UNIT-I

Operating Systems Overview: Operating System Functions, Operating System Structure, Operating System Operations, Protection and Security, Kernel Data Structures, Computing Environments, Open-Source Operating Systems.

Operating Systems Functions

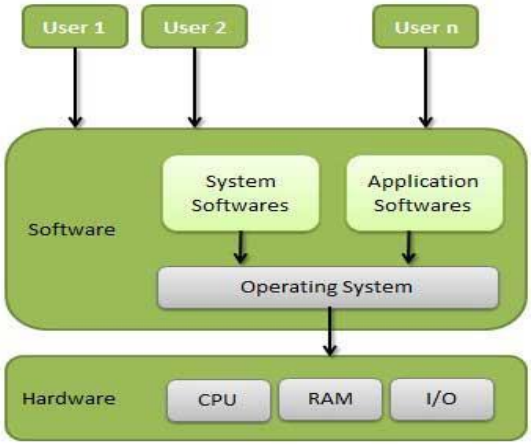
An operating System (OS) is an intermediary between users and computer hardware. It provides users an environment in which a user can execute programs conveniently and efficiently.

(or)

In technical terms, It is a software which manages hardware. An operating System controls the allocation of resources and services such as memory, processors, devices and information.

(or)

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be access directly by the CPU. So for a program to be executed, it must in the main memory. Operating System does the following activities for memory management.

- Keeps tracks of primary memory i.e. what part of it are in use by whom, what part are not in use.
- In multiprogramming, OS decides which process will get memory when and how much.
- Allocates the memory when the process requests it to do so.
- De-allocates the memory when the process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, OS decides which process gets the processor when and how much time. This function is called process scheduling. Operating System does the following activities for processor management.

- Keeps tracks of processor and status of process. Program responsible for this task is known as traffic controller.
- Allocates the processor(CPU) to a process.
- De-allocates processor when processor is no longer required.

Device Management

OS manages device communication via their respective drivers. Operating System does the following activities for device management.

- Keeps tracks of all devices. Program responsible for this task is known as the I/O controller.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Operating System does the following activities for file management.

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities

Following are some of the important activities that Operating System does.

Security -- By means of password and similar other techniques, preventing unauthorized access to programs and data.
Control over system performance -- Recording delays between request for a service and response from the system.
Job accounting -- Keeping track of time and resources used by various jobs and users.
Error detecting aids -- Production of dumps, traces, error messages and other debugging and error detecting aids.
Coordination between other software's and users -- Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Operating Systems Objectives/Goals

- **Convenience**

The primary goal of an operating system is convenience for the user. If an application program is a set of machine instructions then it is completely responsible for controlling the computer hardware. It is a complicated task. To simplify this task, a set of system programs are provided, called utilities and they implement frequently used functions which assist in program creation, management of files and control of Input/Output devices.

- **Efficiency**

The secondary goal of an operating system is efficient operation of the system. Operating system is responsible for managing the resources. That is the movement, storage and processing of data.A portion of operating system is in main memory. This includes the Kernel or nucleus, which contains the most frequently used functions in the operating system. The remainder of main memory contains other user programs and data. Operating system determine how much processor time is to be devoted to the execution of a program. That is the efficient utilization of the resources.

- **Ability to Evolve**

Operating system should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions.

Operating system will evolve over time for a number of reasons:

- Hardware upgrades plus new types of hardware. For example, view several applications at the same time through windows.
- New services, that is new measurement and control tools may be added.
- Fixes, that is faults will be discovered and fixes.

EXTRA-Computer Operating Systems

Operating systems are there from the very first computer generation. Operating systems keep evolving over the period of time. Following are few of the important types of operating system which are most commonly used.

Serial Processing

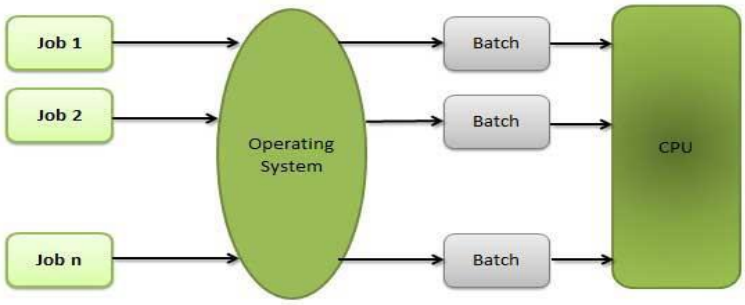
Users access the computer in series. From the late 1940's to mid 1950's, the programmer interacted directly with computer hardware i.e., no operating system. These machines were run with a console consisting of display lights, toggle switches, some form of input device and a printer. Programs in machine code are loaded with the input device like card reader. If an error occur the program was halted and the error condition was indicated by lights. Programmers examine the registers and main memory to determine error. If the program is success, then output will appear on the printer.

Main problem here is the setup time. That is single program needs to load source program into memory, saving the compiled (object) program and then loading and linking together.

Batch processing

Batch processing is a technique in which Operating System collects one programs and data together in a batch before processing starts. Operating system does the following activities related to batch processing.

- OS defines a job which has predefined sequence of commands, programs and data as a single unit.
- OS keeps a number a jobs in memory and executes them without any manual information.
- Jobs are processed in the order of submission i.e first come first served fashion.
- When job completes its execution, its memory is released and the output for the job gets copied into an output spool for later printing or processing.



ADVANTAGES

- Batch processing takes much of the work of the operator to the computer.
- Increased performance as a new job get started as soon as the previous job finished without any manual intervention.

DISADVANTAGES

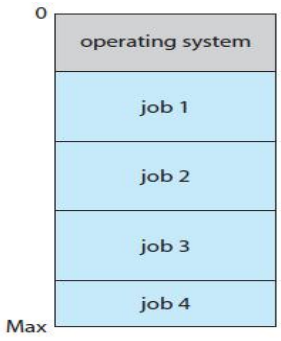
- Difficult to debug program.
- A job could enter an infinite loop.
- Due to lack of protection scheme, one batch job can affect pending jobs.

Operating-System Structure

Multiprogramming

When two or more programs are residing in memory at the same time, then sharing the processor is referred to the multiprogramming. Multiprogramming assumes a single shared processor. Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.

Following figure shows the memory layout for a multiprogramming system.



Operating system does the following activities related to multiprogramming.

- The operating system keeps several jobs in memory at a time.
- This set of jobs is a subset of the jobs kept in the job pool.

- The operating system picks and begins to execute one of the job in the memory.
- Multiprogramming operating system monitors the state of all active programs and system resources using memory management programs to ensures that the CPU is never idle unless there are no jobs

ADVANTAGES

- High and efficient CPU utilization.
- User feels that many programs are allotted CPU almost simultaneously.

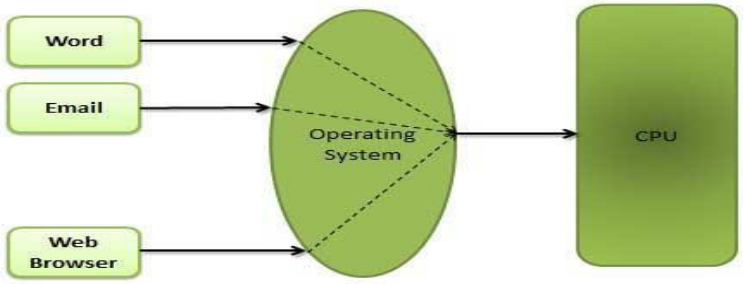
DISADVANTAGES

- CPU scheduling is required.
- To accommodate many jobs in memory, memory management is required.

Multitasking

Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently.



of Timesharing operating systems are following

- Provide advantage of quick response.
- Avoids duplication of software.
- Reduces CPU idle time.

Disadvantages of Timesharing operating systems are following.

- Problem of reliability.
- Question of security and integrity of user programs and data.
- Problem of data communication.

A program loaded into memory and executing is called a **process**.

If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**, When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management. In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling.**

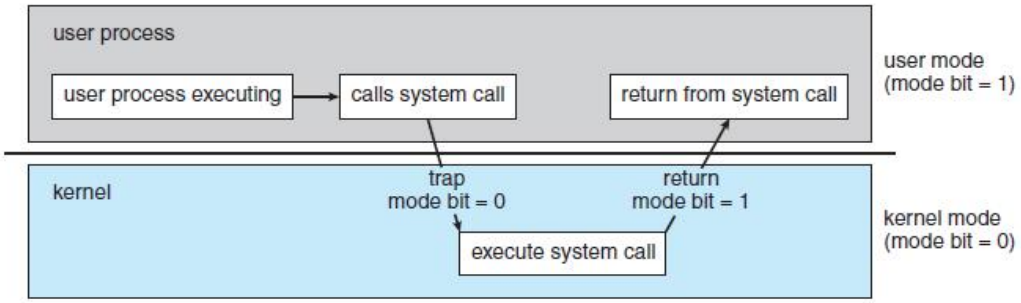
In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk. Amore common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**. Further, it abstracts main memory into a large, uniform array of storage, separating **logical memory** as viewed by the user from physical memory. This arrangement frees programmers from concern over memory-storage limitations.

Operating-System Operations

Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt.

Dual-Mode and Multimode Operation

We need two separate ***modes*** of operation: **user mode** and **kernel mode** (also called **supervisor mode**, **system mode**, or **privileged mode**). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfil the request.



Transition from user to kernel mode.

At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode.

The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

The concept of modes can be extended beyond two modes (in which case the CPU uses more than one bit to set and test the mode). CPUs that support virtualization frequently have a separate mode to indicate when the **virtual machine manager** (**VMM**)—and the virtualization management software—is in control of the system. In this mode, the VMM has more privileges than user processes but fewer than the kernel.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system. For instance, MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode. A user program running awry can wipe out the operating system by writing over it with data; and multiple programs are able to write to a device at the same time, with potentially disastrous results. Modern versions of the Intel CPU do provide dual-mode operation.

Timer

We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a **timer**. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A **variable timer** is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.

We can use the timer to prevent a user program from running too long. A simple technique is to initialize a counter with the amount of time that a program is allowed to run. A program with a 7-minute time limit, for example, would have its counter initialized to 420. Every second, the timer interrupts, and the counter is decremented by 1.

Protection and Security

- Protection – any mechanism for controlling access of processes or users to resources defined by the OS
- Security – defense of the system against internal and external attacks
    - Huge range, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
    - User identities (user IDs, security IDs) include name and associated number, one per user
    - User ID then associated with all files, processes of that user to determine access control
    - Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
    - Privilege escalation allows user to change to effective ID with more rights

In the course of normal system use, the user ID and group ID for a user are sufficient. However, a user sometimes needs to **escalate privileges** to gain extra permissions for an activity. The user may need access to a device that is restricted, for example. Operating systems provide various methods to allow privilege escalation. On UNIX, for instance, the *setuid* attribute on a program causes that program to run with the user ID of the owner of the file, rather than the current user's ID. The process runs with this **effective UID** until it turns off the extra privileges or terminates.

Kernel Data Structures

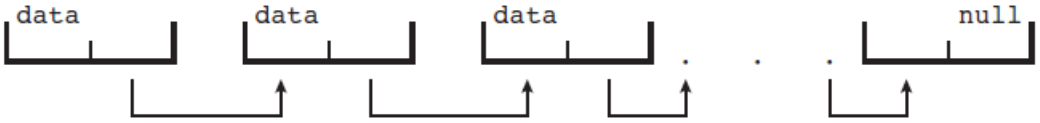
Lists, Stacks, and Queues

An array is a simple data structure in which each element can be accessed directly. For example, main memory is constructed as an array. If the data item being stored is larger than one byte, then multiple bytes can be

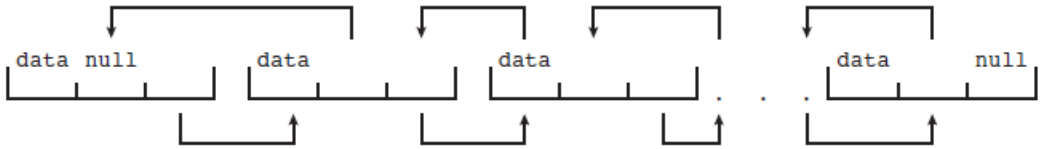
allocated to the item, and the item is addressed as item number × item size. But what about storing an item whose size may vary?

Whereas each item in an array can be accessed directly, the items in a list must be accessed in a particular order. That is, a **list** represents a collection of data values as a sequence. The most common method for implementing this structure is a **linked list**, in which items are linked to one another. Linked lists are of several types:

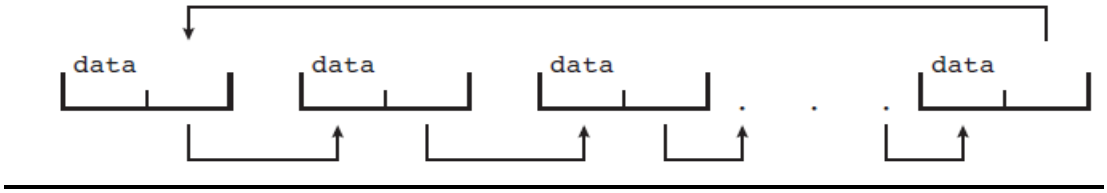
• In a ***singly linked list,*** each item points to its successor, as illustrated in Figure:



• In a ***doubly linked list,*** a given item can refer either to its predecessor or to its successor, as illustrated in Figure:

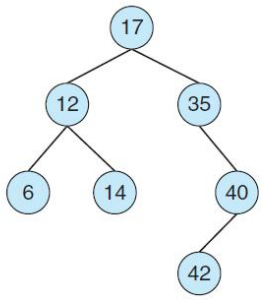


• In a ***circularly linked list,*** the last element in the list refers to the first element, rather than to null, as illustrated in Figure:



Trees

A **tree** is a data structure that can be used to represent data hierarchically. Data values in a tree structure are linked through parent–child relationships. In a **general tree**, a parent may have an unlimited number of children. In a **binary tree**, a parent may have at most two children, which we term the ***left child*** and the ***right child***. A **binary search tree** additionally requires an ordering between the parent's two children in which *le f t child <= right child.* Figure provides an example of a binary search tree.



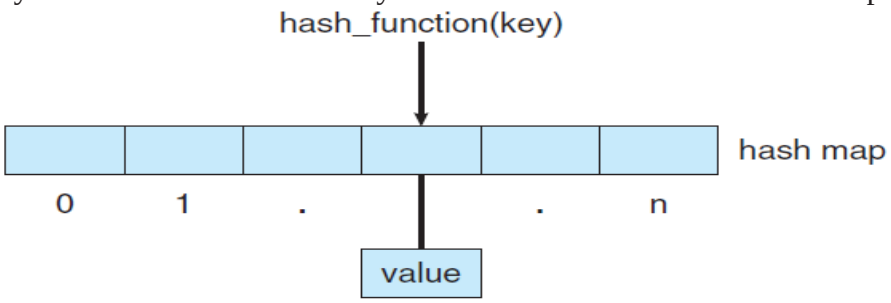
When we search for an item in a binary search tree, the worst-case performance is $O(n)$ (consider how this can occur). To remedy this situation, we can use an algorithm to create a **balanced binary search tree.** Here, a tree containing n items has at most $lg\ n$ levels, thus ensuring worst-case performance of $O(lg\ n)$.

Hash Functions and Maps

A **hash function** takes data as its input, performs a numeric operation on this data, and returns a numeric value. This numeric value can then be used as an index into a table (typically an array) to quickly retrieve the data. Whereas searching for a data item through a list of size *n* can require up to *O*(*n*) comparisons in the worst case, using a hash function for retrieving data from table can be as good as *O*(1) in the worst case, depending on implementation details. Because of this performance, hash functions are used extensively in operating systems.

One potential difficulty with hash functions is that two inputs can result in the same output value—that is, they can link to the same table location. We can accommodate this ***hash collision*** by having a linked list at that table location that contains all of the items with the same hash value.

One use of a hash function is to implement a **hash map**, which associates (or ***maps***) [key:value] pairs using a hash function. For example, we can map the key *operating* to the value *system*. Once the mapping is established, we can apply the hash function to the key to obtain the value from the hash map (Figure).



For example, suppose that a user name is mapped to a password. Password authentication then proceeds as follows: a user enters his user name and password. The hash function is applied to the user name, which is then used to retrieve the password. The retrieved password is then compared with the password entered by the user for authentication.

Bitmaps

A **bitmap** is a string of *n* binary digits that can be used to represent the status of *n* items. For example, suppose we have several resources, and the availability of each resource is indicated by the value of a binary digit: 0 means that there source is available, while 1 indicates that it is unavailable (or vice-versa). The value of the *ith* position in the bitmap is associated with the *ith* resource. As an example, consider the bitmap shown below:

001011101

Resources 2, 4, 5, 6, and 8 are unavailable; resources 0, 1, 3, and 7 are available.

A medium-sized disk drive might be divided into several thousand individual units, called **disk blocks**. A bitmap can be used to indicate the availability of each disk block.

Computing Environments

Traditional Computing

Consider the "typical office environment." Just a few years ago, this environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers.

Web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which provide Web accessibility to their internal servers. **Network computers** (or **thin clients**)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile computers can also connect to **wireless networks** and cellular data networks to use the company's Web portal

These fast data connections are allowing home computers to serve up Web pages and to run networks that include printers, client PCs, and servers. Many homes use **firewalls** to protect their networks from security breaches.

Mobile Computing

Mobile computing refers to computing on handheld smart phones and tablet computers. These devices share the distinguishing physical features of being portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording high-definition video. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on earth.

An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device!

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. Whereas a smart phone or tablet may have 64 GB in storage, it is not uncommon to find 1 TB in storage on a desktop computer.

Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smart phones and tablet computers available from many manufacturers.

Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

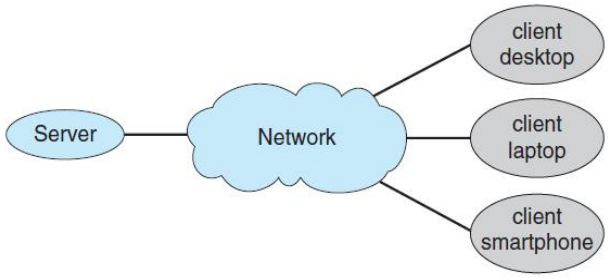
A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a building, or a campus. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks. For example, a **metropolitan-area network (MAN)** could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network(PAN)** between a phone and a headset or a smart phone and a desktop computer.

The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. These networks also vary in their performance and reliability.
A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A distributed operating system provides a less autonomous environment.

Client–Server Computing

Today's systems act as **server systems** to satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system, has the general structure depicted in Figure:



Server systems can be broadly categorized as compute servers and file servers:

• The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.

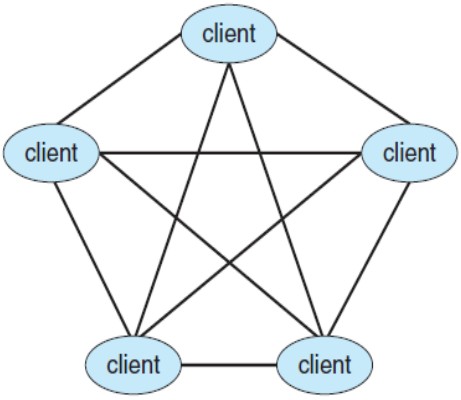
• The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.

Peer-to-Peer Computing

In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network. To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

• When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.

• An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a ***discovery protocol*** must be provided that allows peers to discover services provided by other peers in the network. Figure illustrates the scenario



Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella, that enabled peers to exchange files with one another.

Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP (VoIP)**.

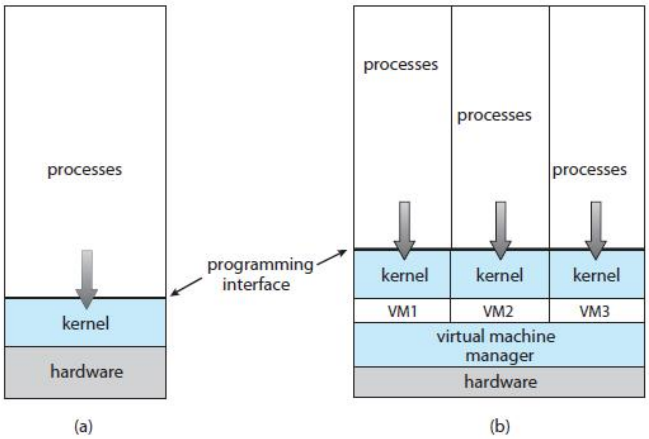
Virtualization

Virtualization is a technology that allows operating systems to run as applications within other operating systems.

Broadly speaking, virtualization is one member of a class of software that also includes emulation. **Emulation** is used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another.

A common example of emulation occurs when a computer language is not compiled to native code but instead is either executed in its high-level form or translated to an intermediate form. This is known as **interpretation**. Some languages, such as BASIC, can be either compiled or interpreted. Java, in contrast, is always interpreted. Thus, we can run Java programs on "Java virtual machines," but technically those virtual machines are Java emulators.

With **virtualization**, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Later, in response to problems with running multiple Microsoft Windows XP applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on XP. That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications.



Windows was the **host** operating system, and the VMware application was the virtual machine manager VMM. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others.

On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running Mac OS X on the x86 CPU can run a Windows guest to allow execution of Windows applications. Companies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware, ESX, and Citrix XenServer no longer run on host operating systems but rather ***are*** the hosts.

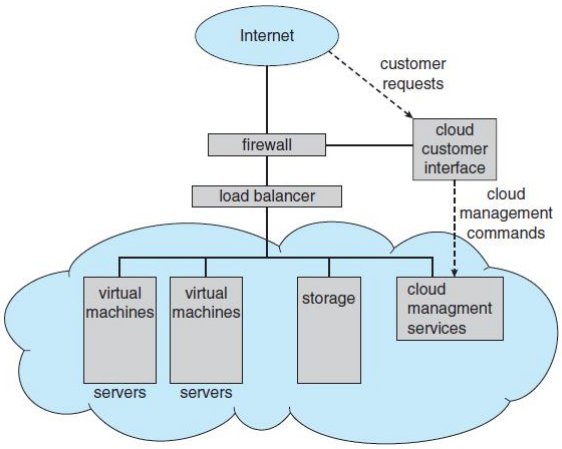
Cloud Computing

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud **(EC2)** facility has thousands of servers, millions of virtual machines, and peta bytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use. There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service **(SaaS)**—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service **(PaaS)**—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service **(IaaS)**—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

At a higher level, the VMMs themselves are managed by cloud management tools, such as Vwarev Cloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Figure illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.



Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerably. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired

Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator can notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it had smashed into the car it was building

Open-Source Operating Systems

Open-source operating systems are those available in source-code format rather than as compiled binary code. Linux is the most famous open source operating system, while Microsoft Windows is a well-known example of the opposite **closed-source** approach. Apple's Mac OS X and iOS operating systems comprise a hybrid approach. They contain an open-source kernel named Darwin yet include proprietary, closed-source components as well.

Source code allows the programmer to produce binary code that can be executed on a system. Doing the opposite—**reverse engineering** the source code from the binaries—is quite a lot of work, and useful items such as comments are never recovered.

Learning operating systems by examining the source code has other benefits as well. With the source code in hand, a student can modify the operating system and then compile and run the code to try out those changes, which is an excellent learning tool.

There are many benefits to open-source operating systems, including a community of interested (and usually unpaid) programmers who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes. Open-source code is more secure than closed-source code because many more eyes are viewing the code. Certainly, open-source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using and viewing the code.

Linux

As an example of an open-source operating system, consider **GNU/Linux**. The GNU project produced many UNIX-compatible tools, including compilers, editors, and utilities, but never released a kernel.

In 1991, a student in Finland, Linus Torvalds, released a rudimentary UNIX-like kernel using the GNU compilers and tools and invited contributions worldwide. The advent of the Internet meant that anyone interested could download the source code, modify it, and submit changes to Torvalds. Releasing updates once a week allowed this so-called Linux operating system to grow rapidly, enhanced by several thousand programmers.

The resulting GNU/Linux operating system has spawned hundreds of unique **distributions**, or custom builds, of the system. Major distributions include RedHat, SUSE, Fedora, Debian, Slackware, and Ubuntu. Distributions vary in function, utility, installed applications, hardware support, user interface, and purpose.

For example, RedHat Enterprise Linux is geared to large commercial use. PCLinuxOS is a **LiveCD**—an operating system that can be booted and run from a CD-ROM without being installed on a system's hard disk. One variant of PCLinuxOS—called "PCLinuxOS Supergamer DVD"—is a **LiveDVD** that includes graphics drivers and games.

You can run Linux on a Windows system using the following simple, free approach:

1. Download the free "VMware Player" tool from

   http://www.vmware.com/download/player/
   and install it on your system.
2. Choose a Linux version from among the hundreds of "appliances," or virtual machine images, available from VMware at

   http://www.vmware.com/appliances/

   These images are preinstalled with operating systems and applications and include many flavors of Linux.
3. Boot the virtual machine within VMware Player.

BSD UNIX

It started in 1978 as a derivative of AT&T's UNIX. Releases from the University of California at Berkeley (UCB) came in source and binary form, but they were not open source because a license from AT&T was required. BSD UNIX's development was slowed by a lawsuit by AT&T, but eventually a fully functional, open-source version, 4.4BSD-lite, was released in 1994.

Just as with Linux, there are many distributions of BSD UNIX, including FreeBSD, NetBSD, OpenBSD, and DragonflyBSD. To explore the source code of FreeBSD, simply download the virtual machine image of the version of interest and boot it within VMware, as described above for Linux. The source code comes with the distribution and is stored in /usr/src/. The kernel source code is in /usr/src/sys. For example, to examine the virtual memory implementation code in the FreeBSD kernel, see the files in /usr/src/sys/vm.

Darwin, the core kernel component of Mac OS X, is based on BSD UNIX and is open-sourced as well. That source code is available from http://www.opensource.apple.com/. Every Mac OS X release has its open source components posted at that site. The name of the package that contains the kernel begins with "xnu." Apple also provides extensive developer tools, documentation, and support at http://connect.apple.com.

Solaris

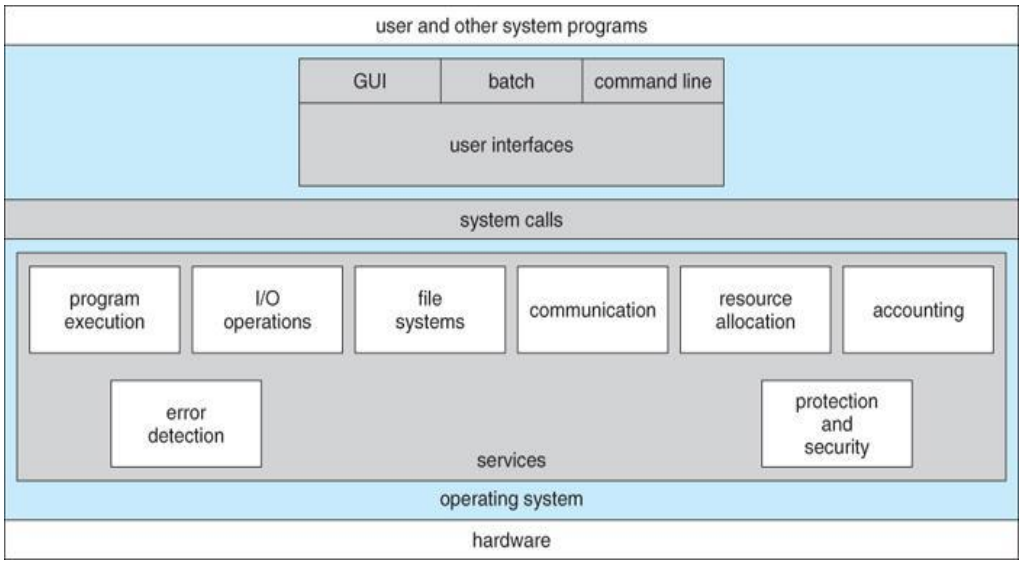
Solaris is the commercial UNIX-based operating system of Sun Microsystems. Originally, Sun's **SunOS** operating system was based on BSD UNIX. Sun moved to AT&T's System V UNIX as its base in 1991. In 2005, Sun open-sourced most of the Solaris code as the OpenSolaris project. The purchase of Sun by Oracle in 2009, however, left the state of this project unclear. The source code as it was in 2005 is still available via a source code browser and for download at http://src.opensolaris.org/source.

Several groups interested in using OpenSolaris have started from that base and expanded its features. Their working set is Project Illumos, which has expanded from the OpenSolaris base to include more features and to be the basis for several products. Illumos is available at http://wiki.illumos.org.

Operating System Structures: Operating System Services, User and Operating-System Interface, System Calls, Types of System Calls, System Programs, Operating-System Structure, Operating System-Debugging, System Boot

Operating system services

OS provide environments in which programs run, and services for the users of the system. These operating system services are provided for the convenience of the programmer, to make the programming task easier. Figure shows one view of the various operating-system services and how they interrelate.



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

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the system these may be a command-line interface (e.g. sh, csh, ksh, tcsh, etc.), a GUI interface (e.g. Windows, X-Windows, KDE, Gnome, etc.), or a batch command systems. The latter are generally older systems using punch cards of job-control language, JCL, but may still be used today for specialty systems designed for a single purpose.
- **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and storage devices.
- **File-System Manipulation** - In addition to raw data storage, the OS is also responsible for maintaining directory and subdirectory structures, mapping file names to specific blocks of data storage, and providing tools for navigating and utilizing the file system.
- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented as either shared memory or message passing, (or some systems may offer both.)
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately, with a minimum of harmful repercussions. Some systems may include complex error avoidance or recovery systems, including backups, RAID drives, and other redundant systems. Debugging and diagnostic tools aid users and administrators in tracing down the cause of problems.

Other systems aid in the efficient operation of the OS itself:

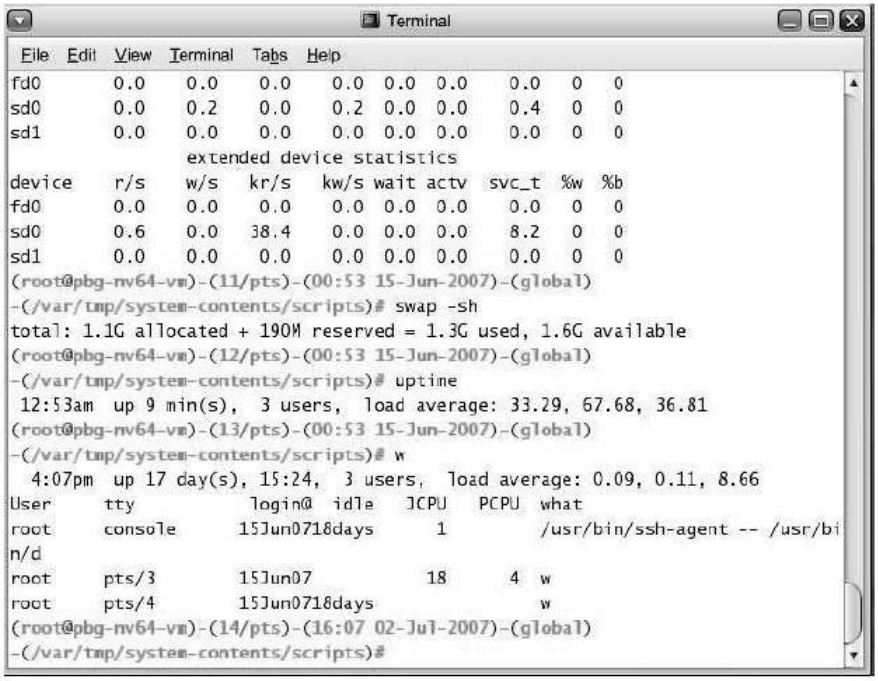
- **Resource Allocation** - E.g. CPU cycles, main memory, storage space, and peripheral devices. Some resources are managed with generic systems and others with very carefully designed and specially tuned systems, customized for a particular resource and operating environment.
- **Accounting** - Keeping track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** - Preventing harm to the system and to resources, either through wayward internal processes or malicious outsiders. Authentication, ownership, and restricted access are obvious parts of this system. Highly secure systems may log all process activity down to excruciating detail, and security regulation dictate the storage of those records on permanent non-erasable medium for extended times in secure (off-site) facilities.

User and Operating-System Interface

Command Interpreters

Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on (on interactive systems). On systems with multiple command interpreters to choose from, the interpreters are known as **shells**.

For example, on UNIX and Linux systems, a user may choose among several different shells, including the ***Bourne shell***, ***C shell***, ***Bourne-Again shell***, ***Korn shell***, and others. Third-party shells and free user-written shells are also available. Most shells provide similar functionality, and a user's choice of which shell to use is generally based on personal preference. Figure shows the Bourne shell command interpreter being used on Solaris 10.



The Bourne Shell command interpreter in solaries 10

The main function of the command interpreter is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way. These commands can be implemented in two general ways.

In one approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call.
An alternative approach—used by UNIX, among other operating systems —implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file

rm file.txt

would search for a file called rm, load the file into memory, and execute it with the parameter file.txt.

Graphical User Interfaces

Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window and-menu system characterized by a **desktop** metaphor. The user moves the mouse to position its pointer on images, or **icons**, on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory—known as a **folder**—or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system (Mac OS) has undergone various changes over the years, the most significant being the adoption of the ***Aqua*** interface that appeared with Mac OS X. Microsoft's first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made cosmetic changes in the appearance of the GUI along with several enhancements in its functionality.

Smart phones and handheld tablet computers typically use a touch screen interface. Here, users interact by making **gestures** on the touch screen—for example, pressing and swiping fingers across the screen. Figure illustrates the touch screen of the Apple iPad. Whereas earlier smart phones included a physical keyboard, most smart phones now simulate a keyboard on the touch screen.

UNIX systems have been dominated by command-line interfaces. Various GUI interfaces are available, however. These include the Common Desktop Environment (CDE) and X-Windows systems, which are common on commercial versions of UNIX, such as Solaris and IBM's AIX system. In addition, there has been significant development in GUI designs from various open-source projects, such as ***K Desktop Environment*** (or ***KDE***) and the ***GNOME*** desktop by the GNU project. Both the KDE and GNOME desktops run on Linux and various UNIX systems and are available under open-source licenses, which means their source code is readily available for reading and for modification under specific license terms.

The iPod touch screen

Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface. For them, it is more efficient, giving them faster access to the activities they need to perform.

Command line interfaces usually make repetitive tasks easier, in part because they have their own programmability. The program is not compiled into executable code but rather is interpreted by the command-line interface. These **shell scripts** are very common on systems that are command-line oriented, such as UNIX and Linux.

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the MS-DOS shell interface. Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI. However, with the release of Mac OS X (which is in part implemented using a UNIX kernel), the operating system now provides both a Aqua interface and a command-line interface. Figure is a screenshot of the Mac OS X GUI.

Mac OS X GUI

Systems calls

- a **system call** is how a program requests a service from an operating system's kernel.
- The **system call** provides an interface to the operating system services.
- System calls provide a means for user or application programs to call upon the services of the operating system.
- Generally written in C or C++, although some are written in assembly for optimal performance.
- The following Figure illustrates the sequence of system calls required to copy a file:

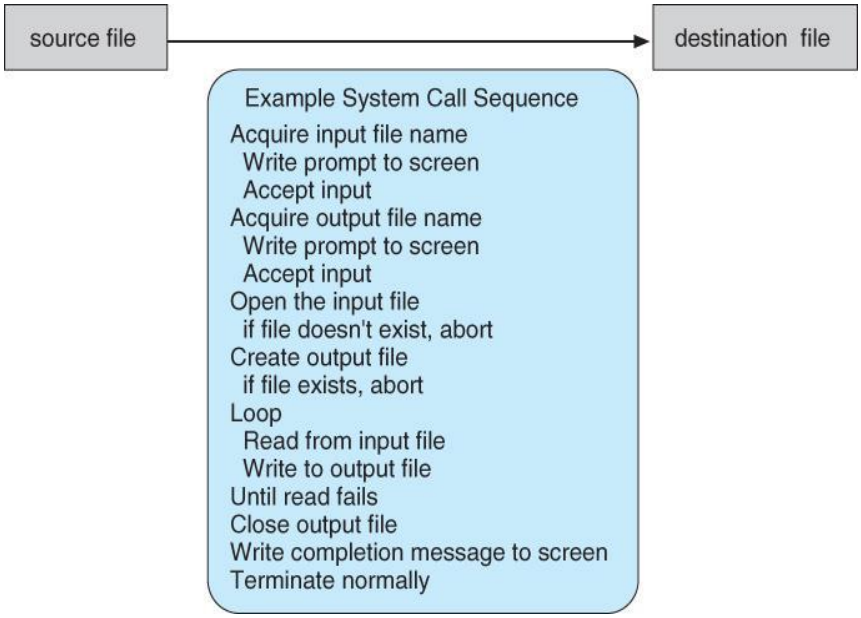


Fig. Example of how system calls are used.

- Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API.
- The use of APIs instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the **system call interface**, using a table lookup to access specific numbered system calls, as shown in Figure :

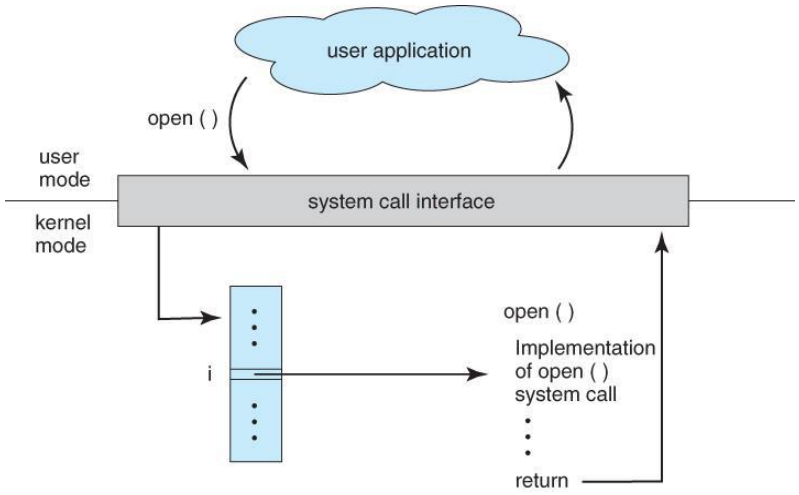
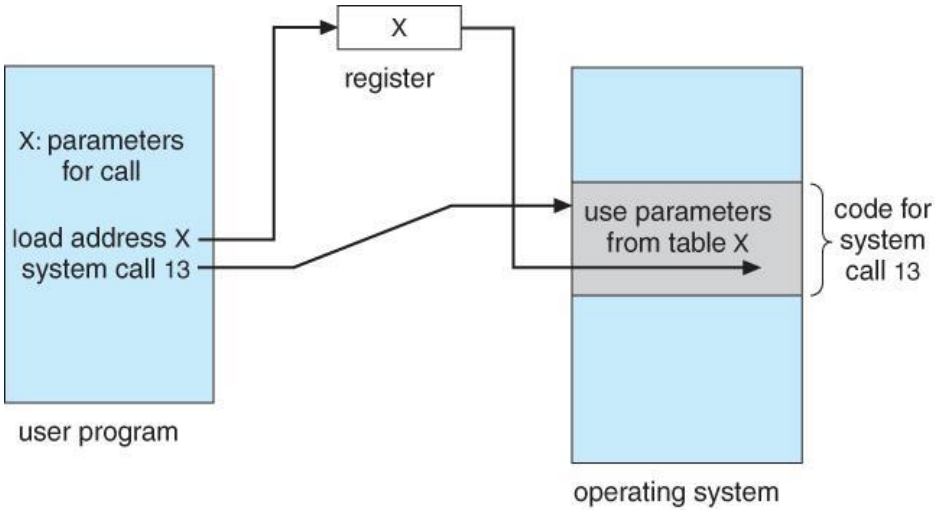


Fig. The handling of a user application invoking the open( ) system call

- Parameters are generally passed to system calls via registers, or less commonly, by values pushed onto the stack. Large blocks of data are generally accessed indirectly, through a memory address passed in a register or on the stack, as shown in Figure:



Passing of parameters as a table

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 2.8 Types of system calls.

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()`

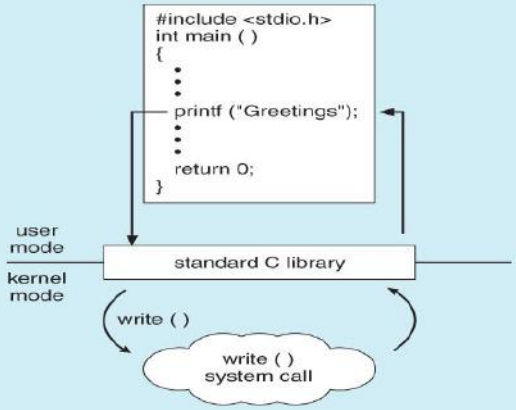
Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

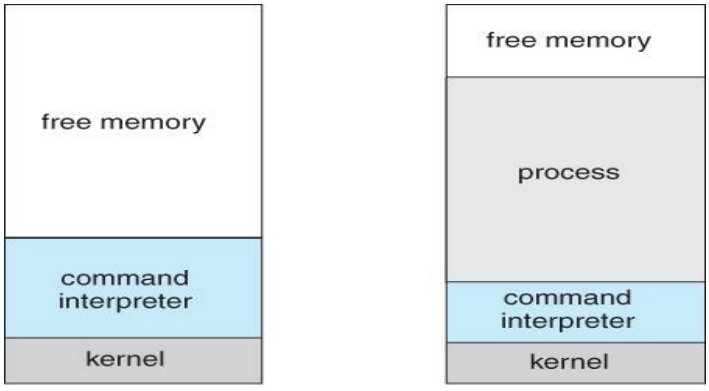
EXAMPLE OF STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. For example, let's assume a C program invokes the printf( ) statement. The C library intercepts this call and invokes the necessary ssystem call(s) in the operating system - in this instance, the write( ) system call. The C library takes the value returned by write( ) and passes it back to the user program. This is shown below:



Process Control

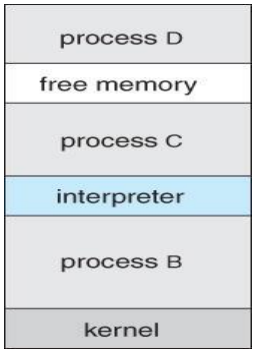
- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed, and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed
- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
- Compare DOS (a single-tasking system) with UNIX (a multi-tasking system).
    - When a process is launched in DOS, the command interpreter first unloads as much of itself as it can to free up memory, then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure :



MS-DOS execution. (a) At system startup. (b) Running a program.

- Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process.
    - The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.

- In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate (clone) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
- The child process then executes an "exec" system call, which replaces its code with that of the desired process.
- The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".)



FreeBSD running multiple programs

File Management

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- These operations may also be supported for directories as well as ordinary files.
- (The actual directory structure may be implemented using ordinary files on the file system, or through other means. Further details will be covered in chapters 11 and 12.)

Device Management

- Device management system calls include request device, release device, read, write, reposition, get/set device attributes, and logically attach or detach devices.
- Devices may be physical (e.g. disk drives), or virtual / abstract (e.g. files, partitions, and RAM disks).
- Some systems represent devices as special files in the file system, so that accessing the "file" calls upon the appropriate device drivers in the OS. See for example the /dev directory on any UNIX system.

Information Maintenance

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- Systems may also provide the ability to dump memory at any time, single step programs pausing execution after each instruction, and tracing the operation of programs, all of which can help to debug programs.

Communication

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The message passing model must support calls to:
  - Identify a remote process and/or host with which to communicate.

