation can be defined as:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

and the signal() semaphore operation can be defined as:

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

<i>P₀</i>	<i>P₁</i>
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that P₀ executes wait(S) and then P₁ executes wait(Q). When P₀ executes wait(Q), it must wait until P₁ executes signal(Q). Similarly, when P₁ executes wait(S), it must wait until P₀ executes signal(S). Since these signal() operations cannot be executed, P₀ and P₁ are deadlocked.

We say that 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. Other types of events may result in deadlocks, as we show in Chapter 7. In that chapter, we describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is **indefinite blocking or starvation**, a situation in which processes wait indefinitely within the semaphore.

Classical problems of synchronization

These problems are used for testing nearly every newly proposed synchronization scheme. In our solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

The Bounded-Buffer Problem

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer process is shown in Figure (a), and the code for the consumer process is shown in Figure (b). Note the symmetry between the

```
do {
    .
    .
    /* produce an item in next_produced */
    .
    wait(empty);
    wait(mutex);
    .
    /* add next_produced to the buffer */
    .
    signal(mutex);
    signal(full);
} while (true);
```

```
do {
    .
    wait(full);
    wait(mutex);
    .
    /* remove an item from buffer to next_consumed */
    .
    signal(mutex);
    signal(empty);
    .
    /* consume the item in next_consumed */
    .
} while (true);
```

Figure The structure of the producer process.

Figure The structure of the consumer process.

producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Readers–Writers Problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as **the readers–writers problem**.

In the solution to the readers–writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0. The semaphore rw mutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated. The read count variable keeps track of how many processes are currently reading the object. The semaphore rw mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process and reader process is shown in Figure below:

```
do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);

do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

Figure The structure of a writer process.

Figure The structure of a reader process.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time.

Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.



Figure The situation of the dining philosophers.

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a `wait()` operation on that semaphore. She releases her chopsticks by executing the `signal()` operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher *i* is shown in Figure below.

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    . . .
    /* eat for awhile */
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    /* think for awhile */
    . . .
} while (true);
```

Figure The structure of philosopher *i*.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.

Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in **timing errors** that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences do not always occur.

To deal with such errors, researchers have developed high-level language constructs called “**monitors**”.

The monitor construct ensures that only one process at a time is active within the monitor.

```
monitor monitor name
{
    /* shared variable declarations */

    function P1 ( . . . ) {
        .
    }

    function P2 ( . . . ) {
        .
    }

    .

    function Pn ( . . . ) {
        .
    }

    initialization_code ( . . . ) {
        .
    }
}
```

Figure Syntax of a monitor.

Consequently, the programmer does not need to code this synchronization constraint explicitly. However, the monitor construct, as defined so far, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition construct**. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

condition x, y;

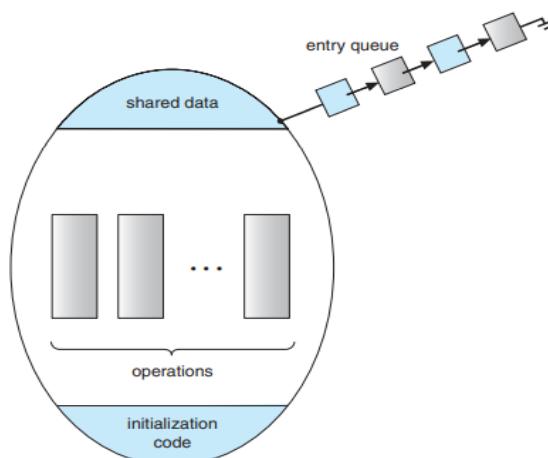


Figure Schematic view of a monitor.

The only operations that can be invoked on a condition variable are **wait()** and **signal()**. The operation **x.wait();**

means that the process invoking this operation is suspended until another process invokes **x.signal();**

The **x.signal()** operation resumes exactly one suspended process. If no process is suspended, then the **signal()** operation has no effect; that is, the state of **x** is the same as if the operation had never been executed.

Note, however, that conceptually both processes can continue with their execution. Two possibilities exist:

1. **Signal and wait.** P either waits until Q leaves the monitor or waits for another condition.
2. **Signal and continue.** Q either waits until P leaves the monitor or waits for another condition.

Dining-Philosophers Solution Using Monitors

Next, we illustrate monitor concepts by presenting a deadlock-free solution to the dining-philosophers problem. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable **state[i] = EATING** only if her two neighbors are not eating: (**state[(i+4) % 5] != EATING**) and (**state[(i+1) % 5] != EATING**).

```

monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
            state[i] = EATING;
            self[i].signal();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}

```

Figure A monitor solution to the dining-philosopher problem.

We also need to declare condition
self[5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor **Dining-Philosophers**, whose definition is shown in Figure above. Each philosopher, before starting to eat, must invoke the operation **pickup()**. This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the **putdown()** operation. Thus, philosopher *i* must invoke the operations **pickup()** and **putdown()** in the following sequence:

```

DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);

```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Principles of 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.”**

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.

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, semaphores, mutex locks, and files).

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.

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 $\{P_0, P_1, \dots, P_n\}$ 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 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n 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.

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: $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.

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

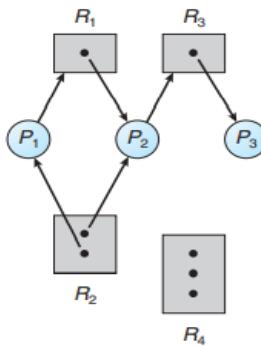
Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j , whereas an assignment edge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

When process P_i requests an instance of resource type R_j , 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 below depicts the following situation.

- The sets P , R , and E :
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- 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 R4
- Process states:
- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.



Given the definition of a
be shown that, if the graph
process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

Figure Resource-allocation graph.

resource-allocation graph, it can
contains no cycles, then no
deadlock may occur.

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 above. Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, we add a request edge P3 → R2 to the graph.

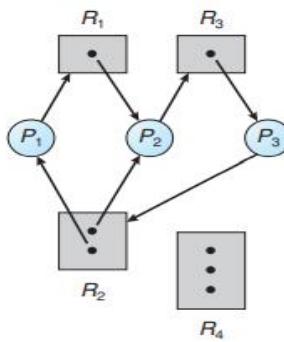


Figure Resource-allocation graph with a deadlock.

At this point, two minimal cycles exist in the system:

- P1 → R1 → P2 → R3 → P3 → R2 → P1
- P2 → R3 → P3 → R2 → P2

Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process P1 or process P2 to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

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

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

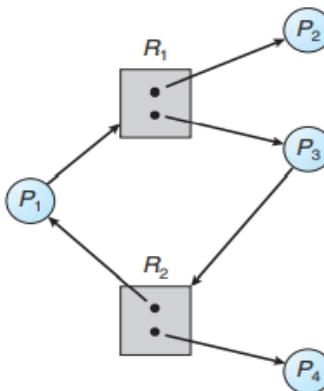


Figure Resource-allocation graph with a cycle but no deadlock.

However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, 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.

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 a **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).

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.

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. For example, a mutex lock cannot be simultaneously shared by several processes.

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.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. 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.

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.

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.

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, R_i . After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$.

Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$. 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.

Deadlock Avoidance

Deadlock-prevention algorithms, 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.

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 is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+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. 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 behavior of the processes controls unsafe states.

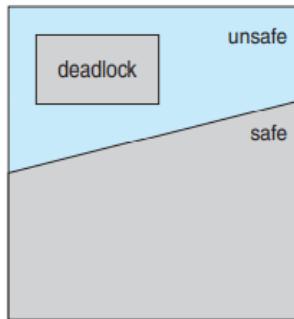


Figure Safe, unsafe, and deadlocked state spaces.

To illustrate, we consider a system with twelve magnetic tape drives and three processes: P0, P1, and P2. Process P0 requires ten tape drives, process P1 may need as many as four tape drives, and process P2 may need up to nine tape drives. Suppose that, at time t0, process P0 is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

	Maximum Needs	Current Needs
P ₀	10	5
P ₁	4	2
P ₂	9	2

At time t0, the system is in a safe state. The sequence satisfies the safety condition. Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P2 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 t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state. At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives. Since process P0 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 P2 may request six additional tape drives and have to wait, resulting in a deadlock. Our mistake was in granting the request from process P2 for one more tape drive. If we had made P2 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.

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 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 $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j 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 P_i requests resource R_j , the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_i , the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

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

Now suppose that process P_1 requests resource R_2 . The request can be granted only if converting the request edge $P_1 \rightarrow R_2$ to an assignment edge $R_2 \rightarrow P_1$ 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 P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource-allocation graph of Figure below

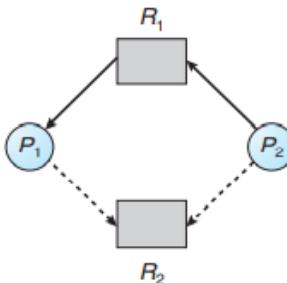


Figure Resource-allocation graph for deadlock avoidance.