    - Establish a connection between the two processes.
    - Open and close the connection as needed.
    - Transmit messages along the connection.
    - Wait for incoming messages, in either a blocking or non-blocking state.
    - Delete the connection when no longer needed.
- The shared memory model must support calls to:
    - Create and access memory that is shared amongst processes (and threads.)
    - Provide locking mechanisms restricting simultaneous access.
    - Free up shared memory and/or dynamically allocate it as needed.
- Message passing is simpler and easier, (particularly for inter-computer communications), and is generally appropriate for small amounts of data.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared, (particularly when most processes are reading the data rather than writing it, or at least when only one or a small number of processes need to change any given data item.)

Protection

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non-priveleged users to be granted elevated access permissions under carefully controlled temporary circumstances.
- Once only of concern on multi-user systems, protection is now important on all systems, in the age of ubiquitous network connectivity.

System Programs

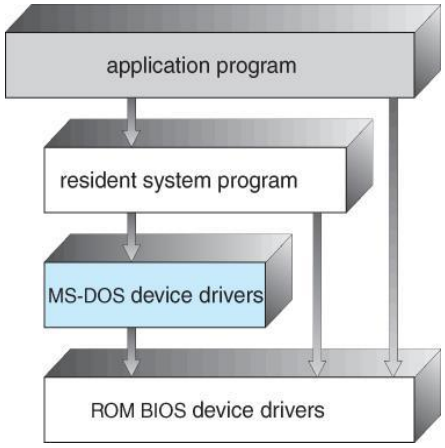
- System programs provide OS functionality through separate applications, which are not part of the kernel or command interpreters. They are also known as **system utilities** or system applications.
- Most systems also ship with useful applications such as calculators and simple editors, (e.g. Notepad). Some debate arises as to the border between system and non-system applications.
- System programs may be divided into these categories:
    - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
    - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
    - **File modification** - e.g. text editors and other tools which can change file contents.
    - **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
    - **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
    - **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
    - **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process schedulers, and system error monitoring services.
- Most operating systems today also come complete with a set of **application programs** to provide additional services, such as copying files or checking the time and date.

Operating-System Structure

For efficient performance and implementation an OS should be partitioned into separate subsystems, each with carefully defined tasks, inputs, outputs, and performance characteristics. These subsystems can then be arranged in various architectural configurations:

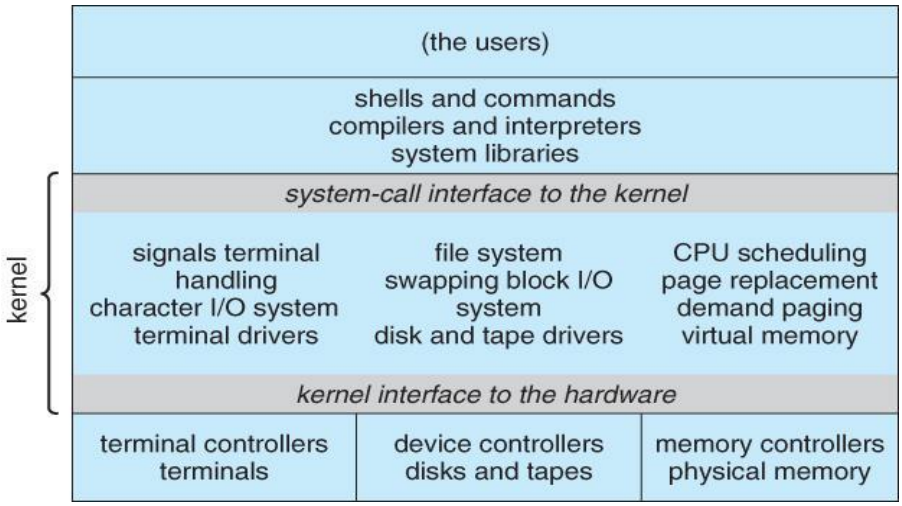
Simple Structure

- When DOS was originally written its developers had no idea how big and important it would eventually become. It was written by a few programmers in a relatively short amount of time, without the benefit of modern software engineering techniques, and then gradually grew over time to exceed its original expectations.
- It does not break the system into subsystems, and has no distinction between user and kernel modes, allowing all programs direct access to the underlying hardware. (Note that user versus kernel mode was not supported by the 8088 chip set anyway, so that really wasn't an option back then.)



MS-DOS layer structure

The original UNIX OS used a simple layered approach, but almost all the OS was in one big layer, not really breaking the OS down into layered subsystems:

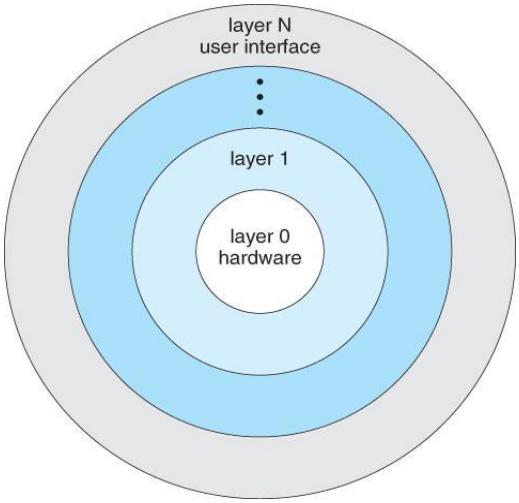


Traditional UNIX system structure

This monolithic structure was difficult to implement and maintain.

Layered Approach

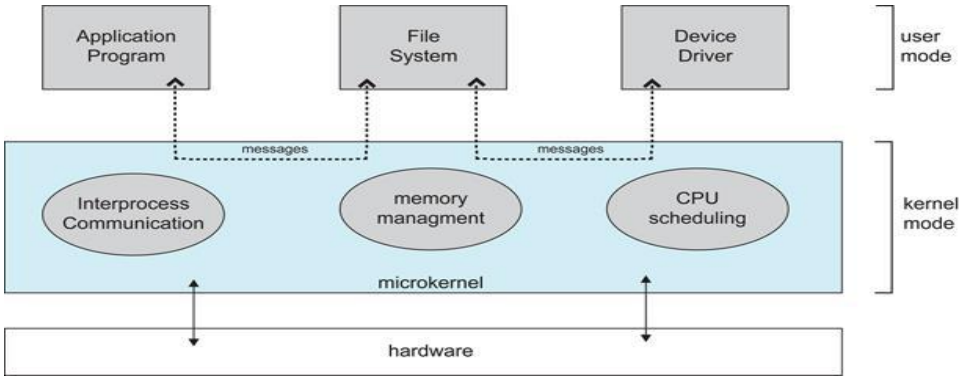
- Another approach is to break the OS into a number of smaller layers, each of which rests on the layer below it, and relies solely on the services provided by the next lower layer.
- This approach allows each layer to be developed and debugged independently, with the assumption that all lower layers have already been debugged and are trusted to deliver proper services.
- The problem is deciding what order in which to place the layers, as no layer can call upon the services of any higher layer, and so many chicken-and-egg situations may arise.
- Layered approaches can also be less efficient, as a request for service from a higher layer has to filter through all lower layers before it reaches the HW, possibly with significant processing at each step.



A layered operating system

Microkernels

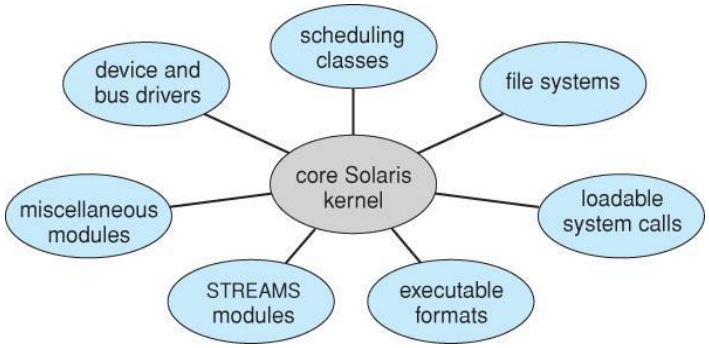
- The basic idea behind micro kernels is to remove all non-essential services from the kernel, and implement them as system applications instead, thereby making the kernel as small and efficient as possible.
- Most microkernels provide basic process and memory management, and message passing between other services, and not much more.
- Security and protection can be enhanced, as most services are performed in user mode, not kernel mode.
- System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
- Windows NT was originally microkernel, but suffered from performance problems relative to Windows 95. NT 4.0 improved performance by moving more services into the kernel, and now XP is back to being more monolithic.
- Another microkernel example is QNX, a real-time OS for embedded systems.



Architecture of a typical microkernel

Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of *modules* which can be linked in dynamically. See for example the Solaris structure, as shown in Figure below.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly.



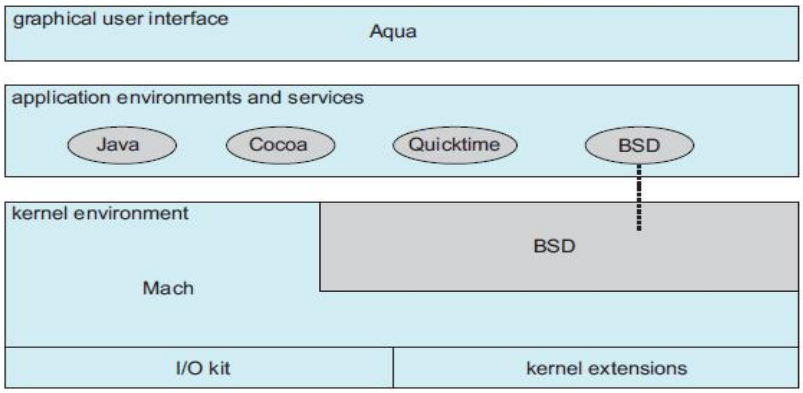
Solaris loadable modules

Hybrid Systems

- 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 behaviour 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.

Mac OS X

- The Apple Mac OS X operating system uses a hybrid structure. As shown in Figure, it is a layered system. The top layers include the ***Aqua*** user interface and a set of application environments and services.

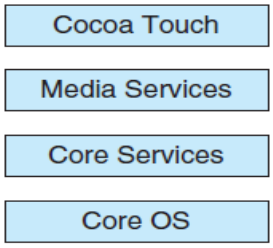


The Mac OS X structure

- Notably, the **Cocoa** environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications. Below these layers is the ***kernel environment***, which consists primarily of the Mach microkernel and the BSD UNIX kernel.
- Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.
- In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules (which Mac OS X refers to as **kernel extensions**). As shown in Figure, the BSD application environment can make use of BSD facilities directly.

iOS

- iOS is a mobile operating system designed by Apple to run its smart phone, the ***iPhone***, as well as its tablet computer, the ***iPad***. iOS is structured on the Mac OS X operating system, with added functionality pertinent to mobile devices, but does not directly run Mac OS X applications. The structure of iOS appears in Figure.



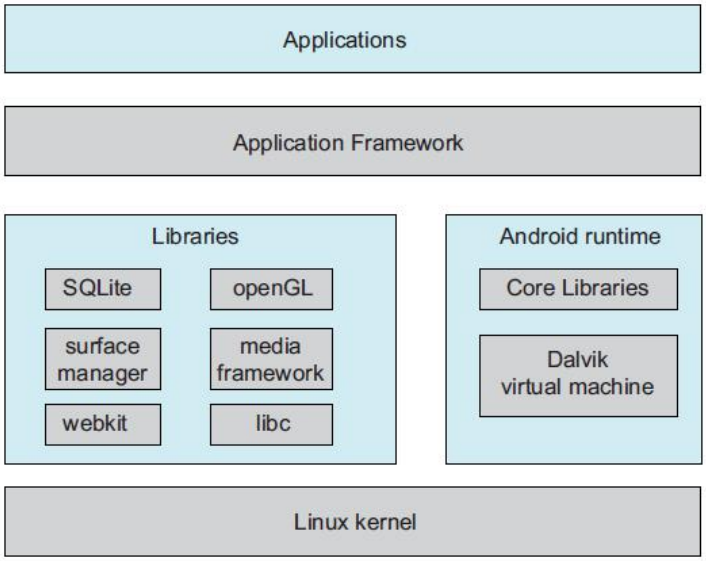
Architecture of Apple's iOS

- **Cocoa Touch** is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices. The fundamental difference between Cocoa, mentioned earlier, and Cocoa Touch is that the latter provides support for hardware features unique to mobile devices, such as touch screens. The **media services** layer provides services for graphics, audio, and video.
- The **core services** layer provides a variety of features, including support for cloud computing and databases. The bottom layer represents the core operating system, which is based on the kernel environment.

Android

- The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smart phones and tablet computers. Whereas iOS is designed to run on Apple mobile devices and is close-sourced, Android runs on a variety of mobile

platforms and is open-sourced, partly explaining its rapid rise in popularity. The structure of Android appears in Figure.



Architecture of Google's Android

- Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks for developing mobile applications. At the bottom of this software stack is the Linux kernel, although it has been modified by Google and is currently outside the normal distribution of Linux releases.
- The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. However, rather than using the standard Java API, Google has designed a separate Android API for Java development.
- The Java class files are first compiled to Java byte code and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities.
- The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia.
- The libc library is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

Operating-System Debugging

Debugging is the activity of finding and fixing errors in a system, both in hardware and in software.

Failure Analysis

If a process fails, most operating systems write the error information to a **log file** to alert system operators or users that the problem occurred. The operating system can also take a **core dump**—a capture of the memory of the process— and store it in a file for later analysis. (Memory was referred to as the "core" in the early days of computing.) Running programs and core dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process.

Debugging user-level process code is a challenge. Operating-system kernel debugging is even more complex because of the size and complexity of the kernel, its control of the hardware, and the lack of user-level debugging tools. A failure in the kernel is called a **crash**. When a crash occurs, error information is saved to a log file, and the memory state is saved to a **crash dump**.

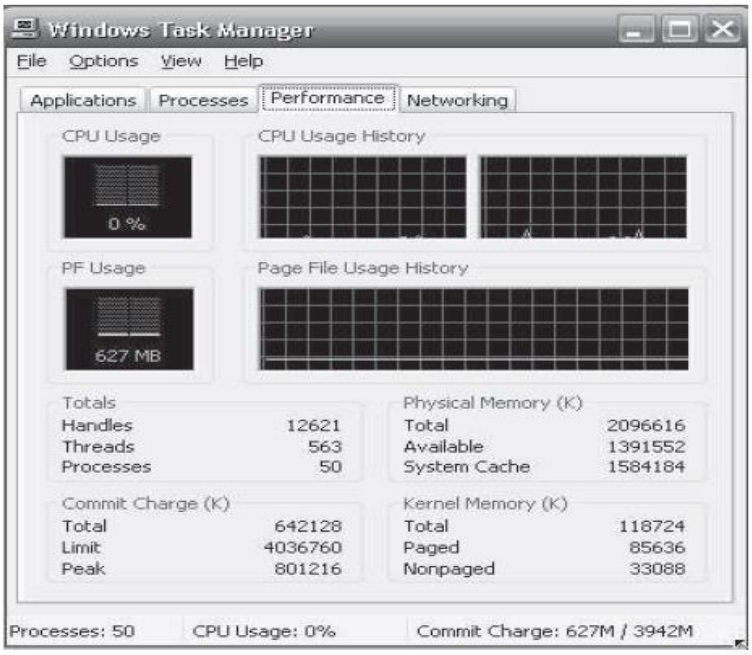
Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks.

Performance Tuning

Performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the operating system must have some means of computing and displaying measures of system behavior. In a number of systems, the operating system does this by producing ***trace listings*** of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies.

Another approach to performance tuning uses single-purpose, interactive tools that allow users and administrators to question the state of various system components to look for bottlenecks. One such tool employs the UNIX command top to display the resources used on the system, as well as a sorted list of the "top" resource-using processes. Other tools display the state of disk I/O, memory allocation, and network traffic.

The **Windows Task Manager** is a similar tool for Windows systems. The task manager includes information for current applications as well as processes, CPU and memory usage, and networking statistics. A screen shot of the task manager appears in Figure



The Windows task manager

DTrace

DTraceis a facility that dynamically adds probes to a running system, both in user processes and in the kernel. These probes can be queried via the D programming language to determine an astonishing amount about the kernel, the system state, and process activities. For example, Figure follows an application as it executes a system call (ioctl()) and shows the functionalcalls within the kernel as they execute to perform the system call. Lines ending with "U" are executed in user mode, and lines ending in "K" in kernel mode.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                          U
  0   -> _XEventsQueued                       U
  0     -> _X11TransBytesReadable             U
  0     <- _X11TransBytesReadable             U
  0     -> _X11TransSocketBytesReadable       U
  0     <- _X11TransSocketBytesreadable       U
  0     -> ioctl                              U
  0       -> ioctl                            K
  0         -> getf                           K
  0           -> set_active_fd                K
  0           <- set_active_fd                K
  0         <- getf                           K
  0         -> get_udatamodel                 K
  0         <- get_udatamodel                 K
...
  0         -> releasef                       K
  0           -> clear_active_fd              K
  0           <- clear_active_fd              K
  0           -> cv_broadcast                 K
  0           <- cv_broadcast                 K
  0         <- releasef                       K
  0       <- ioctl                            K
  0     <- ioctl                              U
  0   <- _XEventsQueued                       U
  0 <- XEventsQueued                          U
```

Solaris 10 `dtrace` follows a system call within the kernel.

Debugging the interactions between user-level and kernel code is nearly impossible without a toolset that understands both sets of code and can instrument the interactions.

Until the DTrace framework and tools became available with Solaris 10, kernel debugging was usually shrouded in mystery and accomplished via happenstance and archaic code and tools. For example, CPUs have a breakpoint feature that will halt execution and allow a debugger to examine the state of the system. Then execution can continue until the next breakpoint or termination. This method cannot be used in a multiuser operating-system kernel without negatively affecting all of the users on the system.

In contrast, DTrace runs on production systems—systems that are running important or critical applications—and causes no harm to the system. It slows activities while enabled, but after execution it resets the system to its pre-debugging state.

DTrace is composed of a compiler, a framework, **providers** of **probes** written within that framework, and **consumers** of those probes. DTrace providers create probes. Kernel structures exist to keep track of all probes that the providers have created. The probes are stored in a hash-table data structure that is hashed by name and indexed according to unique probe identifiers.

When a probe is enabled, a bit of code in the area to be probed is rewritten to call dtrace probe(probe identifier) and then continue with the code's original operation. Different providers create different kinds of probes. For example, a kernel system-call probe works differently from a user-process probe, and that is different from an I/O probe.
DTrace features a compiler that generates a byte code that is run in the kernel. This code is assured to be "safe" by the compiler.

Only users with DTrace "privileges" (or "root" users) are allowed to use DTrace, as it can retrieve private kernel data (and modify data if requested). The generated code runs in the kernel and enables probes. It also enables consumers in user mode and enables communications between the two.

A DTrace consumer is code that is interested in a probe and its results. A consumer requests that the provider create one or more probes. When a probe fires, it emits data that are managed by the kernel. Within the kernel, actions called **enabling control blocks**, or ECBs, are performed when probes fire. One probe can cause multiple ECBs to execute if more than one consumer is interested in that probe. Each ECB contains a predicate ("if statement") that can filter out that ECB. Otherwise, the list of actions in the ECB is executed.

Once the probe consumer terminates, its ECBs are removed. If there are no ECBs consuming a probe, the probe is removed. That involves rewriting the code to remove the dtraceprobe() call and put back the original code.

An example of Dcode and its output shows some of its utility. The following program shows the DTrace code to enable scheduler probes and record the amount of CPU time of each process running with user ID 101 while those probes are enabled (that is, while the program runs):

```
sched:::on-cpu
uid == 101
{
        self->ts = timestamp;
}
sched:::off-cpu
self->ts
{
        @time[execname] = sum(timestamp - self->ts);
        self->ts = 0;
}
```

The output of the program, showing the processes and how much time (in nanoseconds) they spend running on the CPUs, is shown in Figure

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
    gnome-settings-d              142354
    gnome-vfs-daemon              158243
    dsdm                          189804
    wnck-applet                   200030
    gnome-panel                   277864
    clock-applet                  374916
    mapping-daemon                385475
    xscreensaver                  514177
    metacity                      539281
    Xorg                         2579646
    gnome-terminal               5007269
    mixer_applet2                7388447
    java                        10769137
```

The output of the D code

Because DTrace is part of the open-source OpenSolaris version of the Solaris 10 operating system, it has been added to other operating systems when those systems do not have conflicting license agreements. For example, DTrace has been added to Mac OS X and FreeBSD and will likely spread further due to its unique capabilities.

System Boot

After an operating system is generated, it must be made available for use by the hardware. But how does the hardware know where the kernel is or how to load that kernel? The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution.

This program is in the form of **read-only memory (ROM)**, because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, tablets, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory (EPROM)**, which is read only except when explicitly given a command to become writable. All forms of ROM are also known as **firmware**, since their characteristics fall somewhere between those of hardware and those of software. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the operating system in firmware and copy it to RAM for fast execution. A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution.

GRUB is an example of an open-source bootstrap program for Linux systems. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk. A disk that has a boot is called a **boot disk** or **system disk**.

Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.

Processes: Process Concept, Process Scheduling, Operations on Processes, Interprocess Communication, Examples of IPC Systems

Process Concept

- A process is an instance of a program in execution.
- Batch systems work in terms of "**jobs**". Where as in Time Sharing Systems has **User Programs** or **Tasks**.
- Many modern process concepts are still expressed in terms of jobs, (e.g. job scheduling), and the two terms are often used interchangeably.

The Process

- Process memory is divided into four sections as shown in Figure below:
  - The **text** section comprises the compiled program code, read in from non-volatile storage when the program is launched.
  - The **data** section stores global and static variables, allocated and initialized prior to executing main.
  - The **heap** is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
  - The **stack** is used for local variables. Space on the stack is reserved for local variables when they are declared and the space is freed up when the variables go out of scope.
  - Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.
- When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.

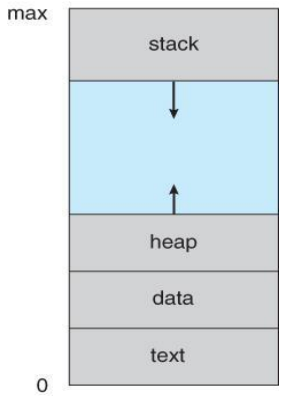


Fig. A process in memory

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. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out).

A process itself can be an execution environment for other code. For example, to run the compiled Java program Program.class, we would enter

java Program

The command java runs the JVM as an ordinary process, which in turns executes the Java program Program in the virtual machine.

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. Processes may be in one of 5 states, as shown in Figure below.
  - **New** - The process is in the stage of being created.
  - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
  - **Running** - The CPU is working on this process's instructions.
  - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
  - **Terminated** - The process has completed.

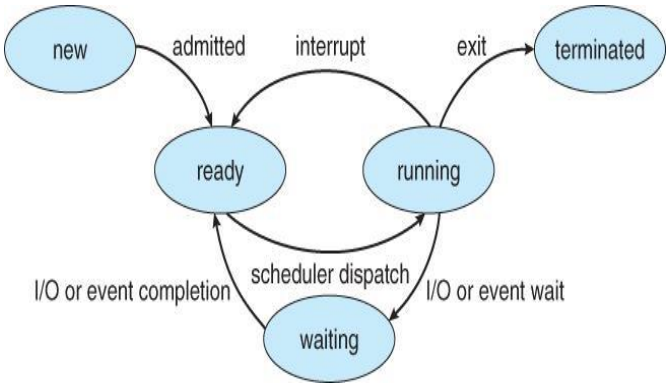


Diagram of process state

Process Control Block

For each process there is a Process Control Block (PCB) or task control block which stores the following (types of) process-specific information, (Specific details may vary from system to system.)

- **Process State** - Running, waiting, etc., as discussed above.
- **Process ID**, and parent process ID.
- **CPU registers and Program Counter** - These need to be saved and restored when swapping processes in and out of the CPU.

- **CPU-Scheduling information** - Such as priority information and pointers to scheduling queues.
- **Memory-Management information** - E.g. page tables or segment tables.
- **Accounting information** - user and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information** - Devices allocated, open file tables, etc.

Process control block (PCB)

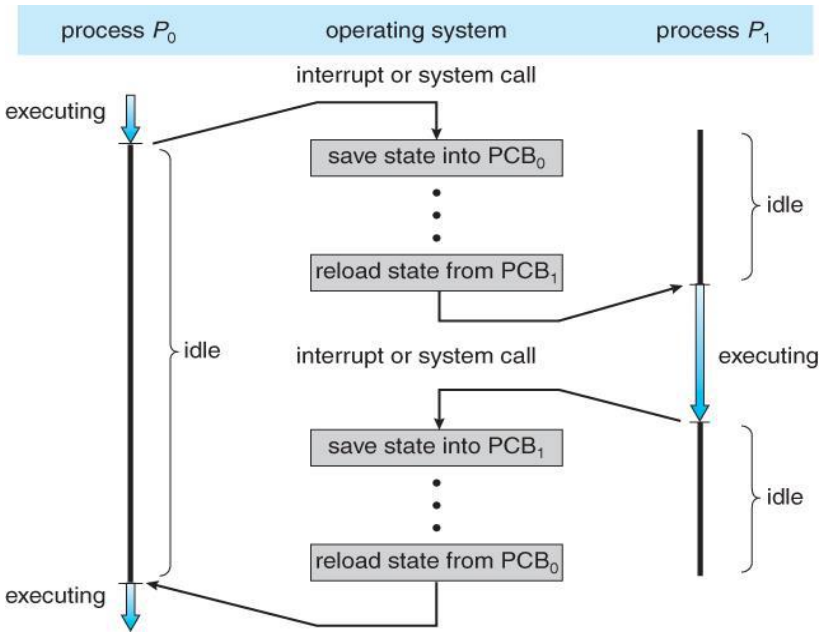


Diagram showing CPU switch from process to process

Threads

A process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time.

The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Most modern operating systems have extended the process concept to allow a process to have multiple

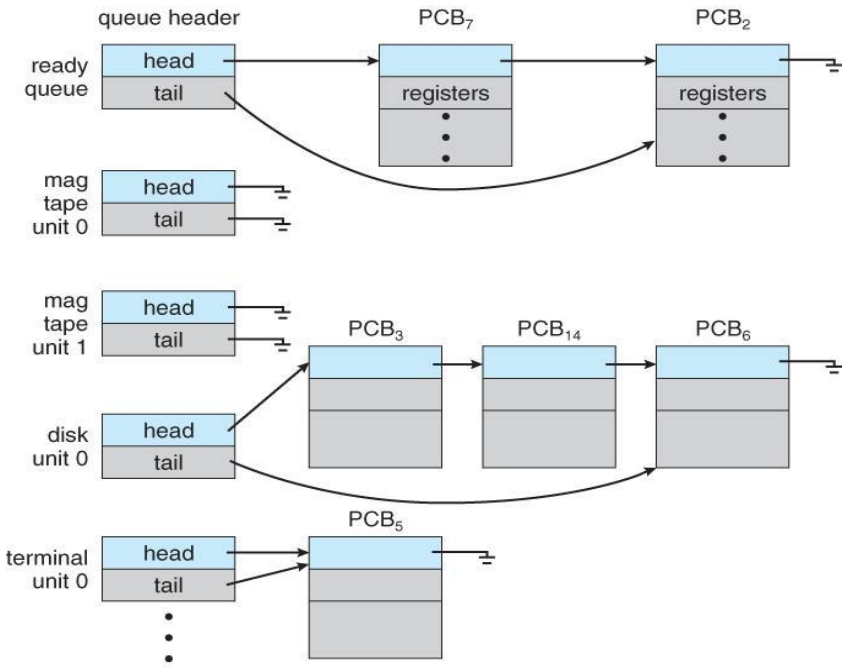
threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel.

Process Scheduling

- The two main objectives of the process scheduling system are to keep the CPU busy at all times and to deliver "acceptable" response times for all programs, particularly for interactive ones.
- The **process scheduler** must meet these objectives by implementing suitable policies for swapping processes in and out of the CPU.
- The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
- Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

Scheduling Queues

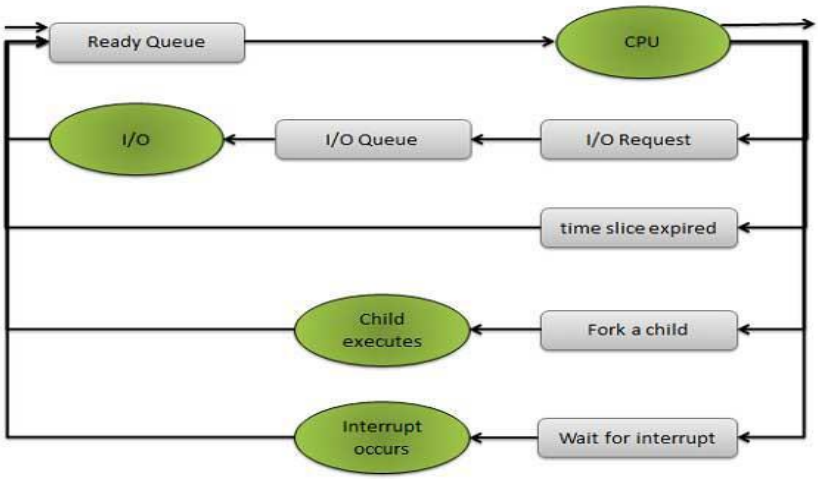
- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a linked list. A ready-queue 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 list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.



The ready queue and various I/O device queues

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Queueing-diagram representation of process scheduling

Queues are of two types

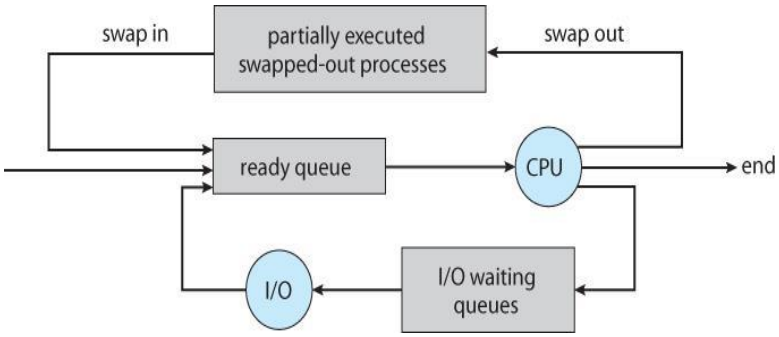
- Ready queue
- Device queue

- A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.
  - The process could issue an I/O request and then it would be placed in an I/O queue.
  - The process could create new sub process and will wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Schedulers

- 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).
- 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. The system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.
- Some systems also employ a **medium-term scheduler**. Medium term scheduling is part of the **swapping.** It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.
- When system loads get high, this scheduler will swap one or more processes out of the ready queue system for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.



Addition of a medium-term scheduling to the queueing diagram

Comparison between Schedulers

S.N.	Long Term Scheduler	Short Term Scheduler	Medium Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch

A context switch is the mechanism to **store and restore the state or context of a CPU in Process Control block** so that a process execution can be resumed from the same point at a later time.

Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved.

Content switching times are highly dependent on hardware support. Context switch requires (n + m) bxK time units to save the state of the processor with n general registers, assuming b are the store operations are required to save n and m registers of two process control blocks and each store instruction requires K time units.

Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

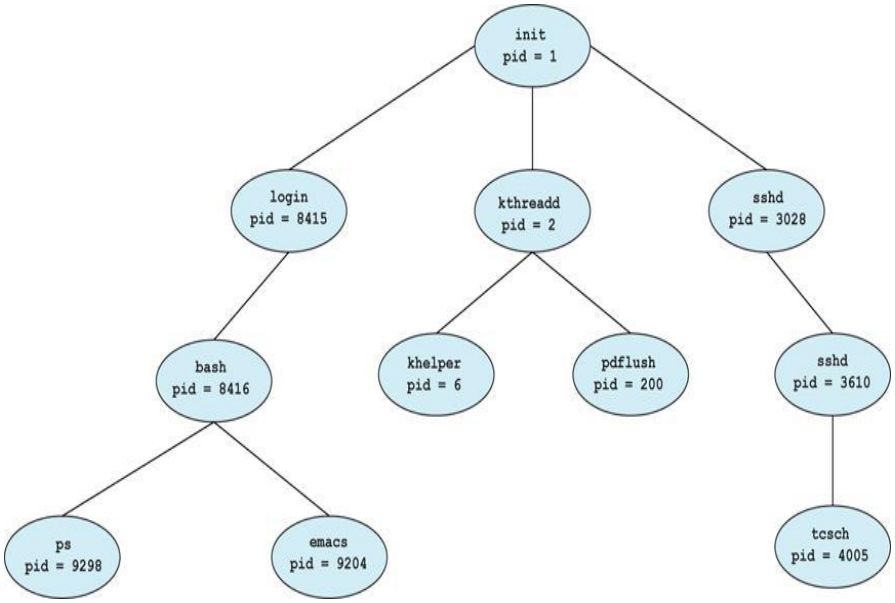
- Program Counter
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting

Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically.

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. Figure illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid.



A tree of processes on a typical Linux system

- 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.
- On UNIX and Linux systems, we can obtain a listing of processes by using the ps command. For example, the command

  ps–el

  will list complete information for all processes currently active in the system.

- 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. For example, consider a process whose function is to display the contents of a file —say, image.jpg—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file ***image.jpg***. Using that file name, it will open the file and write the contents out.

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:

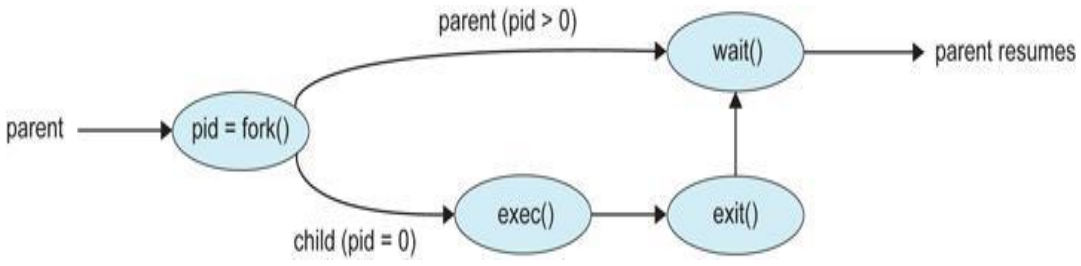
1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

- For example in UNIX operating system A new process is created by the fork() system call. Then the exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Creating a separate process using the UNIX fork() system call

- When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait(), where it completes using the exit() system call. This is also illustrated in Figure



Process creation using the fork( ) system call

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. 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. To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

```
/* exit with status 1 */
exit(1);
```

- In fact, under normal termination, exit() may be called either directly (as shown above) or indirectly (by a return statement in main()). A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

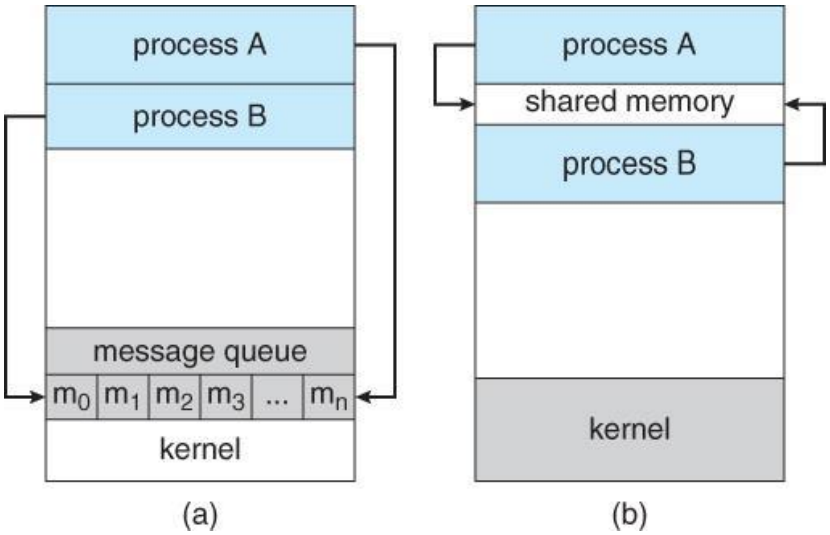
```
pid t pid;
int status;
pid = wait(&status);
```

- 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. Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**.

- Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes. The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Interprocess Communication

- **Independent Processes** operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.
- **Cooperating Processes** are those that can affect or be affected by other processes. There are several reasons why cooperating processes are allowed:
  - **Information Sharing** - There may be several processes which need access to the same file for example. (e.g. pipelines.)
  - **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
  - **Modularity** - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
  - **Convenience** - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.
- Cooperating processes require some type of inter-process communication, which is most commonly one of two types: Shared Memory systems or Message Passing systems. Figure illustrates the difference between the two systems:



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

- **Shared Memory** is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.

- **Message Passing** requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

Shared-Memory Systems

- In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.
- Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

- This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)
- The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.
- This example uses shared memory and a circular queue. Note in the code below that only the producer changes "in", and only the consumer changes "out", and that they can never be accessing the same array location at the same time.
- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10
        typedef struct {
                . . .
        } item;
        item buffer[ BUFFER_SIZE ];
        int in = 0;
        int out = 0;
```

- Then the producer process. Note that the buffer is full when "in" is one less than "out" in a circular sense:

```
// Code
item next_Produced;
while( true ) {
/* Produce an item and store it in next_Produced */
next_Produced = make NewItem( . . . );

/* Wait for space to become available */
```

```
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
    ; /* Do nothing */

/* And then store the item and repeat the loop. */
buffer[ in ] = next_Produced;
in = ( in + 1 ) % BUFFER_SIZE;
}
```

- Then the consumer process. Note that the buffer is empty when "in" is equal to "out":

```
// Code
item next_Consumed;
while( true ) {
/* Wait for an item to become available */
while( in == out )
    ; /* Do nothing */
/* Get the next available item */
next_Consumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in next_Consumed
    ( Do something with it ) */
}
```

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. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

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 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

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.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (which can only receive messages through this mailbox) and the user (which can only send messages to the mailbox). When a process that owns a mailbox terminates, the mailbox disappears.

A mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- Create a new mailbox.
- Send and receive messages through the mailbox.
- Delete a 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.

Different combinations of send() and receive() are possible. When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver.

The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available. This is illustrated in Figures

```
message next_produced;

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

     send(next_produced);
}
```

Figure 3.15 The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

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

Figure 3.16 The consumer process using message passing.

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. The other cases are referred to as systems with automatic buffering.

Examples of IPC Systems

An Example: POSIX Shared Memory

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the shm open() system call, as follows:

shm fd = shm open(name, O_CREAT | O_RDRW, 0666);

The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name. The subsequent parameters specify that the shared-memory object is to be created if it does not yet exist (O_CREAT) and that the object is open for reading and writing (O_RDRW). The last parameter establishes the directory permissions of the shared-memory object.

A successful call to shm open() returns an integer file descriptor for the shared-memory object. Once the object is established, the ftruncate() function is used to configure the size of the object in bytes. The call

ftruncate(shm fd, 4096);

sets the size of the object to 4,096 bytes.

Finally, the mmap() function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.

The programs shown in Figure use the producer–consumer model in implementing shared memory. The producer establishes a shared memory object and writes to shared memory, and the consumer reads from shared memory.

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

Figure 3.17 Producer process illustrating POSIX shared-memory API.

```
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0)

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Figure 3.18 Consumer process illustrating POSIX shared-memory API.

An Example: Mach

The Mach kernel supports the creation and destruction of multiple tasks, which are similar to processes but have multiple threads of control and fewer associated resources. Most communication in Mach—including all inter task information—is carried out by **messages**. Messages are sent to and received from mailboxes, called **ports** in Mach.

Even system calls are made by messages. When a task is created, two special mailboxes—the Kernel mailbox and the Notify mailbox—are also created. The kernel uses the Kernel mailbox to communicate with the task and sends notification of event occurrences to the Notify port.

Only three system calls are needed for message transfer. The msg send() call sends a message to a mailbox. A message is received via msg receive(). Remote procedure calls (RPCs) are executed via msg rpc(), which sends a message and waits for exactly one return message from the sender.

The port allocate() system call creates a new mailbox and allocates space for its queue of messages. The maximum size of the message queue defaults to eight messages. The task that creates the mailbox is that mailbox's owner. The owner is also allowed to receive from the mailbox.

The send and receive operations themselves are flexible. For instance, when a message is sent to a mailbox, the mailbox may be full. If the mailbox is not full, the message is copied to the mailbox, and the sending thread continues. If the mailbox is full, the sending thread has four options:

1. Wait indefinitely until there is room in the mailbox.
2. Wait at most *n* milliseconds.
3. Do not wait at all but rather return immediately.
4. Temporarily cache a message. Here, a message is given to the operating system to keep, even though the mailbox to which that message is being sent is full. When the message can be put in the mailbox, a message is sent back to the sender. Only one message to a full mailbox can be pending at any time for a given sending thread.

The receive operation must specify the mailbox or mailbox set from which a message is to be received. A **mailbox set** is a collection of mailboxes, as declared by the task, which can be grouped together and treated as one mailbox for the purposes of the task. Threads in a task can receive only from a mailbox or mailbox set for which the task has receive access. A port status() system call returns the number of messages in a given mailbox.

The Mach system was especially designed for distributed systems, but Mach was shown to be suitable for systems with fewer processing cores, as evidenced by its inclusion in the Mac OS X system. The major problem with message systems has generally been poor performance caused by double copying of messages: the message is copied first from the sender to the mailbox and then from the mailbox to the receiver. The Mach message system attempts to avoid double-copy operations by using virtual-memory-management techniques

An Example: Windows

Windows provides support for multiple operating environments, or ***subsystems.*** Application programs communicate with these subsystems via a message-passing mechanism. Thus, application programs can be considered clients of a subsystem server.

The message-passing facility in Windows is called the **advanced local procedure call (ALPC)** facility. It is used for communication between two processes on the same machine.

Like Mach, Windows uses a port object to establish and maintain a connection between two processes. Windows uses two types of ports: **connection ports** and **communication ports**.

Server processes publish connection-port objects that are visible to all processes. When a client wants services from a subsystem, it opens a handle to the server's connection-port object and sends a connection request to that port. The server then creates a channel and returns a handle to the client.

The channel consists of a pair of private communication ports: one for client—server messages, the other for server—client messages. Additionally, communication channels support a callback mechanism that allows the client and server to accept requests when they would normally be expecting a reply.

When an ALPC channel is created, one of three message-passing techniques is chosen:

1. For small messages (up to 256 bytes), the port's message queue is used as intermediate storage, and the messages are copied from one process to the other.
2. Larger messages must be passed through a **section object**, which is a region of shared memory associated with the channel.
3. When the amount of data is too large to fit into a section object, an API is available that allows server processes to read and write directly into the address space of a client.

The structure of advanced local procedure calls in Windows is shown in Figure

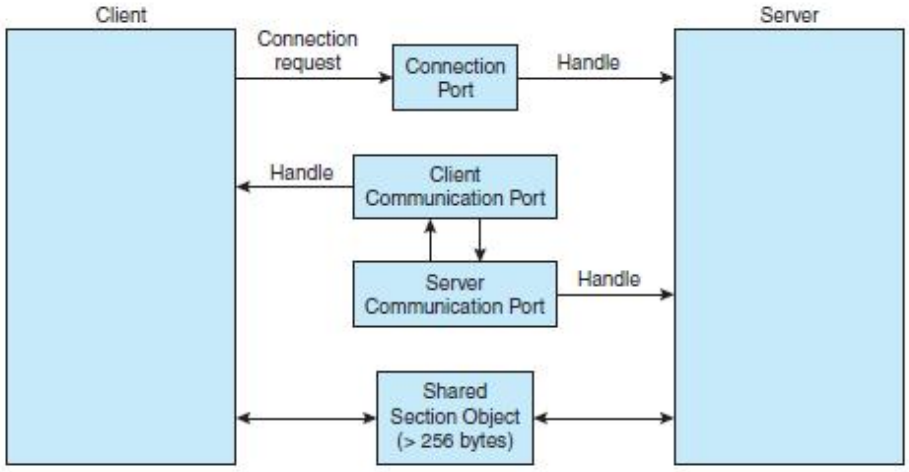


Figure 3.19 Advanced local procedure calls in Windows.