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 as shown below

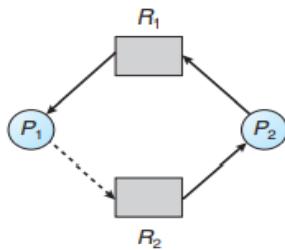


Figure An unsafe state in a resource-allocation graph.

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.

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 $\text{Available}[j]$ equals k , then k instances of resource type R_j are available.
- **Max.** An $n \times m$ matrix defines the maximum demand of each process. If $\text{Max}[i][j]$ equals k , then process P_i may request at most k instances of resource type R_j .
- **Allocation.** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If $\text{Allocation}[i][j]$ equals k , then process P_i is currently allocated k instances of resource type R_j .
- **Need.** An $n \times m$ matrix indicates the remaining resource need of each process. If $\text{Need}[i][j]$ equals k , then process P_i may need k more instances of resource type R_j to complete its task. Note that $\text{Need}[i][j]$ equals $\text{Max}[i][j] - \text{Allocation}[i][j]$.

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 $\text{Work} = \text{Available}$ and $\text{Finish}[i] = \text{false}$ for $i = 0, 1, \dots, n - 1$.
2. Find an index i such that both
 - a. $\text{Finish}[i] == \text{false}$
 - b. $\text{Need}_i \leq \text{Work}$
 If no such i exists, go to step 4.
3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$
 Go to step 2.
4. If $\text{Finish}[i] == \text{true}$ for all i , then the system is in a safe state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

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 P_i . If $\text{Request}_i[j] = k$, then process P_i wants k instances of resource type R_j . When a request for resources is made by process P_i , the following actions are taken:

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\begin{aligned} \text{Available} &= \text{Available} - \text{Request}_i; \\ \text{Allocation}_i &= \text{Allocation}_i + \text{Request}_i; \\ \text{Need}_i &= \text{Need}_i - \text{Request}_i; \end{aligned}$$

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

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:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix Need is defined to be $\text{Max} - \text{Allocation}$ and is as follows:

	<u>Need</u>
	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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 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 $Request_1 \leq Available$ —that is, that $(1,0,2) \leq (3,3,2)$, which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
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 satisfies the safety requirement. Hence, we can immediately grant the request of process P_1 .

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.

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

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 P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q . In the following Figure we present a resource-allocation graph and the corresponding wait-for graph.

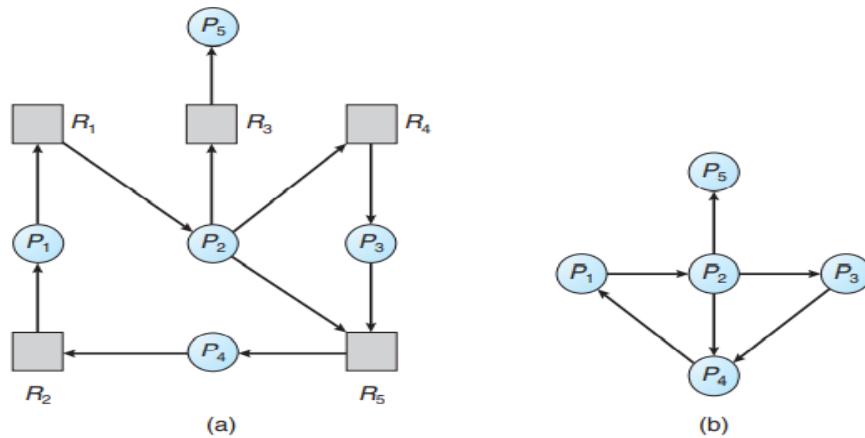


Figure (a) Resource-allocation graph. (b) Corresponding wait-for graph.

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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request.** An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i][j]$ equals k , then process P_i is requesting k more instances of resource type R_j

Algorithm:

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively. Initialize $Work = Available$. For $i = 0, 1, \dots, n-1$, if $\text{Allocation}_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.
2. Find an index i such that both
 - $Finish[i] == false$
 - $\text{Request}_i \leq Work$
 If no such i exists, go to step 4.
3. $Work = Work + \text{Allocation}_i$
 $Finish[i] = true$
 Go to step 2.
4. If $Finish[i] == false$ for some $i, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

To illustrate this algorithm, we consider a system with five processes P0 through P4 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 T0, we have the following resource-allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	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 results in Finish[i] == true for all i.

Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

	<u>Request</u>
	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 P0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

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.

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

Resource Preemption

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. Selecting a victim. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. Rollback. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. Starvation. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation any practical system must address. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Memory Management: Swapping, Contiguous Memory Allocation, Paging, structure of the Page Table, Segmentation.

Virtual Memory Management: Virtual Memory, Demand Paging, Page-Replacement Algorithms, Thrashing.

Memory Management

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.

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. 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.

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.

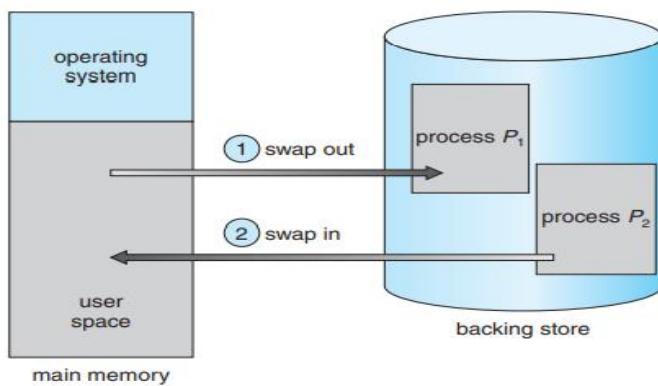


Figure Swapping of two processes using a disk as a backing store.

Standard swapping is not used in modern operating systems. It requires too much swapping time and provides too little execution time to be a reasonable solution.

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.

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

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. 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.

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.

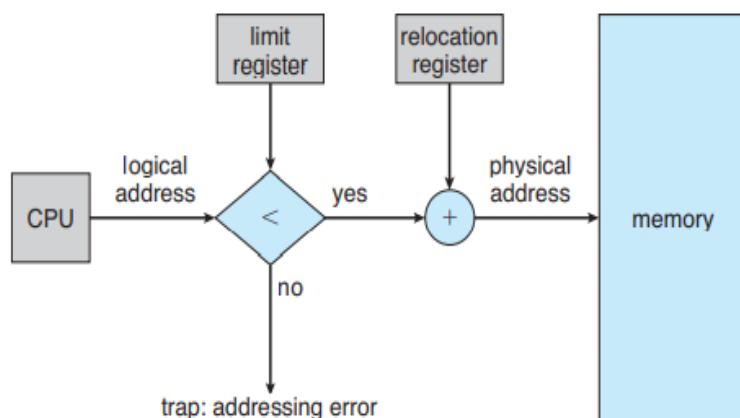


Figure Hardware support for relocation and limit registers.

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.

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. 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.

In general, 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

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.

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. Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous, thus allowing a process to be allocated physical memory wherever such memory is available.

Segmentation

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.

Basic method

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

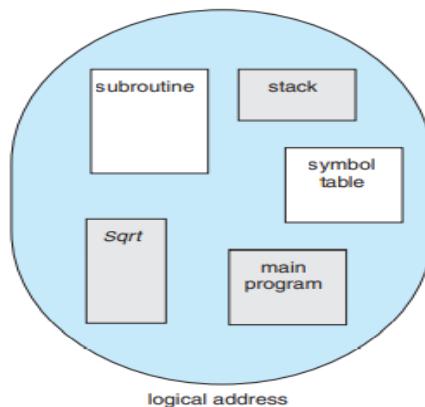


Figure Programmer's view of a program.

Segmentation Hardware

We must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses. This mapping is affected 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.

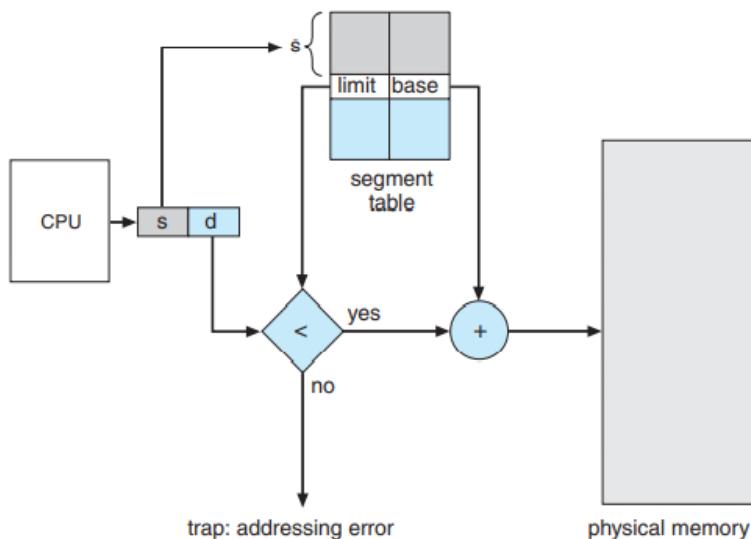
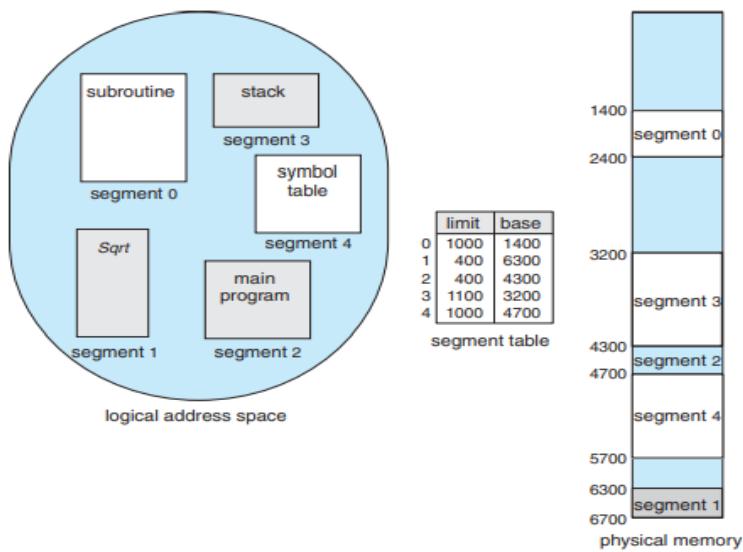


Figure Segmentation hardware.

The use of a segment table is illustrated in Figure above. 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.

Paging

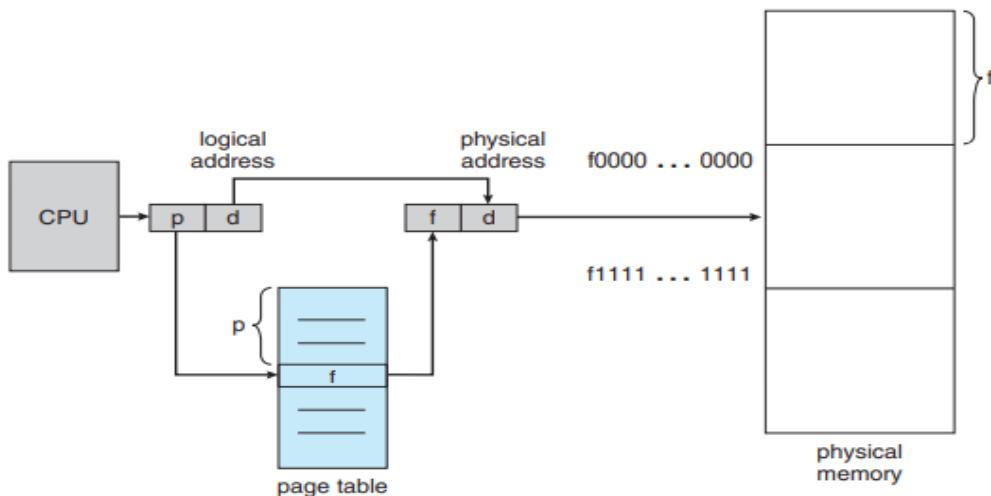
Segmentation permits the physical address space of a process to be non-contiguous. Paging is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction, whereas segmentation does not.

**Figure** Example of segmentation.

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.

The hardware support for paging is illustrated in Figure below. Every address generated by the CPU is divided into two parts: **a page number (p)** and **a page offset (d)**. The page number is used 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.

**Figure** Paging hardware.

The paging model of memory is shown in Figure below.

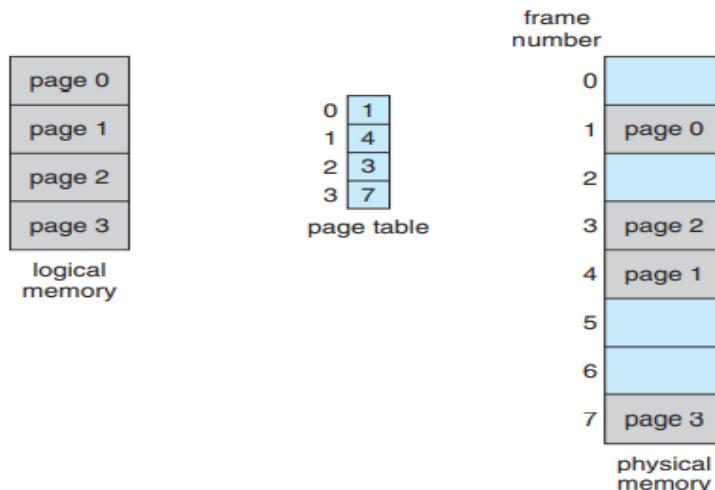


Figure Paging model of logical and physical memory.

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.

Hardware Support

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 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 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 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. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

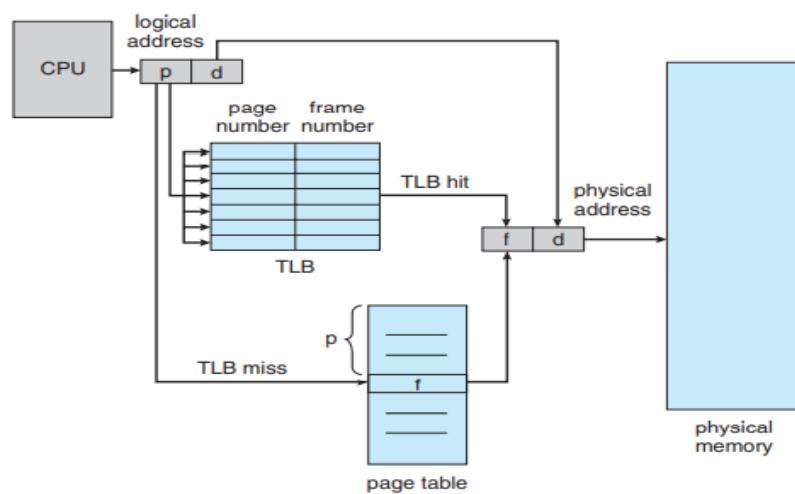


Figure Paging hardware with TLB.

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 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.

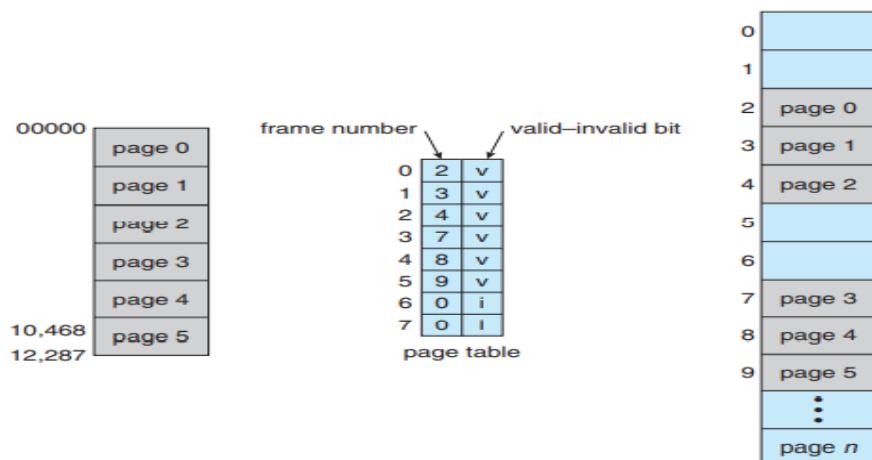


Figure Valid (v) or invalid (i) bit in a page table.

Structure of the Page Table

Hierarchical Paging

Most modern computer systems support a large logical address space. In such an environment, the page table itself becomes excessively large. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways.

One way is to use a **two-level paging algorithm**, in which the page table itself is also paged.

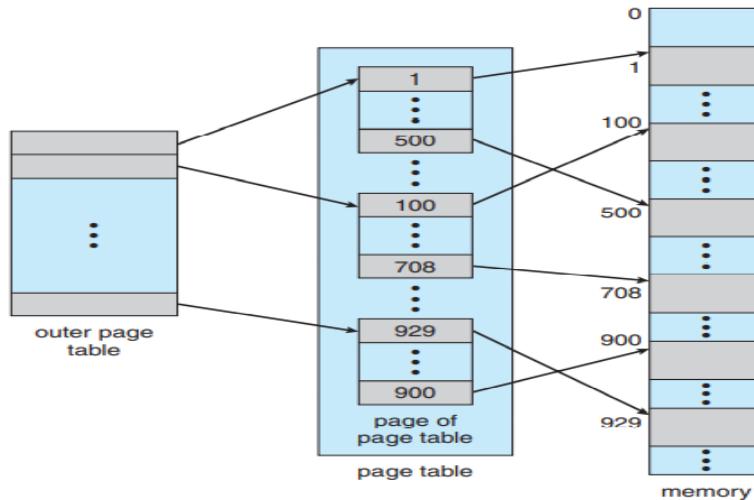
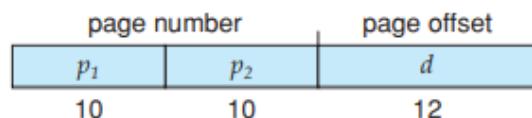


Figure A two-level page-table scheme.