Important Questions from Previous Papers

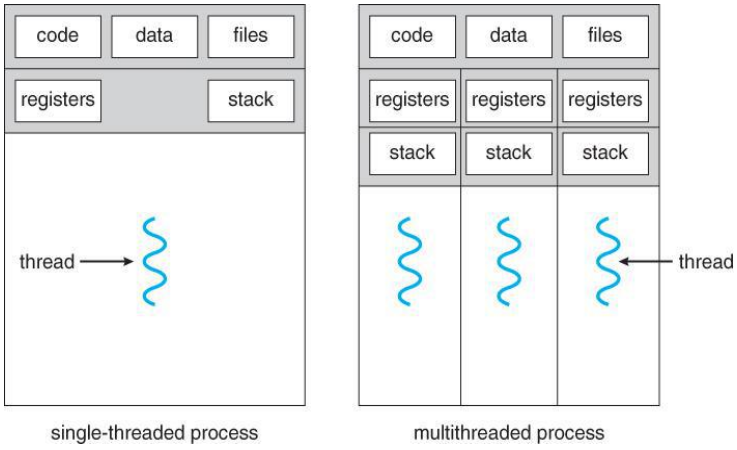
1. a) What are the goals of an operating system?
   b) Explain the services provided by the operating system to the user.
2. a) Enumerate on time-shared operating system.
   b) In what ways is the modular kernel approach similar to the layered approach? In what ways does it differ from the layered approach?
3. Explain the concept of multitasking. Also mention its advantages.
4. What are system calls? Give an example of how system calls are used.
5. a) Explain how the networks are characterized based on the distance between their nodes.
   b) Explain in detail about protection and security offered by operating system in a computer system.
6. What is the main function of a microkernel? What are its benefits?
7. Write notes on the following:
   a) Multitasking.
   b) Real-time embedded systems.
   c) System calls.
8. Discuss the different classes of computer systems whose functions are most limited and whose objective is to deal with limited computation domains.
9. Give a brief note on the operating system structure.
10. a) Discuss in detail distributed systems.
    b) Explain the system calls used in file management.
11. Discuss in detail the functions provided by the operating system.
12. What is the purpose of PCB? Explain various pieces of information contained in PCB.
13. Define the following:
    (i) Job pool. (ii) Job scheduling. (iii) CPU scheduling.
14. What are the benefits and disadvantages of the each of the following? Consider both the system level and the programmer level:
   (a) Synchronous and asynchronous communication.
   (b) Automatic and explicit buffering.
   (c) Send by copy and send by reference.
   (d) Fixed-sized and variable-sized messages.
15. a) Describe the actions taken by a thread library to context switch between user-level threads.
   b) What are independent processes and cooperating processes? What are the reasons for process cooperation?
16. What is the objective of time sharing system? Explain how it is achieved.
17. a) What is the objective of multiprogramming? Explain how it is achieved.
    b) What are the different states a process can be in? Explain process state diagram in detail.
18. a)What is a thread? Illustrate the difference between a traditional single-threaded process and a multithreaded process.
    b) With the help of a queuing diagram explain the working of a medium-term scheduling.
19. a) Explain context switching.
    b) Give a note on process creation.
20. What are the various operations performed on a process? Explain each one in detail.

UNIT-II

Threads: Overview, Multicore Programming, Multithreading Models, Thread Libraries, Implicit Threading, Threading Issues.

Overview

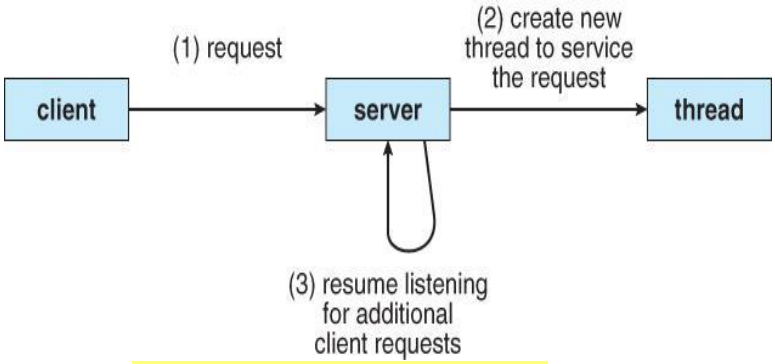
- A ***thread*** is a basic unit of CPU utilization or it is a light weight process, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)
- Traditional (heavy weight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure, multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.



Single-threaded and multithreaded processes

Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- A web browser might have one thread display images or text while another thread retrieves data from the network, for example.For example in a **word processor**, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- A single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it.
- One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests. This is illustrated in Figure



Multithreaded server architecture

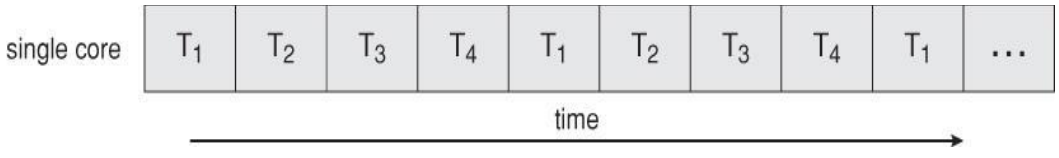
- Finally, most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling. For example, Solaris has a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the system.

Benefits

- There are four major categories of benefits to multi-threading:
    1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
    2. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
    3. **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
    4. **Scalability**, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

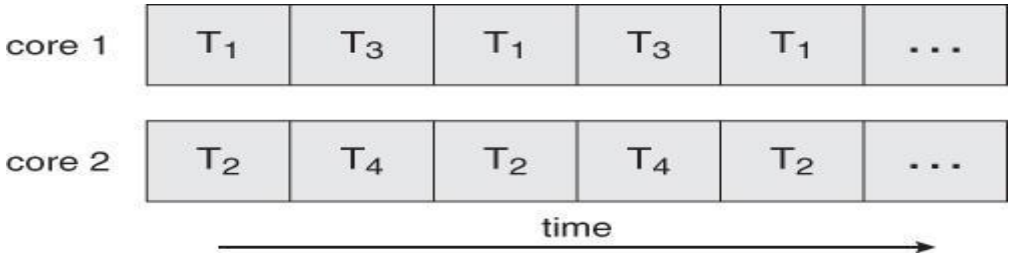
Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure.



Concurrent execution on a single-core system.

- On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in Figure.



Parallel execution on a multicore system

- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- As multi-threading becomes more pervasive and more important (thousands instead of tens of threads), CPUs have been developed to support more simultaneous threads per core in hardware.

Programming Challenges

- For application programmers, there are five areas where multi-core chips present new challenges:
    1. **Identifying tasks** - Examining applications to find activities that can be performed concurrently.
    2. **Balance** - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
    3. **Data splitting** - To prevent the threads from interfering with one another.
    4. **Data dependency** - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
    5. **Testing and debugging** - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

Types of Parallelism

In theory there are two different ways to parallelize the workload:

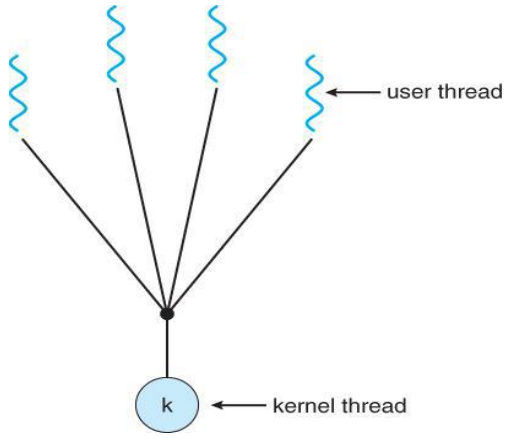
1. **Data parallelism** divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.
2. **Task parallelism** divides the different tasks to be performed among the different cores and performs them simultaneously.

Multithreading Models

- There are two types of threads to be managed in a modern system: **User threads and kernel threads.**
- User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
- Kernel threads are supported within the kernel of the OS itself. All modern OS's support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

Many-To-One Model

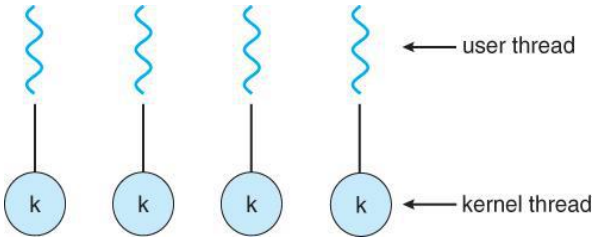
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
- Because a single kernel thread can operate only on a single CPU, the many-to-one model does not allow individual processes to be split across multiple CPUs.
- Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.



Many-to-one model

One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.

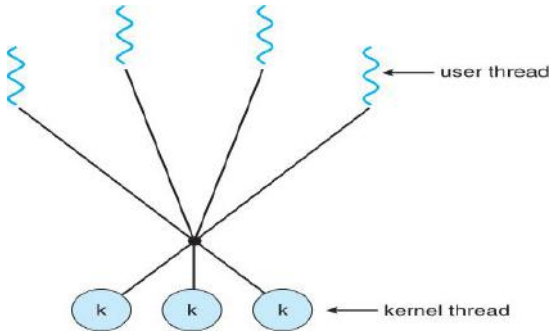


One-to-one model

Many-To-Many Model

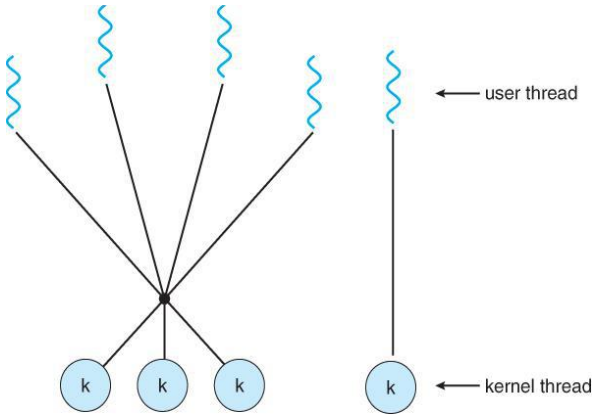
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.

- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.



Many-to-many model

- One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation.
- IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9.



Two-level model

Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
    1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    2. Win32 threads - provided as a kernel-level library on Windows systems.
    3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

- Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes —the so-called ***fork-join*** strategy. Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed. Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution.

Pthreads

- **Pthreads** refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***.
- Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux, Mac OS X, and Solaris. Although Windows doesn't support Pthreads natively, some third party implementations for Windows are available.
- The C program shown in Figure demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread.

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
pthread _t tid; /* the thread identifier */
pthread_attr_t attr; /* set of thread attributes */
if (argc != 2) {
fprintf(stderr,"usage: a.out <integer value>\n");
return -1;
}
if (atoi(argv[1]) < 0) {
fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
return -1;
}
/* get the default attributes */
pthread_ attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);
printf("sum = %d\n",sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{
int i, upper = atoi(param);
sum = 0;
for (i = 1; i <= upper; i++)
sum += i;
pthread_exit(0);
}
```

Multithreaded C program using the Pthreads API.

- All Pthreads programs must include the pthread.h header file. The statement pthread_t tid declares the identifier for the thread we will create.
- The pthread_attr_t attr declaration represents the attributes for the thread. We set the attributes in the function call pthread_attr_init(&attr).
- A separate thread is created with the pthread_create() function call.
- This program follows the fork-join strategy described earlier: after creating the summation thread, the parent thread
- will wait for it to terminate by calling the pthread_join() function.
- The summation thread will terminate when it calls the function pthread_exit(). Once the summation thread has returned, the parent thread will output the value of the shared data sum.

Windows Threads

- Windows thread API in the C program shown in Figure. We must include the windows.h header file when using the Windows API.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

Multithreaded C program using the Windows API.

- Threads are created in the Windows API using the Create Thread() function, and—just as in Pthreads—a set of attributes for the thread is passed to this function. These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.
- Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, as the value is set by the summation thread. In the Windows API using the WaitForSingleObject() function, which causes the creating thread to block until the summation thread has exited.
- Waiting for multiple threads to complete, the WaitForMultipleObjects() function is used. This function is passed four parameters:
  1. The number of objects to wait for
  2. A pointer to the array of objects
  3. A flag indicating whether all objects have been signaled
  4. A timeout duration (or INFINITE)

- For example, if THandles is an array of thread HANDLE objects of size N, the parent thread can wait for all its child threads to complete with this statement:

  WaitForMultipleObjects(N, THandles, TRUE, INFINITE);

Java Threads

- Java threads are available on any system that provides a JVM including Windows, Linux, and Mac OS X. The Java thread API is available for Android applications as well.
- There are two techniques for creating threads in a Java program. One approach is to create a new class that is derived from the Thread class and to override its run() method. An alternative—and more commonly used— technique is to define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
public abstract void run();
}
```

- When a class implements Runnable, it must define a run() method. The code implementing the run() method is what runs as a separate thread.
- Figure shows in next page the Java version of a multithreaded program that determines the summation of a non-negative integer.
- The Summation class implements the Runnable interface. Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.
- Creating a Thread object does not specifically create the new thread; rather, the start() method creates the new thread. Calling the start() method for the new object does two things:
  1. It allocates memory and initializes a new thread in the JVM.
  2. It calls the run() method, making the thread eligible to be run by the JVM. (Note again that we never call the run() method directly. Rather, we call the start() method, and it calls the run() method on our behalf.)
- When the summation program runs, the JVM creates two threads. The first is the parent thread, which starts execution in the main() method. The second thread is created when the start() method on the Thread object is invoked.
- This child thread begins execution in the run() method of the Summation class. After outputting the value of the summation, this thread terminates when it exits from its run() method.

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}

public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Java program for the summation of a non-negative integer.

Implicit Threading

- Programmers must address not only the challenges, but additional difficulties as well. These difficulties, which relate to program correctness.
- One way to address these difficulties and better support the design of multithreaded applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed **implicit threading**, is a popular trend today.
- There are alternative approaches for designing multithreaded programs that can take advantage of multicore processors through implicit threading.

Thread Pools

- Whenever the server receives a request, it creates a separate thread to service the request. Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems.
- The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work. The second issue is more troublesome.
- Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.
- The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request for service.
- Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.
- Thread pools offer these benefits:
  1. Servicing a request with an existing thread is faster than waiting to create a thread.
  2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.
  3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.
- The Windows API provides several functions related to thread pools. Using the thread pool API is similar to creating a thread with the Thread Create() function. Here, a function that is to run as a separate thread is defined. Such a function may appear as follows:

```
DWORD WINAPI PoolFunction(AVOID Param) {
/*
* this function runs as a separate thread.
*/
}
```

- A pointer to PoolFunction() is passed to one of the functions in the thread pool API, and a thread from the pool executes this function. One such member in the thread pool API is the QueueUserWorkItem() function, which is passed three parameters:

• LPTHREAD START ROUTINE Function—a pointer to the function that is to run as a separate thread
• PVOID Param—the parameter passed to Function
• ULONG Flags—flags indicating how the thread pool is to create and manage execution of the thread

- An example of invoking a function is the following:
  QueueUserWorkItem(&PoolFunction, NULL, 0);
- This causes a thread from the thread pool to invoke PoolFunction() on behalf of the programmer. In this instance, we pass no parameters to PoolFunction(). Because we specify 0 as a flag, we provide the thread pool with no special instructions for thread creation.
- The java.util.concurrent package in the Java API provides a thread-pool utility as well.

OpenMP

- OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel.
- Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel. The following C program illustrates a compiler directive above the parallel region containing the printf() statement:

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
/* sequential code */
#pragma omp parallel
{
printf("I am a parallel region.");
}
/* sequential code */
return 0;
}
```

When OpenMP encounters the directive

```
#pragma omp parallel
```

it creates as many threads are there are processing cores in the system. Thus, for a dual-core system, two threads are created, for a quad-core system, four are created; and so forth. All the threads then simultaneously execute the parallel region. As each thread exits the parallel region, it is terminated.

- OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops. For example, assume we have two arrays a and b of size N. We wish to sum their contents and place the results in array c. We can have this task run in parallel by using the following code segment, which contains the compiler directive for parallelizing for loops:

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
c[i] = a[i] + b[i];
}
```

- OpenMP divides the work contained in the for loop among the threads it has created in response to the directive

```
#pragma omp parallel for
```

- OpenMP is available on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems.

Grand Central Dispatch

- Grand Central Dispatch (GCD)—a technology for Apple's Mac OS X and iOS operating systems—is a combination of extensions to the C language, an API, and a run-time library that allows application developers to identify sections of code to run in parallel.
- GCD identifies extensions to the C and C++ languages known as **blocks**. A block is simply a self-contained unit of work. It is specified by a caret ˆ inserted in front of a pair of braces *{ }*. A simple example of a block is shown below:

```
ˆ{ printf("I am a block"); }
```

- GCD schedules blocks for run-time execution by placing them on a **dispatch queue**. When it removes a block from a queue, it assigns the block to an available thread from the thread pool it manages. GCD identifies two types of dispatch queues: ***serial*** and ***concurrent***.
- Blocks placed on a serial queue are removed in FIFO order. Once a block has been removed from the queue, it must complete execution before another block is removed. Each process has its own serial queue (known as its **main queue**).
- Blocks placed on a concurrent queue are also removed in FIFO order, but several blocks may be removed at a time, thus allowing multiple blocks to execute in parallel. There are three system-wide concurrent dispatch queues, and they are distinguished according to priority: low, default, and high. Quite simply, blocks with a higher priority should be placed on the high priority dispatch queue.
- The following code segment illustrates obtaining the default-priority concurrent queue and submitting a block to the queue using the dispatch async() function:

```
dispatch queue t queue = dispatch get global queue
(DISPATCH QUEUE PRIORITY DEFAULT, 0);
dispatch async(queue, ^{ printf("I am a block."); });
```

- Internally, GCD's thread pool is composed of POSIX threads.

Other Approaches

- Other commercial approaches include parallel and concurrent libraries, such as Intel's Threading Building Blocks (TBB) and several products from Microsoft. The Java language and API have seen significant movement toward supporting concurrent programming as well. A notable example Is the java.util.concurrent package, which supports implicit thread creation and management.

Threading Issues

The fork() and exec() System Calls

- 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.
- That is, 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. 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.

- Examples of synchronous signal include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal.

- When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire. Typically, an asynchronous signal is sent to another process.
- A signal may be ***handled*** by one of two possible handlers:
  1. A default signal handler
  2. A user-defined signal handler
- Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.
- Delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered? In general, the following options exist:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.
- The standard UNIX function for delivering a signal is

  kill(pid_t pid, int signal)

  This function specifies the process (pid) to which a particular signal (signal) is to be delivered.

  POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

  pthread_kill(pthread_t tid, int signal)

- Although Windows does not explicitly provide support for signals, it allows us to emulate them using **asynchronous procedure calls (APCs)**. The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.

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.
- 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.
- In Pthreads, thread cancellation is initiated using the pthread_cancel() function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating—and then canceling— a thread:

```
pthread_t tid;
/* create the thread */
pthread_create(&tid, 0, worker, NULL);
. . .
/* cancel the thread */
pthread_cancel(tid);
```

- Invoking pthread_cancel()indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request. Pthreads supports three cancellation modes. Each mode is defined as a state and a type, as illustrated in the table below

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- The default cancellation type is deferred cancellation. Here, cancellation occurs only when a thread reaches a **cancellation point**. One technique for establishing a cancellation point is to invoke the pthread_testcancel() function. If a cancellation request is found to be pending, a function known as a **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated. The following code illustrates how a thread may respond to a cancellation
- The following code illustrates how a thread may respond to a cancellation request using deferred cancellation:

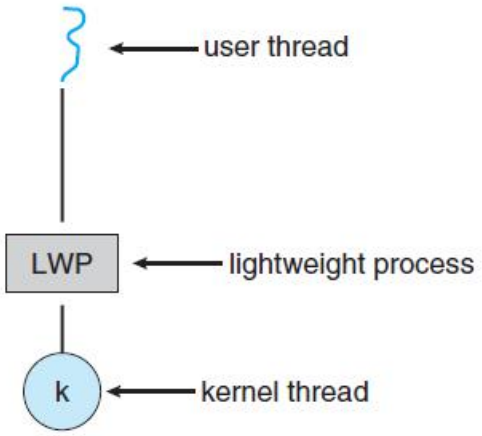
```
while (1) {
/* do some work for awhile */
/* . . . */
/* check if there is a cancellation request */
pthread_test cancel();
}
```

Thread-Local Storage

- Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**.)
- For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-local storage.
- It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations.
- Most thread libraries—including Windows and Pthreads—provide some form of support for thread-local storage; Java provides support as well.

Scheduler Activations

- Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a **lightweight process**, or **LWP**—is shown in Figure



Lightweight Process

- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks.
- One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.
- Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.
- The kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor.