For example, consider again the system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where p_1 is an index into the outer page table and p_2 is the displacement within the page of the inner page table.

The address-translation method for this architecture is shown in Figure below. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

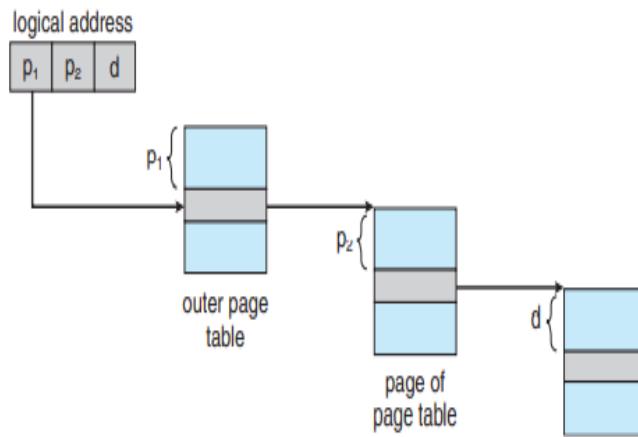


Figure Address translation for a two-level 32-bit paging architecture.

Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure below:

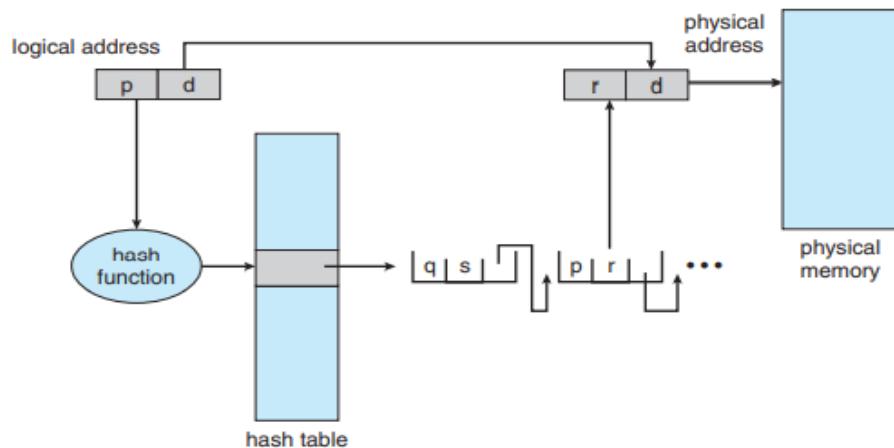


Figure Hashed page table.

Inverted Page Tables

Usually, each process has an associated page table. The page table has one entry for each page that the process is using (or one slot for each virtual address, regardless of the latter's validity). This table representation is a natural one, since processes reference pages through the pages' virtual addresses. The operating system must then translate this reference into a physical memory address. Since the table is sorted by virtual address, the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly. One of the drawbacks of this method is that each page table may consist of millions of entries.

These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. The following figure shows the operation of an inverted page table.

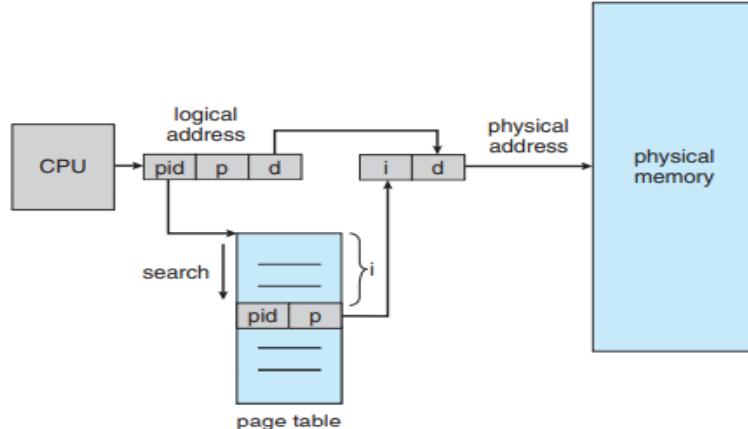


Figure Inverted page table.

Virtual Memory Management

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.

Virtual Memory

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. 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.

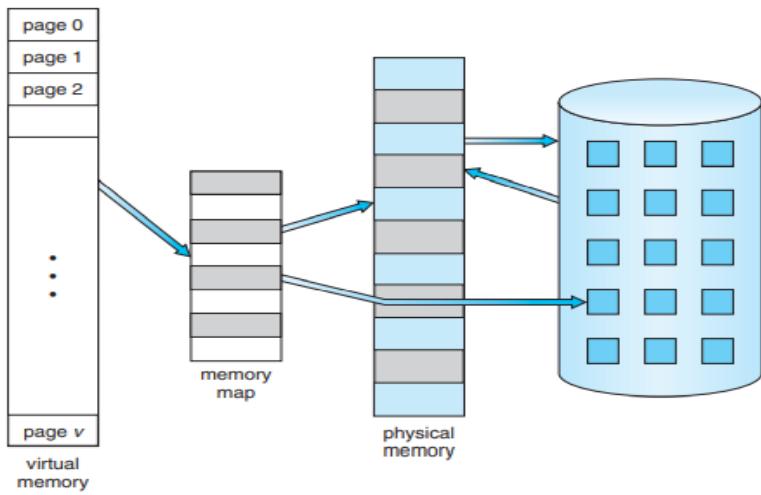


Figure Diagram showing virtual memory that is larger than physical memory.

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 below.

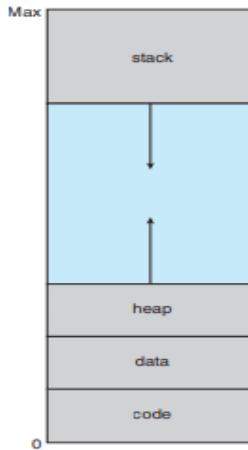


Figure Virtual address space.

We allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

In addition to separating logical memory from physical memory, virtual memory allows files and memory to be shared by two or more processes through page sharing . This leads to the following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes.
- Similarly, processes can share memory. Two or more processes can communicate through the use of shared memory. Virtual memory allows one process to create a region of memory that it can share with another process.
- Pages can be shared during process creation with the fork() system call, thus speeding up process creation.

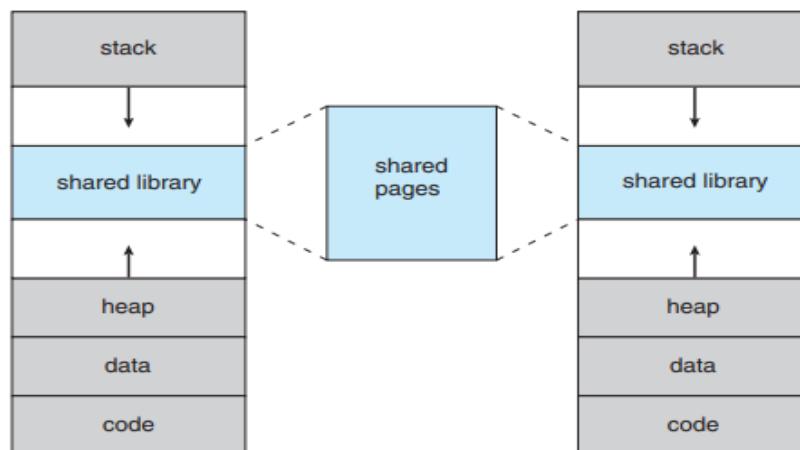


Figure Shared library using virtual memory.

Demand Paging

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 9.4) 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.

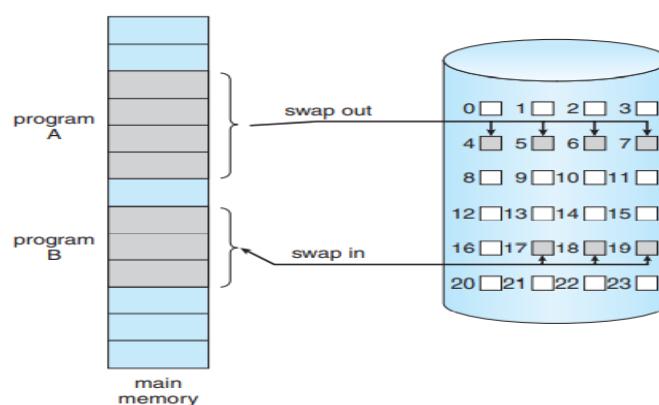


Figure Transfer of a paged memory to contiguous disk space.