Process Synchronization: Critical Section Problem, Peterson's Solution, Synchronization Hardware, Mutex Locks, Semaphores, Classical Problems of Synchronization Monitors, Synchronization Examples, Alternative Approaches

- Cooperating processes (those that can effect or be effected by other simultaneously running processes), and as an example, we used the producer-consumer cooperating processes:

Producer code:

```
item next_Produced;

while( true ) {

/* Produce an item and store it in next_Produced */
next_Produced = make NewItem( . . . );

/* Wait for space to become available */
while( ( ( in + 1 ) % BUFFER_SIZE ) == out )
    ; /* Do nothing */

/* And then store the item and repeat the loop. */
buffer[ in ] = next_Produced;
in = ( in + 1 ) % BUFFER_SIZE;

}
```

Consumer code:

```
item next_Consumed;

while( true ) {

/* Wait for an item to become available */
while( in == out )
    ; /* Do nothing */

/* Get the next available item */
next_Consumed = buffer[ out ];
out = ( out + 1 ) % BUFFER_SIZE;

/* Consume the item in next_Consumed
   ( Do something with it ) */

}
```

- The only problem with the above code is that the maximum number of items which can be placed into the buffer is BUFFER_SIZE - 1. One slot is unavailable because there always has to be a gap between the producer and the consumer.
- We could try to overcome this deficiency by introducing a counter variable, as shown in the following code segments:

Producer Process:

```
while (true)
{
      /* produce an item in nextProduced */
      while (counter == BUFFER_SIZE)
         ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumer Process:

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

- The Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a ***race condition***. In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process.

- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.

- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level: (1) Fetch counter from memory into a register, (2) increment or decrement the register, and (3) Store the new value of counter back to memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

Producer:

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

Consumer:

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

Interleaving:

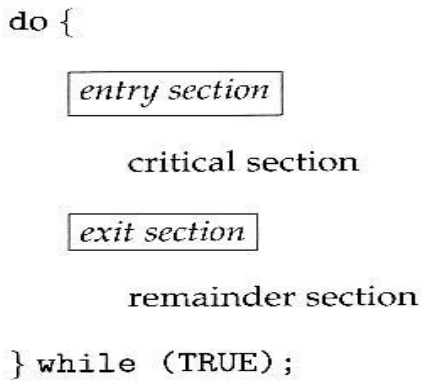
T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- Note that race conditions are ***notoriously difficult*** to identify and debug, because by their very nature they only occur on rare occasions, and only when the timing is just exactly right. (or wrong!) Race conditions are also very difficult to reproduce.
- Obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so lets look at some ways in which this is done, as well as some classic problems in this area.

The Critical-Section Problem.

- A critical section is a code segment that accesses shared variables and has to be executed as an atomic action.
- Critical section problem refers to the problem of how to ensure that at most one process is executing its critical section at a given time.
- The producer-consumer problem described above is a specific example of a more general situation known as the ***critical section*** problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:
    - Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
    - The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
    - The code following the critical section is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.

- The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.



General structure of a typical process Pi

- A solution to the critical section problem must satisfy the following three conditions:
  1. **Mutual Exclusion** - Only one process at a time can be executing in their critical section.
  2. **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (I.e. processes cannot be blocked forever waiting to get into their critical sections.)
  3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted.
- **Kernel processes** can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:
  - **Non-preemptive kernels** do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.
  - **Preemptive kernels** allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.
- Non-preemptive kernels include Windows XP, 2000, traditional UNIX, and Linux prior to 2.6; Preemptive kernels include Linux 2.6 and later, and some commercial UNIXes such as Solaris and IRIX.

Peterson's Solution

- Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.
- Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. (I.e. j = 1 - i)
- Peterson's solution requires two shared data items:

- **int turn** - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
- **boolean flag[2]** - Indicates when a process ***wants to*** enter into their critical section. When process i wants to enter their critical section, it sets flag[i] to true.

- In the following diagram, the entry and exit sections are enclosed in boxes.
  - In the entry section, process i first raises a flag indicating a desire to enter the critical section.
  - Then turn is set to ***j*** to allow the ***other*** process to enter their critical section ***if process j so desires.***
  - The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
  - Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

The structure of process Pi in Peterson's solution.

- To prove that the solution is correct, we must examine the three conditions listed above:
  1. **Mutual exclusion** - If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute "turn = j" will be blocked.
  2. **Progress** - Each process can only be blocked at the while if the other process wants to use the critical section (flag[j] = = true), AND it is the other process's turn to use the critical section (turn = = j). If both of those conditions are true, then the other process (j) will be allowed to enter the critical section, and upon exiting the critical section, will set flag[j] to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.
  3. **Bounded Waiting** - As each process enters their entry section, they set the turn variable to be the other processes turn. Since no process ever sets it back to their own turn, this ensures that each process will have to let the other process go first at most one time before it becomes their turn again.
- Note that the instruction "turn = j" is ***atomic,*** that is it is a single machine instruction which cannot be interrupted.

Synchronization Hardware

- To generalize the solution(s) expressed above, each process when entering their critical section must set some sort of ***lock***, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed.
- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels.
- Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors. There is also a question as to how this approach affects timing if the clock interrupt is disabled.
- Another approach is for hardware to provide certain ***atomic*** operations. These operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value, as shown in Figures

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 5.3 The definition of the `TestAndSet()` instruction.

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with `TestAndSet()`.

- The compare and swap() instruction, in contrast to the test and set() instruction, operates on three operands; it is defined in Figure 5.5. The operand value is set to new value only if the expression (*value == exected) 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 as shown in Figures

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

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

   return temp;
}
```

Figure 5.5 The definition of the compare_and_swap() instruction.

- 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

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

      /* critical section */

   lock = 0;

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

Figure 5.6 Mutual-exclusion implementation with the compare_and_swap() instruction.

- The above examples satisfy the mutual exclusion requirement, but unfortunately do not guarantee **bounded waiting**. If there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter, and any one process could have the bad luck to wait forever until they got their turn in the critical section.

- The following figure illustrates a solution using test-and-set that does satisfy this requirement, using two shared data structures, boolean lock and boolean waiting[N], where N is the number of processes in contention for critical sections:

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

Bounded-waiting mutual exclusion with TestAndSet( ).

- The key feature of the above algorithm is that a process blocks on the AND of the critical section being locked and that this process is in the waiting state. When exiting a critical section, the exiting process does not just unlock the critical section and let the other processes have a free-for-all trying to get in. Only if there are no other processes currently waiting is the general lock removed, allowing the next process to come along access to the critical section.
- Unfortunately, hardware level locks are especially difficult to implement in multi-processor architectures.

Mutex Locks

- The hardware-based solutions to the critical-section problem presented 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 ***mut***ual ***ex***clusion.)
- 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

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

- 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**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available.
- This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.
- Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

Semaphores

- The hardware solutions described above can be difficult for application programmers to implement.
- An alternative is to use ***semaphores(S)***, which are shared integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.
- Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It IS okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever.

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

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

Semaphore Usage

- In practice, semaphores can take on one of two forms:
  - **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that provide a separate mutex mechanism. The use of mutexes for this purpose is shown in Figure below.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

Mutual-exclusion implementation with semaphores

- **Counting semaphores** can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one.)
- Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.
  - First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.
  - Then in process P1 we insert the code:

```
        S1;
signal(synch);
```

  - and in process P2 we insert the code:

```
wait(synch);
S2;
```

  - Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

Semaphore Implementation

- The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.
- An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.)
- The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

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

Wait Operation:

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

Signal Operation:

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

- Note that in this implementation the value of the semaphore can actually become negative, in which case its magnitude is the number of processes waiting for that semaphore. This is a result of decrementing the counter before checking its value.
- Key to the success of semaphores is that the wait and signal operations be atomic, that is no other process can execute a wait or signal on the same semaphore at the same time. (Other processes could be allowed to do other things, including working with other semaphores, they just can't have access to **this** semaphore.) On single processors this can be implemented by disabling interrupts during the execution of wait and signal; Multiprocessor systems have to use more complex methods, including the use of spinlocking.

Deadlocks and Starvation

- One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of ***deadlocks***, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other (blocked) processes, as illustrated in the following example.

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- Another problem to consider is that of ***starvation***, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

Priority Inversion

- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a ***priority inversion.*** If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.

$$L < M < H$$

- One solution is a ***priority-inheritance protocol,*** in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily inherit the high priority from the waiting

process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.

Classic Problems of Synchronization

The following classic problems are used to test virtually every new proposed synchronization algorithm.

The Bounded-Buffer Problem

- This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.
- In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively (and initialized to 0 and N respectively.) The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.

```
do {
   . . .
   // produce an item in nextp
   . . .
   wait(empty);
   wait(mutex);
   . . .
   // add nextp to buffer
   . . .
   signal(mutex);
   signal(full);
}while (TRUE);
```

Figure 5.9 The structure of the producer process.

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

Figure 5.10 The structure of the consumer process.

The Readers-Writers Problem

- In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data

in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.

- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
    - The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)
    - The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.
- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:
    - readcount is used by the reader processes, to count the number of readers currently accessing the data.
    - mutex is a semaphore used only by the readers for controlled access to readcount.
    - rw_mutex is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch rw_mutex. (Eighth edition called this variable wrt.)
    - Note that the first reader to come along will block on rw_mutex if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount.

- Some hardware implementations provide specific reader-writer locks, which are accessed using an argument specifying whether access is requested for reading or writing. The use of reader-writer locks is beneficial for situation in which: (1) processes can be easily identified as either readers or writers, and (2) there are significantly more readers than writers, making the additional overhead of the reader-writer lock pay off in terms of increased concurrency of the readers.

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

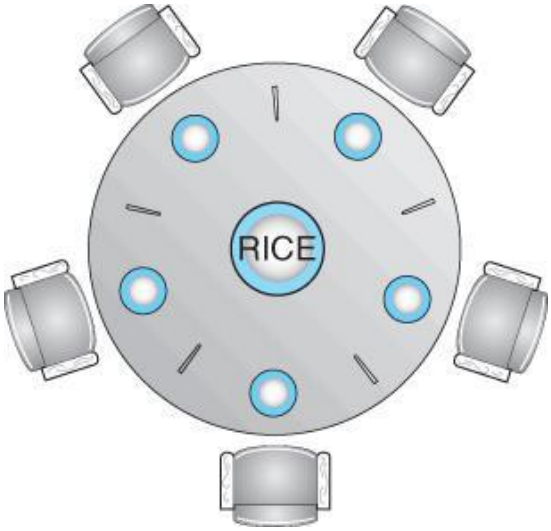
Figure 5.11 The structure of a writer process.

```
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 5.12 The structure of a reader process.

The Dining-Philosophers Problem

- The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:
    - Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center, as shown in the diagram below. (There is exactly one chopstick between each pair of dining philosophers.)
    - These philosophers spend their lives alternating between two activities: eating and thinking.
    - When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
    - When a philosopher thinks, it puts down both chopsticks in their original locations.



The situation of the dining philosophers

- One possible solution, as shown in the following code section, is to use a set of five semaphores (chopsticks[5]), and to have each hungry philosopher first wait on their left chopstick (chopsticks[i]), and then wait on their right chopstick (chopsticks[(i + 1) % 5])
- But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

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

The structure of philosopher i.

- Some potential solutions to the problem include:
    - Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
    - Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)

- Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first. (Will this solution always work? What if there are an even number of philosophers?)

- Note carefully that a deadlock-free solution to the dining philosophers problem does not necessarily guarantee a starvation-free one.

Monitors

- Semaphores can be very useful for solving concurrency problems, ***but only if programmers use them properly.*** If even one process fails to abide by the proper use of semaphores, either accidentally or deliberately, then the whole system breaks *down.*
- All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering into the critical section and signal(mutex) afterward. Next examine the various difficulties that may result.
- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    …..
  critical section
    …..
wait(mutex);
```

  In this situation, several processes may be executing in their critical sections simultaneously , violating the mutual-exclusion requirement.

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
    …..
   critical section
  …..
wait(mutex);
```

  In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both.

  In this case, either mutual exclusion is violated or a deadlock will occur.

- For this reason a higher-level language construct has been developed, called ***monitors***.

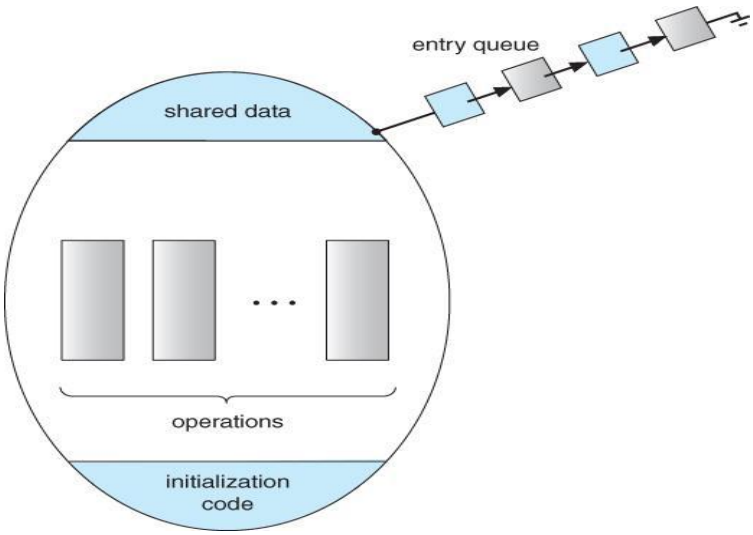
Monitor Usage

- A monitor is essentially a class, in which all data is private, and with the special restriction that only one method within any given monitor object may be active at the same time. An additional restriction is that monitor methods may only access the shared data within the monitor and any data passed to them as parameters. I.e. they cannot access any data external to the monitor.

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

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Syntax of a monitor.

- Figure shows a schematic of a monitor, with an entry queue of processes waiting their turn to execute monitor operations (methods.)



Schematic view of a monitor

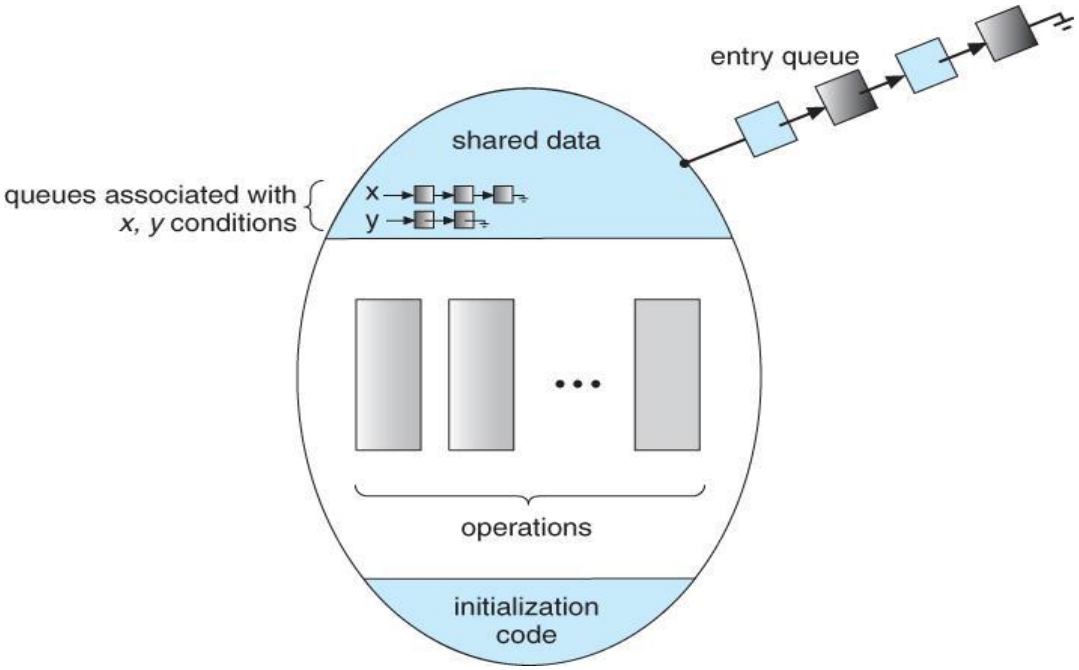
- In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a ***condition***.
    - A variable of type condition has only two legal operations, ***wait*** and ***signal***. I.e. if x was defined as type

      condition, condition x;

      then legal operations would be x.wait( ) and y.signal( )

- The wait operation blocks a process until some other process calls signal, and adds the blocked process onto a list associated with that condition.
- The signal process does nothing if there are no processes waiting on that condition. Otherwise it wakes up exactly one process from the condition's list of waiting processes. (Contrast this with counting semaphores, which always affect the semaphore on a signal call.)

- Figure below illustrates a monitor that includes condition variables within its data space. Note that the condition variables, along with the list of processes currently waiting for the conditions, are in the data space of the monitor - The processes on these lists are not "in" the monitor, in the sense that they are not executing any code in the monitor.



Monitor with condition variables

- But now there is a potential problem - If process P within the monitor issues a signal that would wake up process Q also within the monitor, then there would be two processes running simultaneously within the monitor, violating the exclusion requirement. Accordingly there are two possible solutions to this dilemma:
  Signal and wait - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.
  Signal and continue - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

  There are arguments for and against either choice. Concurrent Pascal offers a third alternative - The signal call causes the signaling process to immediately exit the monitor, so that the waiting process can then wake up and proceed.

- Java and C# (C sharp) offer monitors bulit-in to the language. Erlang offers similar but different constructs.

Dining-Philosophers Solution Using Monitors

- This solution to the dining philosophers uses monitors, and the restriction that a philosopher may only pick up chopsticks when both are available. There are also two key data structures in use in this solution:
    1. **enum { THINKING, HUNGRY,EATING } state[5];** A philosopher may only set their state to eating when neither of their adjacent neighbors is eating. (state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING).
    2. **condition self[5];** This condition is used to delay a hungry philosopher who is unable to acquire chopsticks.
- In the following solution philosophers share a monitor, DiningPhilosophers, and eat using the following sequence of operations:

    1. DiningPhilosophers.pickup( ) - Acquires chopsticks, which may block the process.
    2. eat
    3. DiningPhilosophers.putdown( ) - Releases the chopsticks.

```
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 5.18 A monitor solution to the dining-philosopher problem.

Implementing a Monitor Using Semaphores

- One possible implementation of a monitor uses a semaphore "mutex" to control mutual exclusionary access to the monitor, and a counting semaphore "next" on which processes can suspend themselves after they are already "inside" the monitor (in conjunction with condition variables, see below.) The integer next_count keeps track of how many processes are waiting in the next queue. Externally accessible monitor processes are then implemented as:

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

- Condition variables can be implemented using semaphores as well. For a condition x, a semaphore "x_sem" and an integer "x_count" are introduced, both initialized to zero. The wait and signal methods are then implemented as follows. (This approach to the condition implements the signal-and-wait option described above for ensuring that only one process at a time is active inside the monitor.)

Wait:

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

Signal:

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

Resuming Processes within a Monitor

- When there are multiple processes waiting on the same condition within a monitor, how does one decide which one to wake up in response to a signal on that condition? One obvious approach is FCFS, and this may be suitable in many cases and in some times it is not adequate.
- For this purpose, the **conditional-wait** construct cab be used; it has the form

x.wait(c);

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **Priority Number,** is the stored with the name of the process that is suspended.

- is to assign (integer) priorities, and to wake up the process with the smallest (best) priority.
- When x.signal() is executed, the process with the smallest priority number is resumed next.
- A process that needs to access the resource in question must observe the following sequence:

```
R.aquire(t);
   …
   access the resource;
   ….
R.release();
```

Where R is an instance of type ResourceAllocator.

- Figure below illustrates the use of such a condition within a monitor used for resource allocation. Processes wishing to access this resource must specify the time they expect to use it using the acquire(time) method, and must call the release( ) method when they are done with the resource.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

A monitor to allocate a single resource.

- Unfortunately the use of monitors to restrict access to resources still only works if programmers make the requisite acquire and release calls properly. One option would be to place the resource allocation code into the monitor, thereby eliminating the option for programmers to bypass or ignore the monitor, but then that would substitute the monitor's scheduling algorithms for whatever other scheduling algorithms may have been chosen for that particular resource.
- Concurrent Pascal, Mesa, C#, and Java all implement monitors as described here. Erlang provides concurrency support using a similar mechanism.

Synchronization Examples

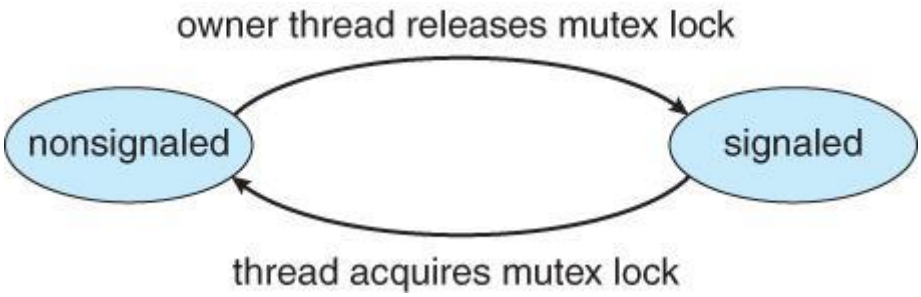
This section looks at how synchronization is handled in a number of different systems.

Synchronization in Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
    - Spin locking-thread will never be preempted
- Also provides dispatcher objects user-land which may act mutexes, semaphores, events, and timers
    - Events

      -An event acts much like a condition variable

    - Timers notify one or more thread when time expired
    - Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block)



Mutex dispatcher object

Synchronization in Linux

- Linux:
    - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
    - Version 2.6 and later, fully preemptive
- Linux provides:
    - semaphores
    - spinlocks
    - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Synchronization in Solaris

- Solaris controls access to critical sections using five tools: semaphores, condition variables, adaptive mutexes, reader-writer locks, and turnstiles. The first two are as described above, and the other three are described here:

Adaptive Mutexes

- Adaptive mutexes are basically binary semaphores that are implemented differently depending upon the conditions:
    - On a single processor system, the semaphore sleeps when it is blocked, until the block is released.
    - On a multi-processor system, if the thread that is blocking the semaphore is running on the same processor as the thread that is blocked, or if the blocking thread is not running at all, then the blocked thread sleeps just like a single processor system.
    - However if the blocking thread is currently running on a different processor than the blocked thread, then the blocked thread does a spinlock, under the assumption that the block will soon be released.
    - Adaptive mutexes are only used for protecting short critical sections, where the benefit of not doing context switching is worth a short bit of spinlocking. Otherwise traditional semaphores and condition variables are used.

Reader-Writer Locks

- Reader-writer locks are used only for protecting longer sections of code which are accessed frequently but which are changed infrequently.

Turnstiles

- A turnstile is a queue of threads waiting on a lock.
- Each synchronized object which has threads blocked waiting for access to it needs a separate turnstile. For efficiency, however, the turnstile is associated with the thread currently holding the object, rather than the object itself.
- In order to prevent priority inversion, the thread holding a lock for an object will temporarily acquire the highest priority of any process in the turnstile waiting for the blocked object. This is called a priority-inheritance protocol.
- User threads are controlled the same as for kernel threads, except that the priority-inheritance protocol does not apply.