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 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 below.

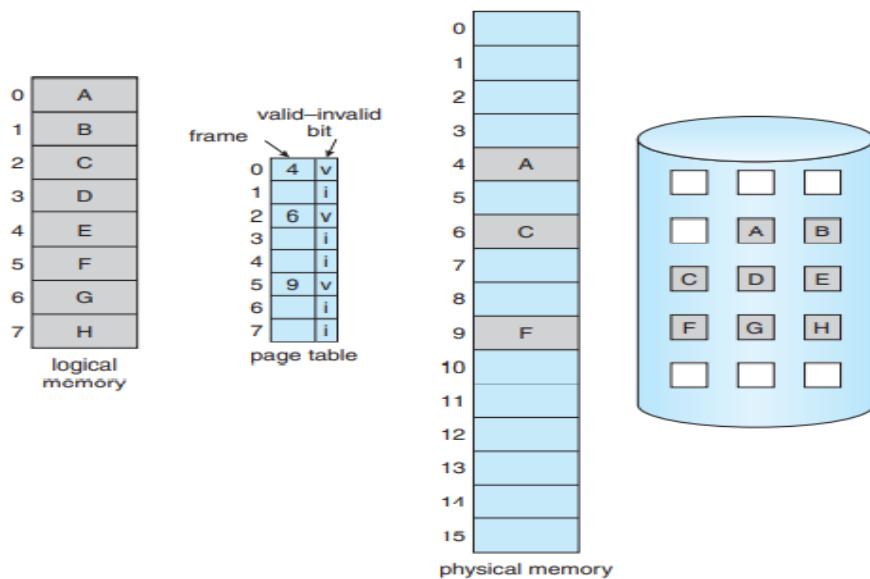


Figure Page table when some pages are not in main memory.

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 straight forward:

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.

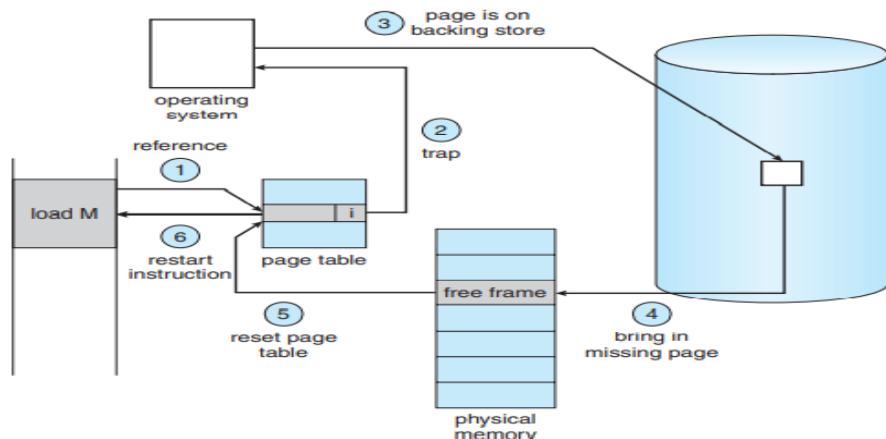


Figure Steps in handling a page fault.

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. 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.

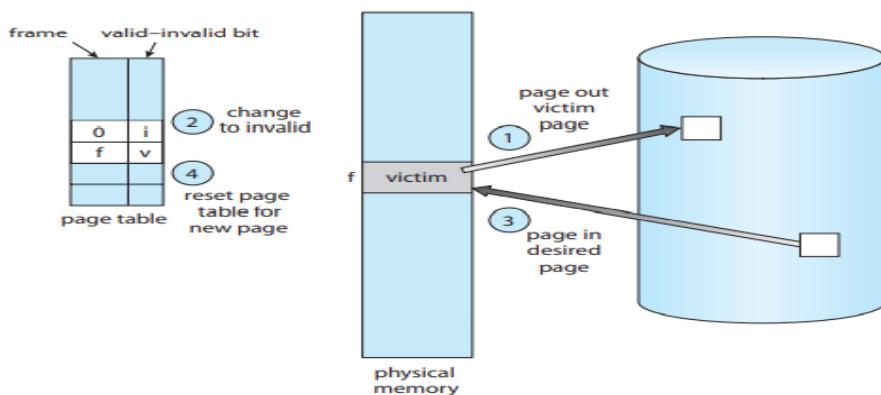


Figure Page replacement.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. 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.

We next illustrate several page-replacement algorithms. In doing so, we use the reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.

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 below. Every time a fault occurs, we show which pages are in our three frames. There are **fifteen faults** altogether.

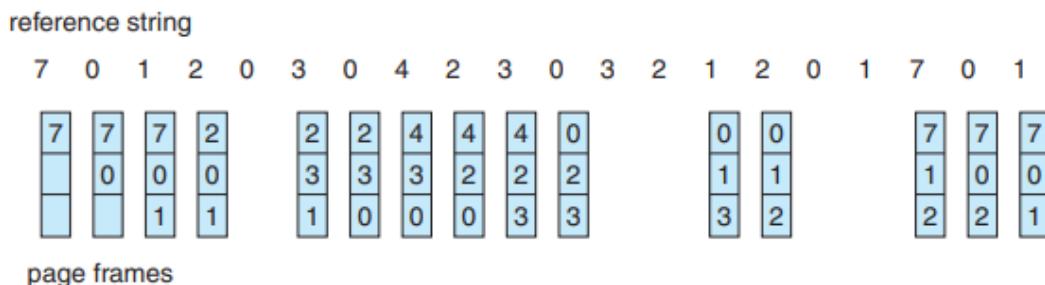


Figure . FIFO page-replacement algorithm.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Figure below shows the curve of page faults for this reference string versus the number of available frames. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly**: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

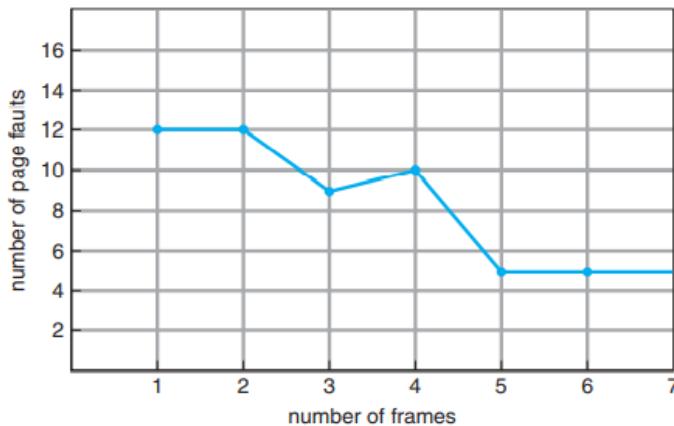


Figure Page-fault curve for FIFO replacement on a reference string.

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm—the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN.

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 below. 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. In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

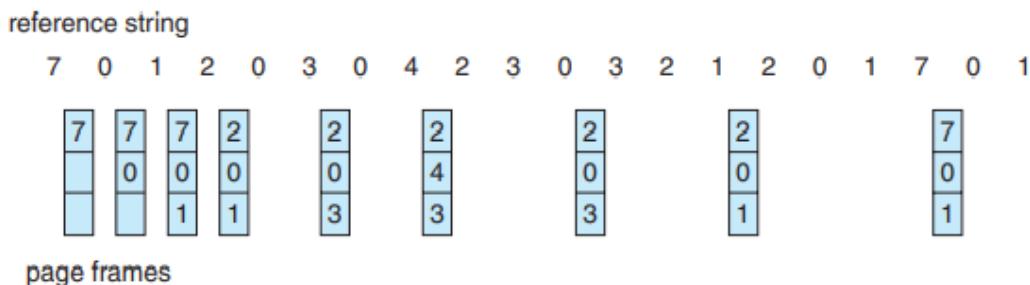


Figure Optimal page-replacement algorithm.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

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.

The result of applying LRU replacement to our example reference string is shown in Figure below. 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.

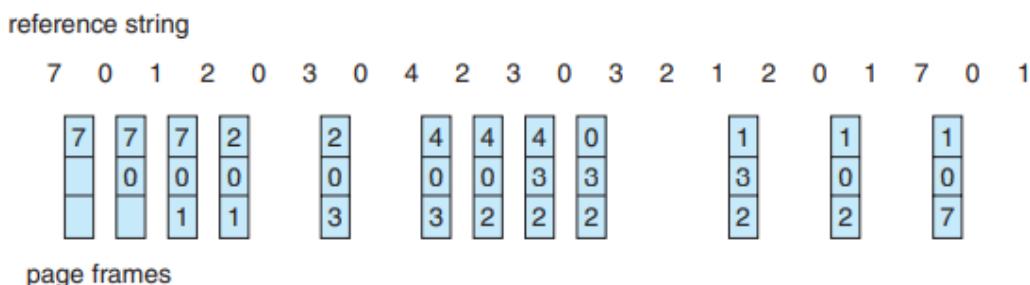


Figure LRU page-replacement algorithm.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly.

Thrashing

In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This **high paging activity is called thrashing**. A process is thrashing if it is spending more time **paging** than **executing**.

Thrashing results in severe performance problems. Consider the following scenario, which is based on the actual behavior of early paging systems. The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. The page fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in Figure below, in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multi-programming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

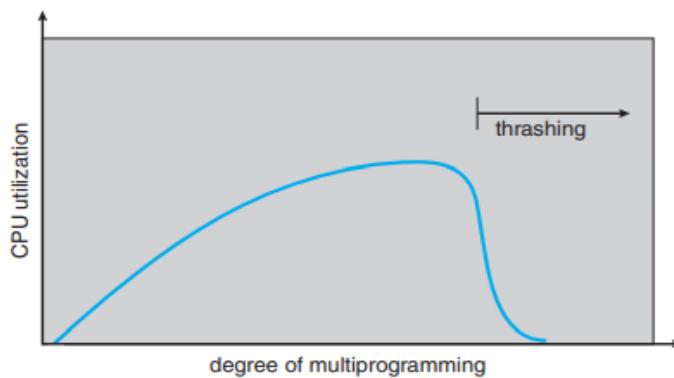


Figure Thrashing.

File system Interface- The concept of a file, Access Methods, Directory structure, File system mounting.

File System implementation- File system structure, allocation methods, free-space management, overview of Mass-storage structure, Disk scheduling.

File System Interface

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: *a collection of files*, each storing related data, and a *directory structure*, which organizes and provides information about all the files in the system.

File Concept

A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of functions, each of which is further organized as declarations followed by executable statements. An executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file's attributes vary from one operating system to another but typically consist of these:

- **Name.** The symbolic file name is the only information kept in human readable form.
- **Identifier.** This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- **Type.** This information is needed for systems that support different types of files.
- **Location.** This information is a pointer to a device and to the location of the file on that device.
- **Size.** The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
- **Protection.** Access-control information determines who can do reading, writing, executing, and so on.
- **Time, date, and user identification.** This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

(Diagram)

Some newer file systems also support extended file attributes, including character encoding of the file and security features such as a file checksum. The above Figure illustrates a file info window on Mac

OS X, which displays a file's attributes. The information about all files is kept in the directory structure, which also resides on secondary storage. Typically, a directory entry consists of the file's name and its unique identifier. The identifier in turn locates the other file attributes.

File Operations

To define a file properly, we need to consider the operations that can be performed on files. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. Let's examine what the operating system must do to perform each of these six basic file operations:

- **Creating a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.

- **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

- **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Both the read and write operations use this same pointer, saving space and reducing system complexity.

- **Repositioning within a file.** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a *file seek*.

- **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

- **Truncating a file.** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

These six basic operations comprise the minimal set of required file operations. Other common operations include appending new information to the end of an existing file and renaming an existing file. These primitive operations can then be combined to perform other file operations.

File Types

When we design a file system—indeed, an entire operating system—we always consider whether the operating system should recognize and support file types. If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period. In this way, the user and the operating system can tell from the name alone what the type of a file is. Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters. Examples include resume.docx, server.c, and ReaderThread.cpp.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. e. Only a file with a .com, .exe, or .sh extension can be executed, for instance. The .com and .exe files are two forms of binary executable files, whereas the .sh file is a shell script containing, in ASCII format, commands to the operating system.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure: Common File types

Access Methods

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files, while others support many access methods, and choosing the right one for a particular application is a major design problem.

Sequential Access

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

Reads and writes make up the bulk of the operations on a file. A read operation—`read_next()`—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, the write operation—`write_next()`—appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n . Sequential access, which is depicted in the following Figure, is based on a tape model of a file.

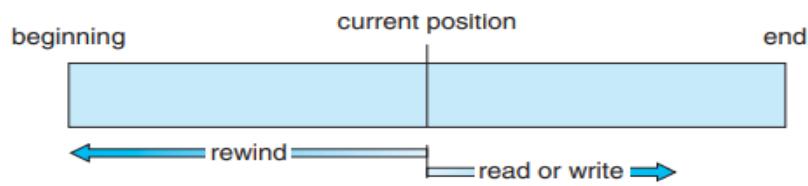


Figure Sequential-access file.

Direct Access

Another method is *direct access* (or *relative access*). Here, a file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have **read(n)**, where n is the block number, rather than **read_next()**, and **write(n)** rather than **write_next()**. An alternative approach is to retain **read_next()** and **write_next()**, as with sequential access, and to add an operation **position_file(n)** where n is the block number. Then, to effect a **read(n)**, we would **position_file(n)** and then **read_next()**.

Other Access Methods

Other access methods can be built on top of a direct-access method. These methods generally involve the construction of an index for the file. The index, like an index in the back of a book, contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

For example, IBM's indexed sequential-access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, we first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block is searched sequentially. In this way, any record can be located from its key by at most two direct-access reads. The Figure below shows a similar situation as implemented by VMS index and relative files.

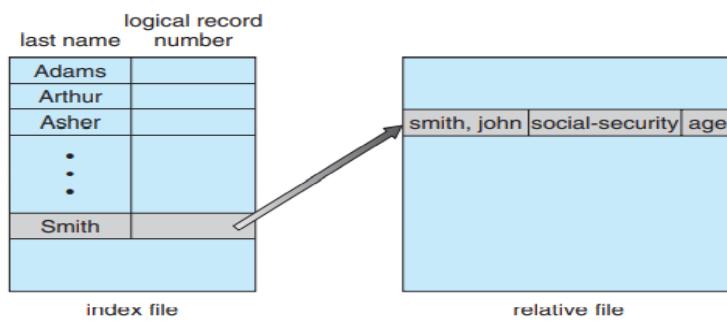


Figure Example of index and relative files.

Directory Structure

The directory can be viewed as a symbol table that translates file names into their directory entries. If we take such a view, we see that the directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:

- **Search for a file.** We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names, and similar names may indicate a relationship among files, we may want to be able to find all files whose names match a particular pattern.

- **Create a file.** New files need to be created and added to the directory.
- **Delete a file.** When a file is no longer needed, we want to be able to remove it from the directory.
- **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.

• **Rename a file.** Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.

• **Traverse the file system.** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

The most common schemes for defining the logical structure of a directory are:

Single-Level Directory

The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand. A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

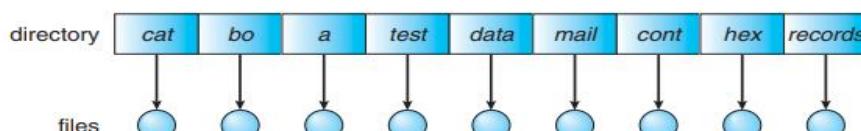


Figure Single-level directory.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names among different users. The standard solution is to create a separate directory for each user.

In the two-level directory structure, each user has his own user file directory (**UFD**). The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system's master file directory (**MFD**) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.

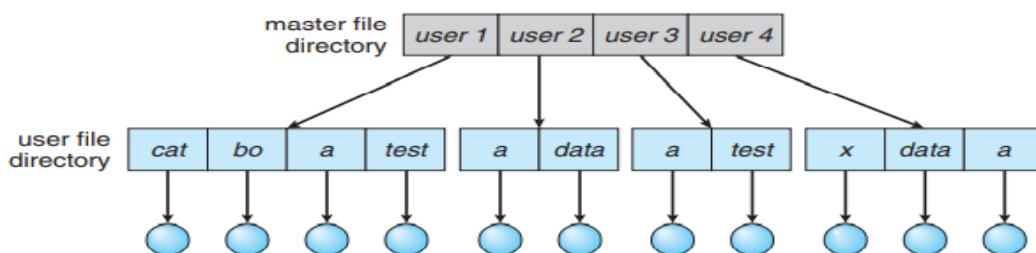


Figure Two-level directory structure.

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.

Tree-Structured Directories

A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.

A directory (or subdirectory) contains a set of files or subdirectories. A directory is simply another file, but it is treated in a special way. All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories. In normal use, each process has a current directory. The current directory should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file. To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory. Thus, the user can change her current directory whenever she wants.