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Alternative Approaches

Multithreaded applications present an increased risk of race conditions and deadlocks. Traditionally, techniques such as mutex locks, semaphores, and monitors have been used to address these issues, but as the number of processing cores increases, it becomes increasingly difficult to design multithreaded applications that are free from race conditions and deadlocks.

Transactional Memory

The concept of **transactional memory** originated in database theory, for example, yet it provides a strategy for process synchronization. A **memory transaction** is a sequence of memory read–write operations that are atomic. If all operations in a transaction are completed, the memory transaction is committed. Otherwise, the operations must be aborted and rolled back.

Consider an example. Suppose we have a function update() that modifies shared data. Traditionally, this function would be written using mutex locks (or semaphores) such as the following:

```
void update ()
{
    acquire();
  /* modify shared data */
    release();
}
```

Using synchronization mechanisms such as mutex locks and semaphores involves many potential problems, including deadlock. Additionally, as the number of threads increases, traditional locking scales less well.

As an alternative to traditional locking methods, new features that take advantage of transactional memory can be added to a programming language. In our example, suppose we add the construct atomic{S}, which ensures that the operations in S execute as a transaction. This allows us to rewrite the update() function as follows:

```
void update ()
{
        atomic {
        /* modify shared data */
        }
}
```

The advantage of using such a mechanism rather than locks is that the transactional memory system—not the developer—is responsible for guaranteeing atomicity.

Transactional memory can be implemented in either software or hardware. **Software transactional memory** (STM), as the name suggests, implements transactional memory exclusively in software—no special hardware is needed.

Hardware transactional memory (HTM) uses hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving shared data residing in separate processors' caches.

OpenMP

OpenMP includes a set of compiler directives and an API. Any code following the compiler directive #pragma omp parallel is identified as a parallel region and is performed by a number of threads equal to the number of processing cores in the system.

The advantage of OpenMP (and similar tools) is that thread creation and management are handled by the OpenMP library and are not the responsibility of application developers.

Along with its #pragma omp parallel compiler directive, OpenMP provides the compiler directive #pragma omp critical, which specifies the code region following the directive as a critical section in which only one thread may be active at a time. In this way, OpenMP provides support for ensuring that threads do not generate race conditions.

As an example of the use of the critical-section compiler directive, first assume that the shared variable counter can be modified in the update() function as follows:

```
void update(int value)
{
counter += value;
}
```

If the update() function can be part of—or invoked from—a parallel region, a race condition is possible on the variable counter.

The critical-section compiler directive can be used to remedy this race condition and is coded as follows:

```
void update(int value)
{
      #pragma omp critical
    {
     counter += value;
    }
}
```

The critical-section compiler directive behaves much like a binary semaphore or mutex lock, ensuring that only one thread at a time is active in the critical section. If a thread attempts to enter a critical section when another thread is currently active in that section (that is, ***owns*** the section), the calling thread is blocked until the owner thread exits.

An advantage of using the critical-section compiler directive in OpenMP is that it is generally considered easier to use than standard mutex locks.

Functional Programming Languages

Most well-known programming languages—such as C, C++, Java, and C#— are known as **imperative** (or **procedural**) languages. Imperative languages are used for implementing algorithms that are state-based. In these languages, the flow of the algorithm is crucial to its correct operation, and state is represented with variables and other data structures.

Functional programming languages, which follow a programming paradigm much different from that offered by imperative languages. The fundamental difference between imperative and functional languages is that functional languages do not maintain state.

That is, once a variable has been defined and assigned a value, its value is immutable—it cannot change. Because functional languages disallow mutable state, they need not be concerned with issues such as race conditions and deadlocks. Several functional languages are presently in use, two of them here: Erlang and Scala.

CPU Scheduling: Scheduling Criteria, Scheduling Algorithms, Thread Scheduling, Multi-Processor Scheduling, Real-Time CPU Scheduling, Algorithmic Evaluation

Basic Concepts

- Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)
- In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

CPU-I/O Burst Cycle

- Almost all processes alternate between two states in a continuing ***cycle***, as shown in Figure below ***:***
    - A CPU burst of performing calculations, and
    - An I/O burst, waiting for data transfer in or out of the system.

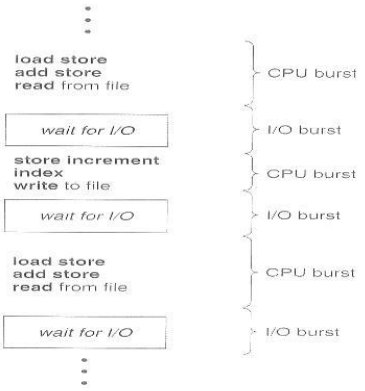


Figure 5.1 Alternating sequence of CPU and I/O bursts.

- CPU bursts vary from process to process, and from program to program, but an extensive study shows frequency patterns similar to that shown in Figure

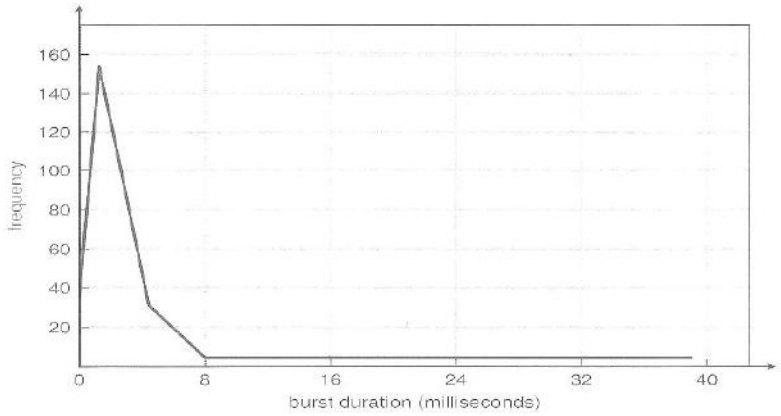


Figure 5.2 Histogram of CPU-burst durations.

CPU Scheduler

- Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue.

Preemptive Scheduling

- CPU scheduling decisions take place under one of four conditions:
    1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
    2. When a process switches from the running state to the ready state, for example in response to an interrupt.
    3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
    4. When a process terminates.
- For conditions 1 and 4 there is no choice - A new process must be selected.
- For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- If scheduling takes place only under conditions 1 and 4, the system is said to be ***non-preemptive***, or ***cooperative***. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be ***preemptive.***
- Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.
- Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures.
- Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs.
- Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.

Dispatcher

- The **dispatcher** is the module that gives control of the CPU to the process selected by the scheduler. This function involves:
    - Switching context.
    - Switching to user mode.
    - Jumping to the proper location in the newly loaded program.
- The dispatcher needs to be as fast as possible, as it is run on every context switch. The time consumed by the dispatcher is known as **dispatch latency**.

Scheduling Criteria

- There are several different criteria to consider when trying to select the "best" scheduling algorithm for a particular situation and environment, including:

- **CPU utilization** - Ideally the CPU would be busy 100% of the time, so as to waste 0 CPU cycles. On a real system CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)
- **Throughput** - Number of processes completed per unit time. May range from 10 / second to 1 / hour depending on the specific processes.
- **Turnaround time** - Time required for a particular process to complete, from submission time to completion. (Wall clock time.)
- **Waiting time** - How much time processes spend in the ready queue waiting their turn to get on the CPU.
    - (**Load average** - The average number of processes sitting in the ready queue waiting their turn to get into the CPU. Reported in 1-minute, 5-minute, and 15-minute averages by "uptime" and "who".)
- **Response time** - The time taken in an interactive program from the issuance of a command to the ***commence*** of a response to that command.

- In general one wants to optimize the average value of a criteria (Maximize CPU utilization and throughput, and minimize all the others.) However some times one wants to do something different, such as to minimize the maximum response time.

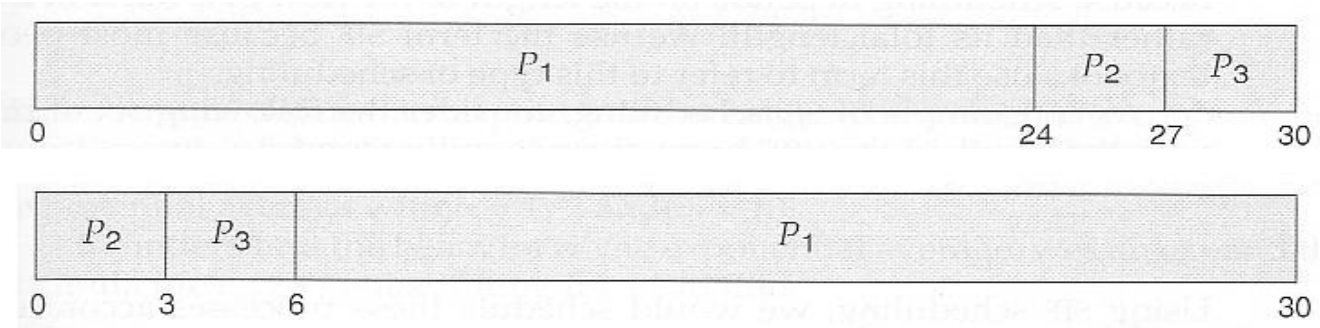
Scheduling Algorithms

The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles.

First-Come First-Serve Scheduling, FCFS

- FCFS is very simple - Just a FIFO queue, like customers waiting in line at the bank or the post office or at a copying machine. Unfortunately, however, FCFS can yield some very long average wait times, particularly if the first process to get there takes a long time. For example, consider the following three processes:

Process	Burst Time
P1	24
P2	3
P3	3

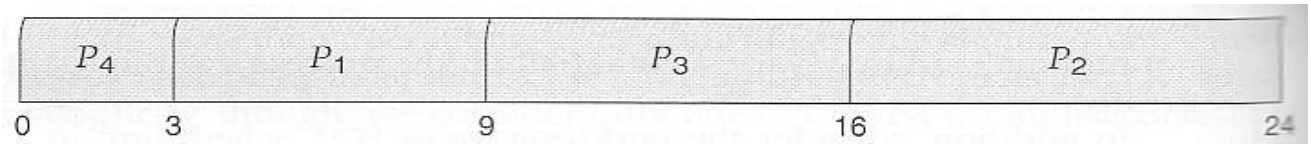


- In the first Gantt chart above, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.
- In the second Gantt chart above, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.
- FCFS can also block the system in a busy dynamic system in another way, known as the ***convoy effect***. When one CPU intensive process blocks the CPU, a number of I/O intensive processes can get backed up behind it, leaving the I/O devices idle. When the CPU hog finally relinquishes the CPU, then the I/O processes pass through the CPU quickly, leaving the CPU idle while everyone queues up for I/O, and then the cycle repeats itself when the CPU intensive process gets back to the ready queue.

Shortest-Job-First Scheduling, SJF

- The idea behind the SJF algorithm is to pick the quickest fastest little job that needs to be done, get it out of the way first, and then pick the next smallest fastest job to do next. (Technically this algorithm picks a process based on the next shortest **CPU burst**, not the overall process time.)
- For example, the Gantt chart below is based upon the following CPU burst times, (and the assumption that all jobs arrive at the same time.)

Process	Burst Time
P1	6
P2	8
P3	7
P4	3



- In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)
- SJF can be proven to be the fastest scheduling algorithm, but it suffers from one important problem: How do you know how long the next CPU burst is going to be?
    - For long-term batch jobs this can be done based upon the limits that users set for their jobs when they submit them, which encourages them to set low limits, but risks their having to re-submit the job if they set the limit too low. However that does not work for short-term CPU scheduling on an interactive system.
    - Another option would be to statistically measure the run time characteristics of jobs, particularly if the same tasks are run repeatedly and predictably. But once again that really isn't a viable option for short term CPU scheduling in the real world.
    - A more practical approach is to *predict* the length of the next burst, based on some historical measurement of recent burst times for this process. One simple, fast, and relatively accurate method is the *exponential average*, which can be defined as follows. (The book uses tau and

t for their variables, but those are hard to distinguish from one another and don't work well in HTML.)

estimate[i + 1] = alpha * burst[i] + (1.0 - alpha) * estimate[i]

- In this scheme the previous estimate contains the history of all previous times, and alpha serves as a weighting factor for the relative importance of recent data versus past history. If alpha is 1.0, then past history is ignored, and we assume the next burst will be the same length as the last burst. If alpha is 0.0, then all measured burst times are ignored, and we just assume a constant burst time. Most commonly alpha is set at 0.5, as illustrated in Figure:

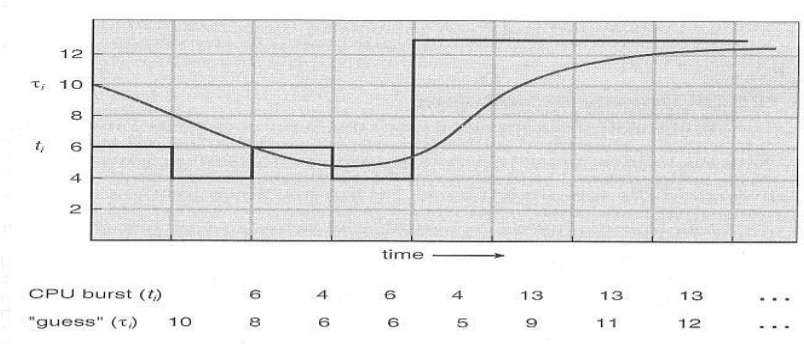
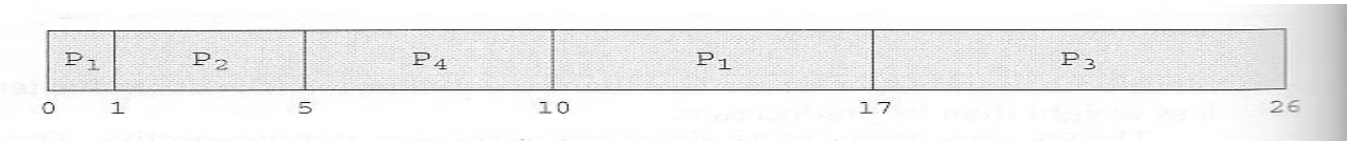


Figure 5.3 Prediction of the length of the next CPU burst.

- SJF can be either preemptive or non-preemptive. Preemption occurs when a new process arrives in the ready queue that has a predicted burst time shorter than the time remaining in the process whose burst is currently on the CPU. Preemptive SJF is sometimes referred to as ***shortest remaining time first scheduling***.
- For example, the following Gantt chart is based upon the following data:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



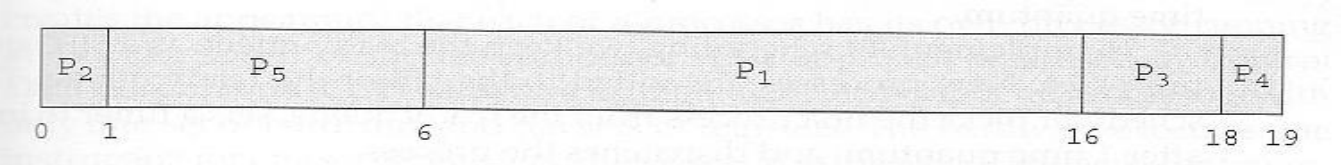
- The average wait time in this case is ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms. (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

Priority Scheduling

- Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
- For example, the following Gantt chart is based upon these process burst times and priorities, and yields an **average waiting time of 8.2 ms:**

Process	Burst Time	Priority
P1	10	3
P2	1	1

P3	2	4
P4	1	5
P5	5	2

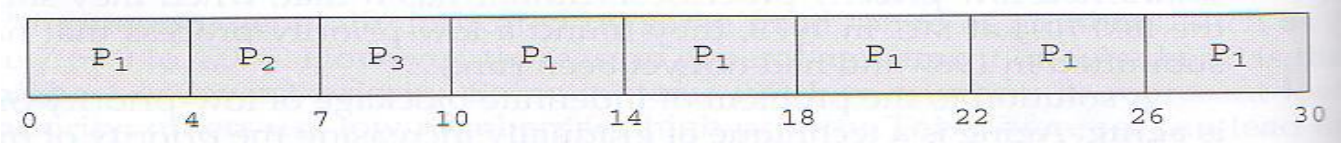


- Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.
- Priority scheduling can be either preemptive or non-preemptive.
- Priority scheduling can suffer from a major problem known as ***indefinite blocking*****, or *starvation***, in which a low-priority task can wait forever because there are always some other jobs around that have higher priority.
    - If this problem is allowed to occur, then processes will either run eventually when the system load lightens (at say 2:00 a.m.), or will eventually get lost when the system is shut down or crashes. (There are rumors of jobs that have been stuck for years.)
    - One common solution to this problem is ***aging*****,** in which priorities of jobs increase the longer they wait. Under this scheme a low-priority job will eventually get its priority raised high enough that it gets run.

Round Robin Scheduling

- Round robin scheduling is similar to FCFS scheduling, except that CPU bursts are assigned with limits called ***time quantum*****.**
- When a process is given the CPU, a timer is set for whatever value has been set for a time quantum.
    - If the process finishes its burst before the time quantum timer expires, then it is swapped out of the CPU just like the normal FCFS algorithm.
    - If the timer goes off first, then the process is swapped out of the CPU and moved to the back end of the ready queue.
- The ready queue is maintained as a circular queue, so when all processes have had a turn, then the scheduler gives the first process another turn, and so on.
- RR scheduling can give the effect of all processors sharing the CPU equally, although the average wait time can be longer than with other scheduling algorithms. In the following example the **average wait time is 5.66 ms.**

Process	**Burst Time**
P1	24
P2	3
P3	3



- The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- BUT, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are (See Figure below). Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.

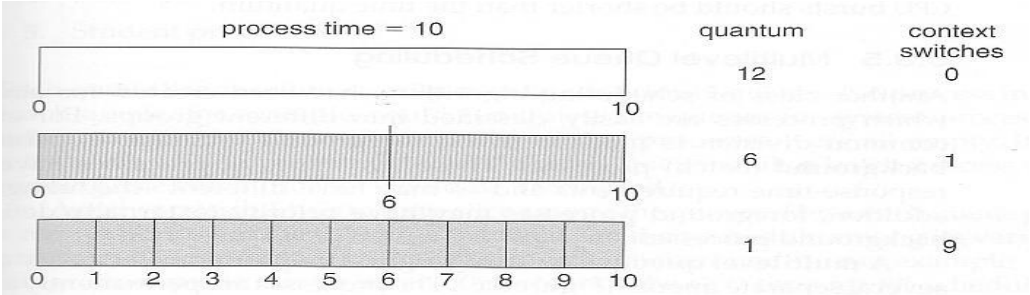


Figure 5.4 The way in which a smaller time quantum increases context switches.

- Turnaround time also varies with quantum time, in a non-apparent manner. In general, turnaround time is minimized if most processes finish their next cpu burst within one time quantum.
- For example, with three processes of 10 ms bursts each, the average turnaround time for 1 ms quantum is 29, and for 10 ms quantum it reduces to 20.
- However, if it is made too large, then RR just degenerates to FCFS. A rule of thumb is that 80% of CPU bursts should be smaller than the time quantum.

Multilevel Queue Scheduling

- When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.

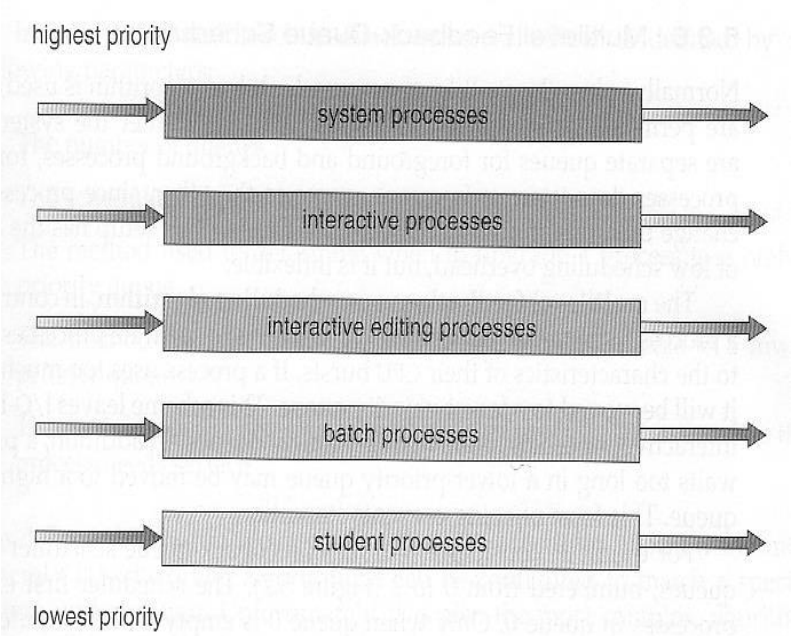


Figure 5.6 Multilevel queue scheduling.

- Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish.

Multilevel Feedback-Queue Scheduling

- Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
    - If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
    - Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.
- Multilevel feedback queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters. Some of the parameters which define one of these systems include:
    - The number of queues.
    - The scheduling algorithm for each queue.
    - The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
    - The method used to determine which queue a process enters initially.

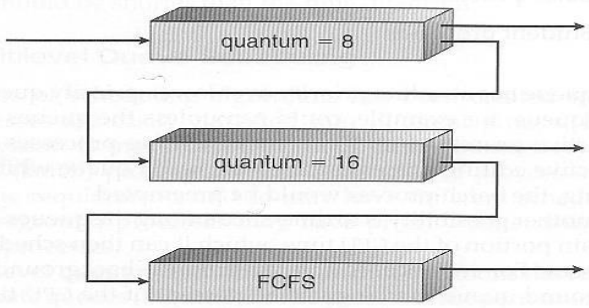


Figure 5.7 Multilevel feedback queues.

Thread Scheduling

To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

Contention Scope

On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **processcontention scope (PCS)**, since competition for the CPU takes place among threads belonging to the same process.