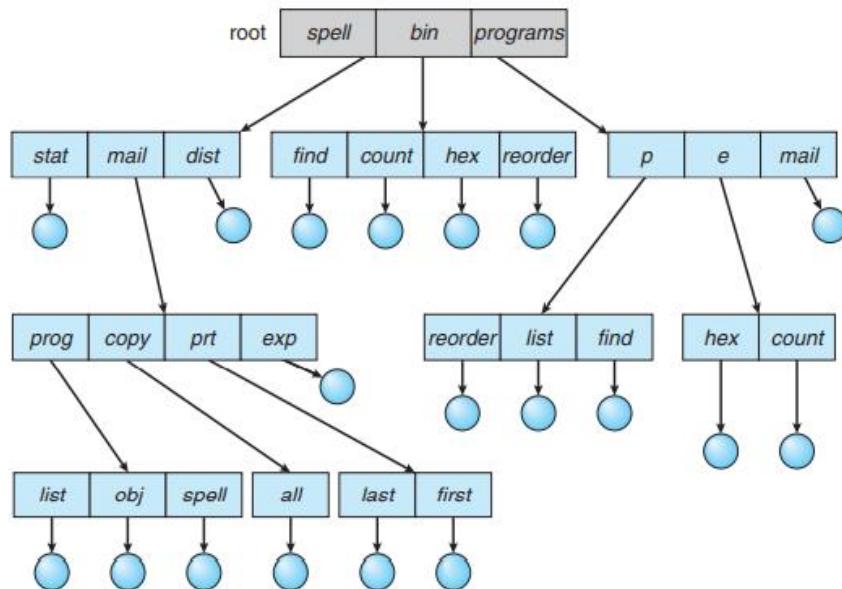


Figure Tree-structured directory structure.

Path names can be of two types: absolute and relative. An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path. A relative path name defines a path from the current directory. For example, in the tree-structured file system of Figure 11.11, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users.

Acyclic-Graph Directories

A tree structure prohibits the sharing of files or directories. An acyclic graph—that is, a graph with no cycles—allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

It is important to note that a shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories.

When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.

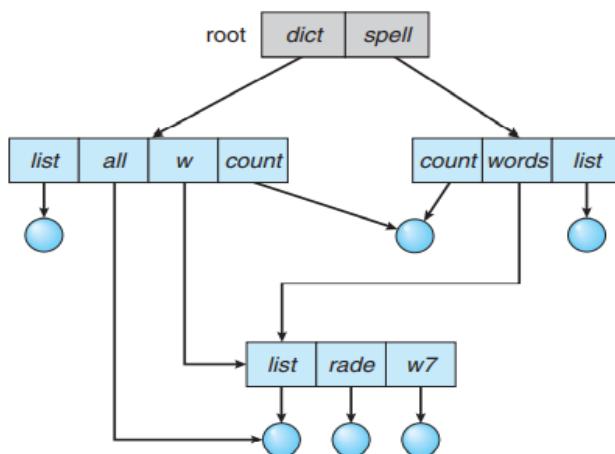


Figure Acyclic-graph directory structure.

File System mounting

Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. More specifically, the directory structure may be built out of multiple volumes, which must be mounted to make them available within the file-system name space.

The mount procedure is straightforward. The operating system is given the name of the device and the mount point—the location within the file structure where the file system is to be attached. Some operating systems require that a file system type be provided, while others inspect the structures of the device and determine the type of file system. Typically, a mount point is an empty directory. For instance, on a UNIX system, a file system containing a user's home directories might be mounted as /home; then, to access the directory structure within that file system, we could precede the directory names with /home, as in /home/jane. Mounting that file system under /users would result in the path name /users/jane, which we could use to reach the same directory.

Next, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems, and even file systems of varying types, as appropriate.

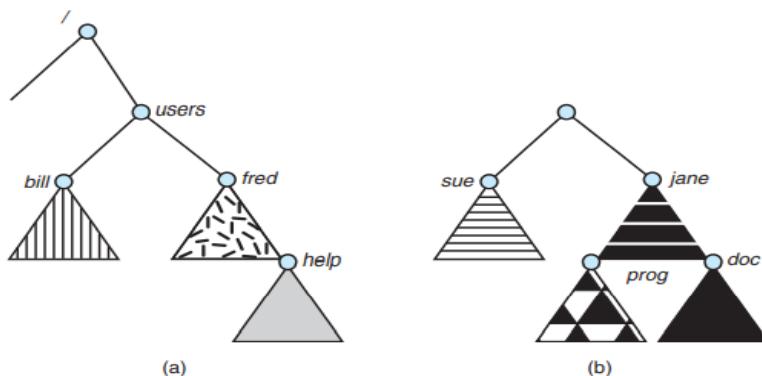


Figure File system. (a) Existing system. (b) Unmounted volume.

To illustrate file mounting, consider the file system depicted in the Figure above, where the triangles represent subtrees of directories that are of interest. Figure (a) shows an existing file system, while Figure (b) shows an unmounted volume residing on /device/dsk. At this point, only the files on the existing file system can be accessed.

File System Implementation

The file system provides the mechanism for on-line storage and access to file contents, including data and programs. The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently.

File System Structure

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.

2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in the Figure below is an example of a **layered design**. Each level in the design uses the features of lower levels to create new features for use by higher levels.

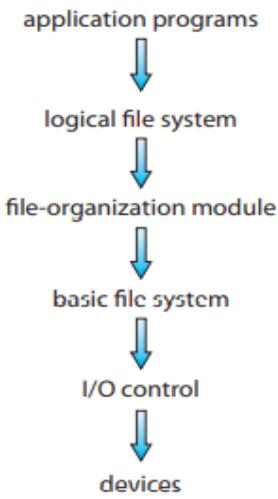


Figure . Layered file system.

- The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- The **file-organization module** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. It maintains file structure via file-control blocks.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules. Unfortunately, layering can introduce more operating-system overhead, which may result in decreased performance. The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

Allocation Methods

The direct-access nature of disks gives us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: **contiguous, linked, and indexed**. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

Contiguous Allocation

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement.

Accessing a file that has been allocated contiguously is easy. For **sequential access**, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For **direct access** to block i of a file that starts at block b , we can immediately access block $b + i$. Thus, both sequential and direct access can be supported by contiguous allocation.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. Another problem with contiguous allocation is determining how much space is needed for a file.

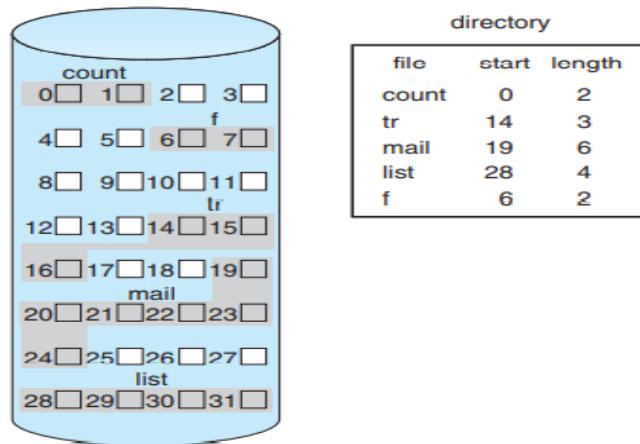


Figure Contiguous allocation of disk space.

Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Figure). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

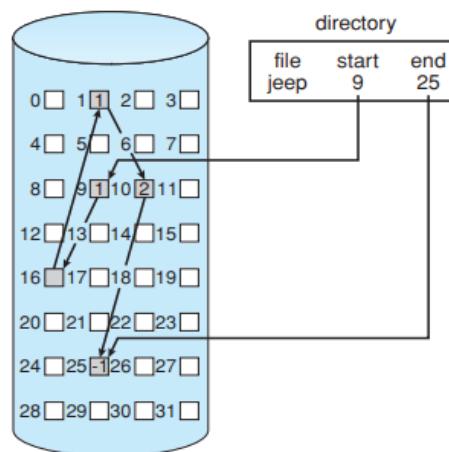


Figure Linked allocation of disk space.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when the file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. Another disadvantage is the space required for the pointers. Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged. A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file.

An important variation on linked allocation is the use of a file-allocation table (**FAT**). This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry. An unused block is indicated by a table value of 0. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in the following Figure for a file consisting of disk blocks 217, 618, and 339.

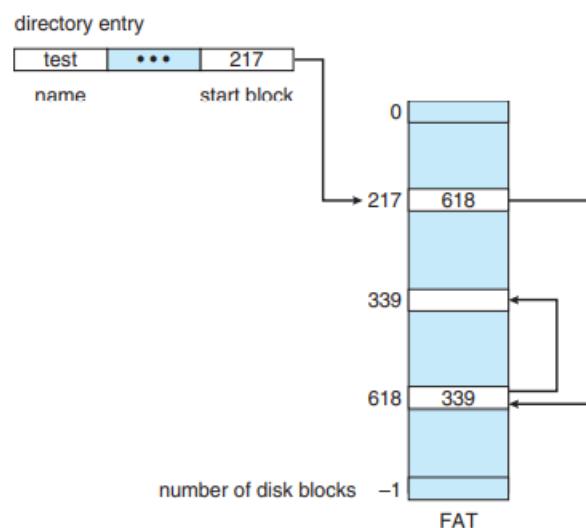


Figure File-allocation table.

Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-block addresses. The i^{th} entry in the index block points to the i^{th} block of the file. The directory contains the address of the index block. To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry.

When the file is created, all pointers in the index block are set to null. When the i^{th} block is first written, a block is obtained from the free-space manager, and its address is put in the i^{th} index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space. Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

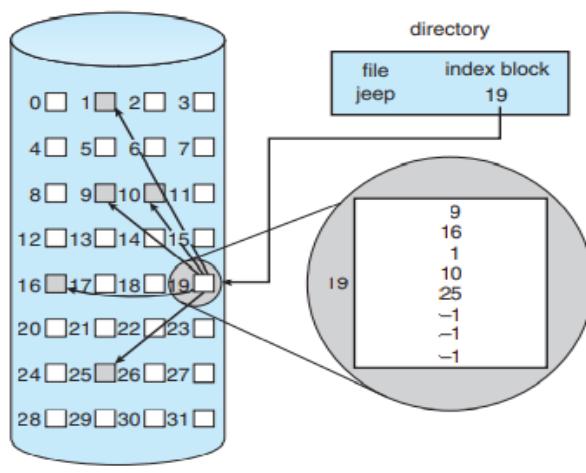


Figure Indexed allocation of disk space.

Free Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

Bit Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000 ...

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.

Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. For example, the blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, we would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on (Figure 12.10). This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time. Fortunately, however, traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed.

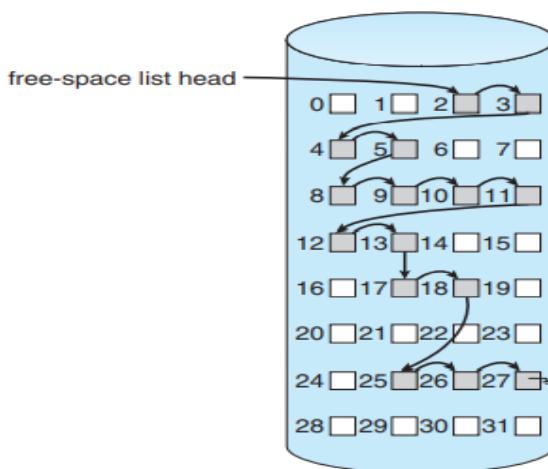


Figure Linked free-space list on disk.

Grouping

A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first $n-1$ of these blocks are actually free. The last block contains the addresses of another n free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, unlike the situation when the standard linked-list approach is used.

Counting

Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.

Overview of Mass Storage Structure

Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple. 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.

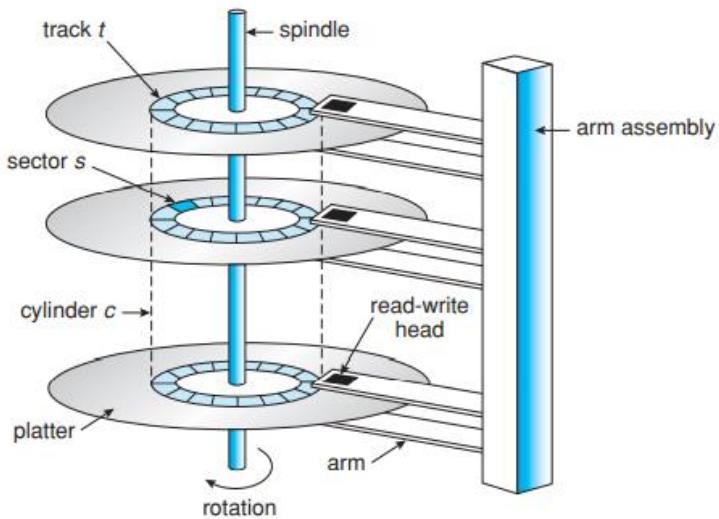


Figure . . . Moving-head disk mechanism.

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.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute (RPM). Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**. Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

Solid-State Disks

Sometimes old technologies are used in new ways as economics change or the technologies evolve. An example is the growing importance of **solid-state disks**, or **SSDs**. Simply described, an SSD is nonvolatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips.

SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power. However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited. One use for SSDs is in storage arrays, where they hold file-system metadata that require high performance. SSDs are also used in some laptop computers to make them smaller, faster, and more energy-efficient.

Because SSDs can be much faster than magnetic disk drives, standard bus interfaces can cause a major limit on throughput. Some SSDs are designed to connect directly to the system bus (PCI, for example). SSDs are changing other traditional aspects of computer design as well. Some systems use them as a direct replacement for disk drives, while others use them as a new cache tier, moving data between magnetic disks, SSDs, and memory to optimize performance.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-5 and SDLT.

Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. 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.

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, in that order.

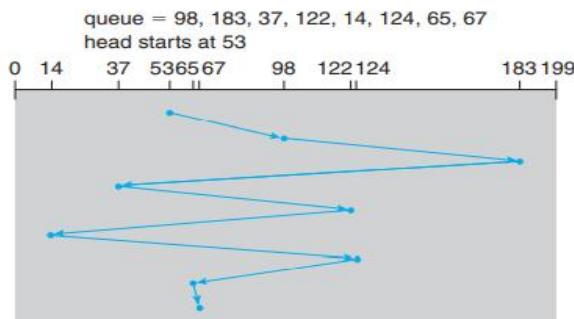


Figure FCFS disk scheduling.

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 the above Figure. 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.

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 . 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.

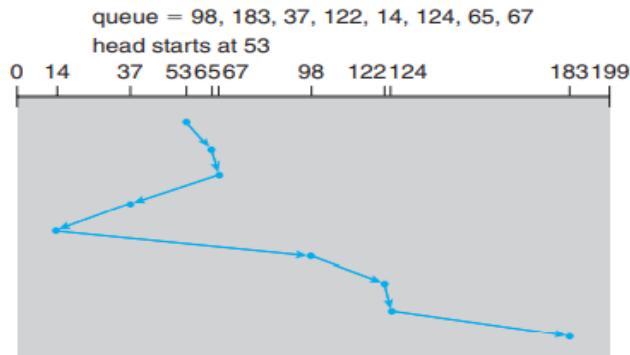


Figure SSTF disk scheduling.

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.

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. 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.

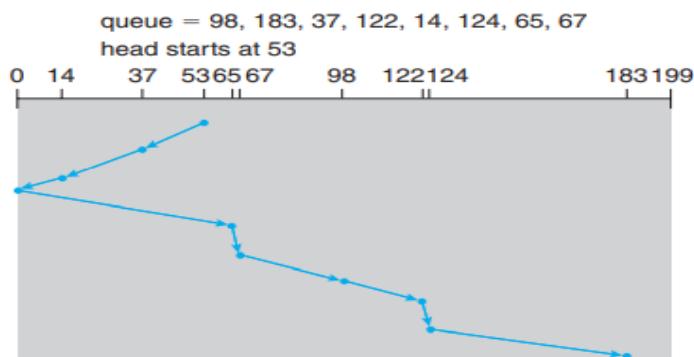


Figure SCAN disk scheduling.

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.

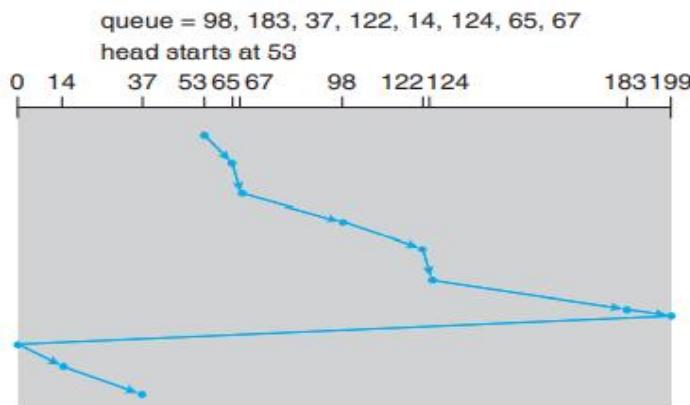


Figure C-SCAN disk scheduling.

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.

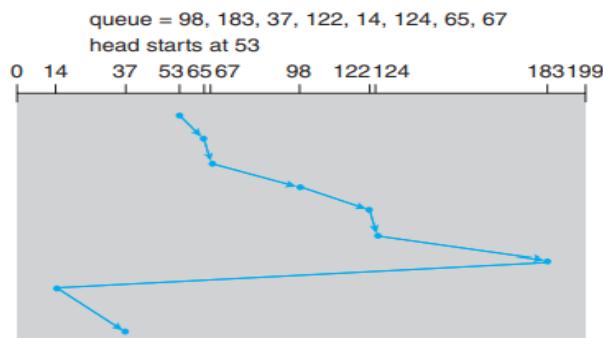


Figure C-LOOK disk scheduling.