(When we say the thread library *schedules* user threads onto available LWPs, we do not mean that the threads are actually running on a CPU. That would require the operating system to schedule the kernel thread onto a physical CPU.) To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.

Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS. Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library.

Pthread Scheduling

Pthreads identifies the following contention scope values:

- PTHREAD SCOPE PROCESS schedules threads using PCS scheduling.
- PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

On systems implementing the many-to-many model, the PTHREAD SCOPE PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations

The PTHREAD SCOPE SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy. The Pthread IPC provides two functions for getting—and setting—the contention scope policy:

- pthread attr setscope(pthread attr t *attr, int scope)
- pthread attr getscope(pthread attr t *attr, int *scope)

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the thread attr setscope() function is passed either the PTHREAD SCOPE SYSTEM or the PTHREAD SCOPE PROCESS value, indicating how the contention scope is to be set. In the case of pthread attr getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns a nonzero value.

Multiple-Processor Scheduling

If multiple CPUs are available, **load sharing** becomes possible—but scheduling problems become correspondingly more complex.

Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code. This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing.

A second approach uses **symmetric multiprocessing** (**SMP**), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully. We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue. Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X.

Processor Affinity

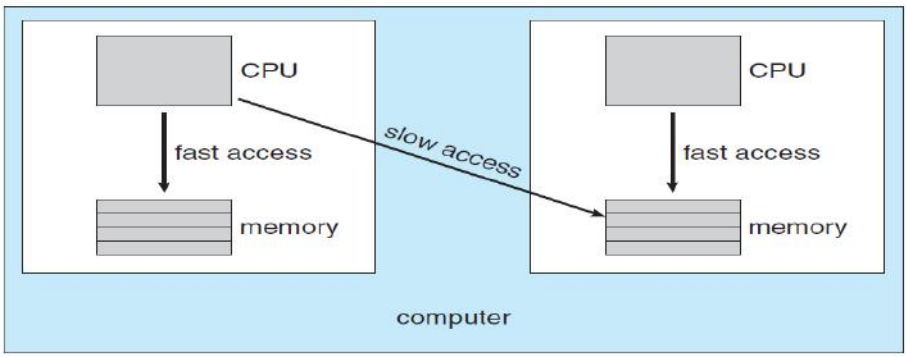
Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated.

Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**—that is, a process has an affinity for the processor on which it is currently running.

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinity**. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors.

Some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it may run. Many systems provide both soft and hard affinity. For example, Linux implements soft affinity, but it also provides the sched_setaffinity() system call, which supports hard affinity.

The main-memory architecture of a system can affect processor affinity issues. Figure illustrates an architecture featuring non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory than to other parts.



Load Balancing

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

There are two general approaches to load balancing: **push migration** and **pull migration**. With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor.

Multicore Processors

A recent practice in computer hardware has been to place multiple processor cores on the same physical chip,
resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor. SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip.

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, may occur for various reasons, such as a cache miss (accessing data that are not in cache memory).

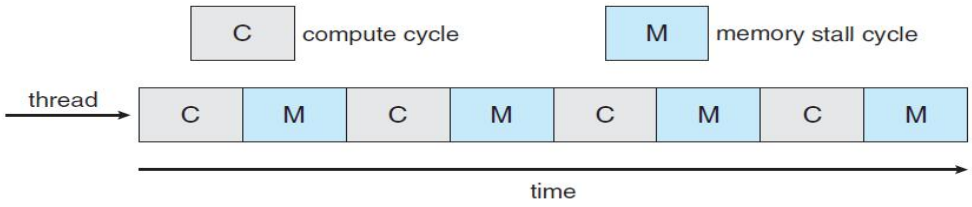


Figure 6.10 Memory stall.

Figure illustrates a dual-threaded processor core on which the execution of thread 0 and the execution of thread 1 are interleaved.

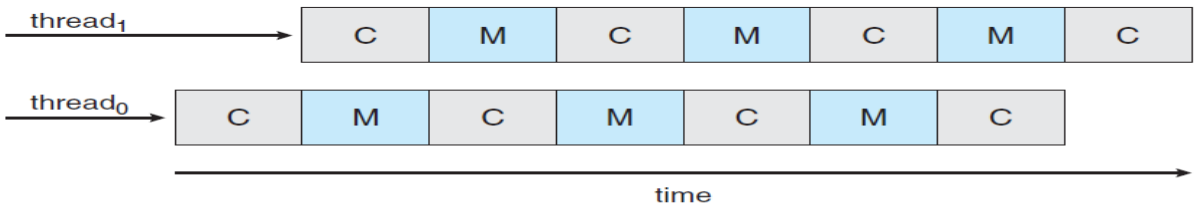


Figure 6.11 Multithreaded multicore system.

Thus, on a dual-threaded, dual-core system, four logical processors are presented to the operating system. The UltraSPARC T3 CPU has sixteen cores per chip and eight hardware threads per core. From the perspective of the operating system, there appear to be 128 logical processors.

In general, there are two ways to multithread a processing core: **coarse grained** and **fine-grained** multithreading. With coarse-grained multithreading, a thread executes on a processor until a long-latency event such as a memory stall occurs. Fine-grained (or interleaved) multithreading switches between threads at a much finer level of granularity—typically at the boundary of an instruction cycle.

Real-Time CPU Scheduling

Soft real-time systems provide no guarantee as to when a critical real-time process will be scheduled. They guarantee only that the process will be given preference over noncritical processes. **Hard real-time systems** have stricter requirements. A task must be serviced by its deadline;

Minimizing Latency

Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure).

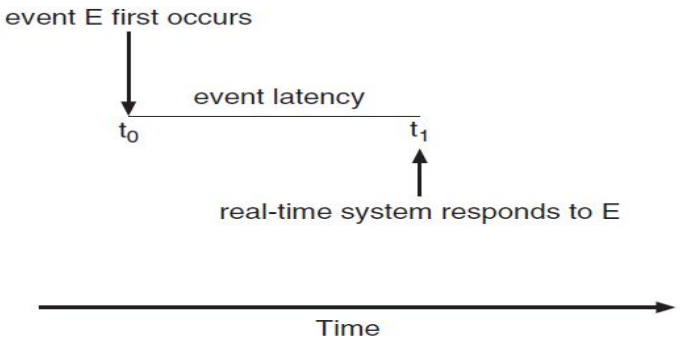


Figure 6.12 Event latency.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency
2. Dispatch latency

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.

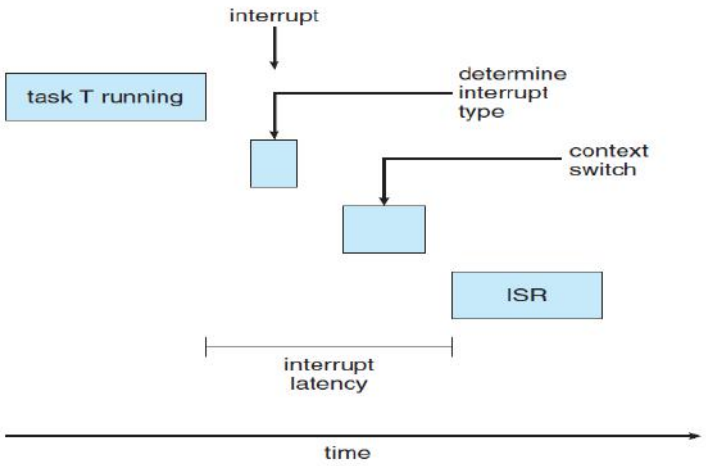


Figure 6.13 Interrupt latency.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**.

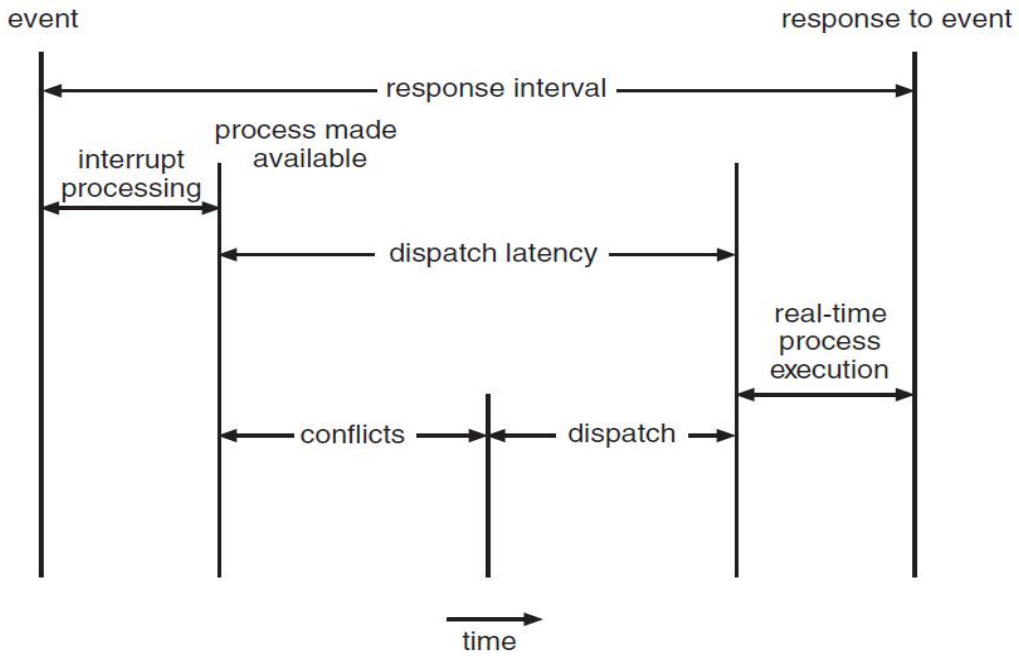


Figure 6.14 Dispatch latency.

In the above Figure, diagram the makeup of dispatch latency. The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes of resources needed by a high-priority process

Priority-Based Scheduling

Priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Note that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features.

Characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Once a periodic process has acquired the CPU, it has a fixed processing time t, a deadline d by which it must be serviced by the CPU, and a period p. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure illustrates the execution of a periodic process over time.

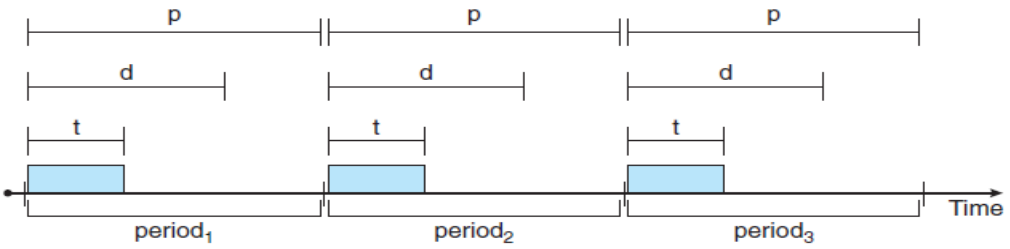


Figure 6.15 Periodic task.

A process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler does one of two things. It either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority.

The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes, $P1$ and $P2$. The periods for $P1$ and $P2$ are 50 and 100, respectively—that is, $p1 = 50$ and $p2 = 100$. The processing times are $t1 = 20$ for $P1$ and $t2 = 35$ for $P2$. The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process Pi as the ratio of its burst to its period—ti/pi —the CPU utilization of $P1$ is $20/50 = 0.40$ and that of $P2$ is $35/100 = 0.35$, for a total CPU utilization of 75 percent.
Suppose we assign $P2$ a higher priority than $P1$. The execution of $P1$ and $P2$ in this situation is shown in Figure. As we can see, $P2$ starts execution first and completes at time 35. At this point, $P1$ starts; it completes its CPU burst at time 55. However, the first deadline for $P1$ was at time 50, so the scheduler has caused $P1$ to miss its deadline.

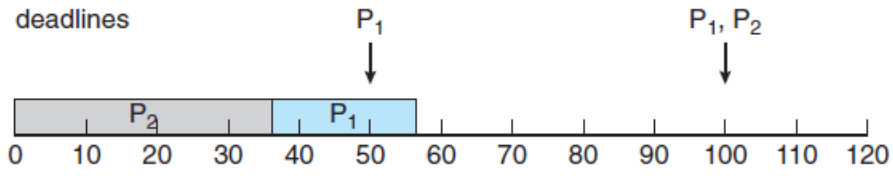


Figure 6.16 Scheduling of tasks when P_2 has a higher priority than P_1.

Now suppose we use rate-monotonic scheduling, in which we assign $P1$ a higher priority than $P2$ because the period of $P1$ is shorter than that of $P2$. The execution of these processes in this situation is shown in Figure.

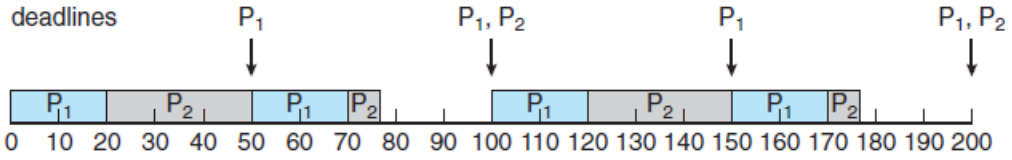


Figure 6.17 Rate-monotonic scheduling.

*P*1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. *P*2 starts running at this point and runs until time 50. At this time, it is preempted by *P*1, although it still has 5 milliseconds remaining in its CPU burst. *P*1 completes its CPU burst at time 70, at which point the scheduler resumes *P*2. *P*2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when *P*1 is scheduled again.

Assume that process *P*1 has a period of *p*1 = 50 and a CPU burst of *t*1 = 25. For *P*2, the corresponding values are *p*2 = 80 and *t*2 = 35. Rate-monotonic scheduling would assign process *P*1 a higher priority, as it has the shorter period. The total CPU utilization of the two processes is (25/50)+(35/80) = 0.94, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. Figure shows the scheduling of processes *P*1 and *P*2.

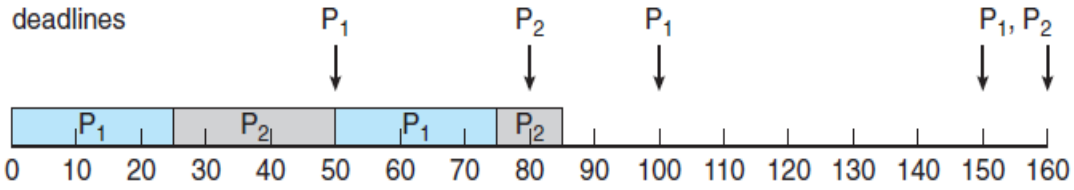


Figure 6.18 Missing deadlines with rate-monotonic scheduling.

Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

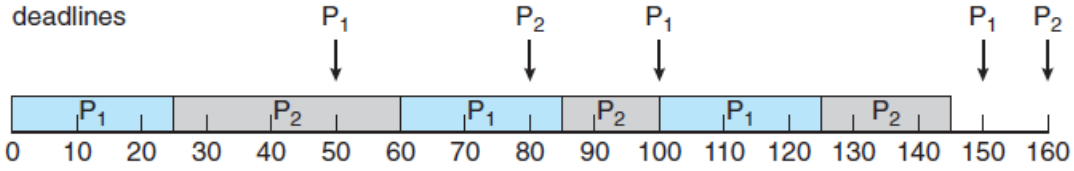


Figure 6.19 Earliest-deadline-first scheduling.

Process *P*1 has the earliest deadline, so its initial priority is higher than that of process *P*2. Process *P*2 begins running at the end of the CPU burst for *P*1. However, whereas rate-monotonic scheduling allows *P*1 to preempt *P*2 at the beginning of its next period at time 50, EDF scheduling allows process *P*2 to continue running. *P*2 now has a higher priority than *P*1 because its next deadline (at time 80) is earlier than that of *P*1 (at time 100). Thus, both *P*1 and *P*2 meet their first deadlines. Process *P*1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. *P*2 begins

running at this point, only to be preempted by *P*1 at the start of its next period at time 100. *P*2 is preempted because *P*1 has an earlier deadline (time 150) than *P*2 (time 160). At time 125, *P*1 completes its CPU burst and *P*2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when *P*1 is scheduled to run once again.

Proportional Share Scheduling

Proportional share schedulers operate by allocating *T* shares among all applications. An application can receive *N* shares of time, thus ensuring that the application will have *N/T* of the total processor time. As an example, assume that a total of *T* = 100 shares is to be divided among three processes, *A*, *B*, and *C*. *A*is assigned 50 shares, *B* is assigned 15 shares, and *C* is assigned 20 shares. This scheme ensures that *A* will have 50 percent of total processor time, *B* will have 15 percent, and *C* will have 20 percent.

Proportional share schedulers must work in conjunction with an admission-control policy to guarantee that an application receives its allocated shares of time. An admission-control policy will admit a client requesting a particular number of shares only if sufficient shares are available. In our current example, we have allocated 50 + 15 + 20 = 85 shares of the total of 100 shares. If a new process *D* requested 30 shares, the admission controller would deny *D* entry into the system.

POSIX Real-Time Scheduling

POSIX defines two scheduling classes for real-time threads:

- SCHED FIFO
- SCHED RR

SCHED FIFO schedules threads according to a first-come, first-served policy using a FIFO queue However, there is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED RR uses a round-robin policy.

It is similar to SCHED FIFO except that it provides time slicing among threads of equal priority. POSIX provides an additional scheduling class—SCHED OTHER—but its implementation is undefined and system specific; it may behave differently on different systems. The POSIX API specifies the following two functions for getting and setting the scheduling policy:

- pthread attr_getsched_policy(pthread attr t *attr, int *policy)
- pthread_attr setsched_policy(pthread attr t *attr, int policy)

Algorithm Evaluation

- The first step in determining which is optimal for a particular operating environment is to determine what criteria are to be used, what goals are to be targeted, and what constraints if any must be applied.
- For example, one might want to "maximize CPU utilization, subject to a maximum response time of 1 second".
- And “Maximize Throughput” subject to a turnaround time is linearly proportional to total execution time.
- Once criteria have been established, then different algorithms can be analyzed and a "best choice" determined. The following sections outline some different methods for determining the "best choice".

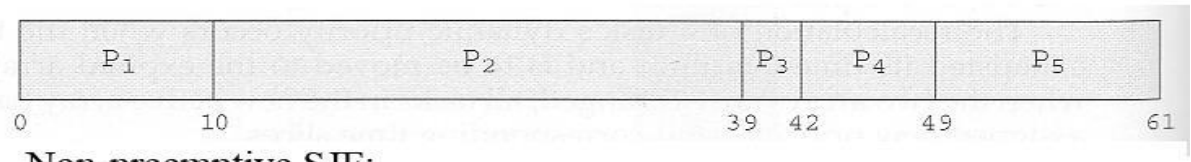
Deterministic Modeling

- It is based on Analytic Evaluation method. If a specific workload is known, then the exact values for major criteria can be fairly easily calculated, and the "best" determined.

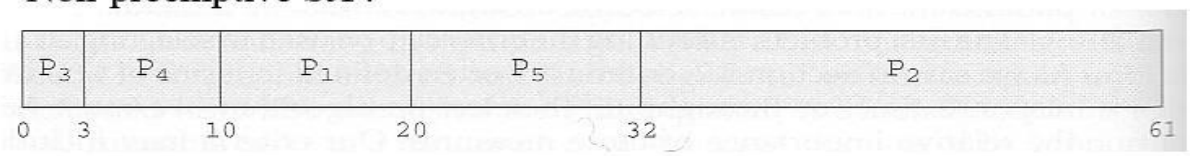
- For example, consider the following workload (with all processes arriving at time 0), and the resulting schedules determined by three different algorithms:

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

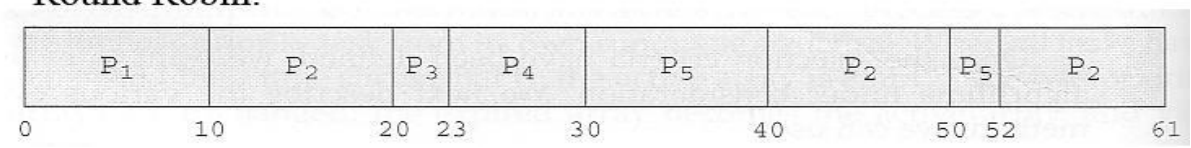
FCFS:



Non-preemptive SJF:



Round Robin:



- The average waiting times for FCFS, SJF, and RR are 28ms, 13ms, and 23ms respectively.
- Deterministic modeling is fast and easy, but it requires specific known input, and the results only apply for that particular set of input.

Queuing Models

- Specific process data is often not available, particularly for future times. However a study of historical performance can often produce statistical descriptions of certain important parameters, such as the rate at which new processes arrive, the ratio of CPU bursts to I/O times, the distribution of CPU burst times and I/O burst times, etc.
- Armed with those probability distributions and some mathematical formulas, it is possible to calculate certain performance characteristics of individual waiting queues.
- For example, **Little's Formula** says that for an average queue length of N, with an average waiting time in the queue of W, and an average arrival of new jobs in the queue of λ, then these three terms can be related by:

$$N = \lambda * W$$

- Queuing models treat the computer as a network of interconnected queues, each of which is described by its probability distribution statistics and formulas such as Little's formula. Unfortunately real systems and modern scheduling algorithms are so complex as to make the mathematics intractable in many cases with real systems.

Simulations

- A Simulation is nothing but creating a Virtual Environment in computers with all the features that of a real time scenario and testing a concept or design or protocol in that environment. If the results are satisfactory than the same concept can be adopted in real time.
- Another approach is to run computer simulations of the different proposed algorithms (and adjustment parameters) under different load conditions, and to analyze the results to determine the "best" choice of operation for a particular load pattern.
- Operating conditions for simulations are often randomly generated using distribution functions similar to those described above.
- A better alternative when possible is to generate ***trace tapes***, by monitoring and logging the performance of a real system under typical expected work loads.
- A compromise is to randomly determine system loads and then save the results into a file, so that all simulations can be run against identical randomly determined system loads.
- Although trace tapes provide more accurate input information, they can be difficult and expensive to collect and store, and their use increases the complexity of the simulations significantly.

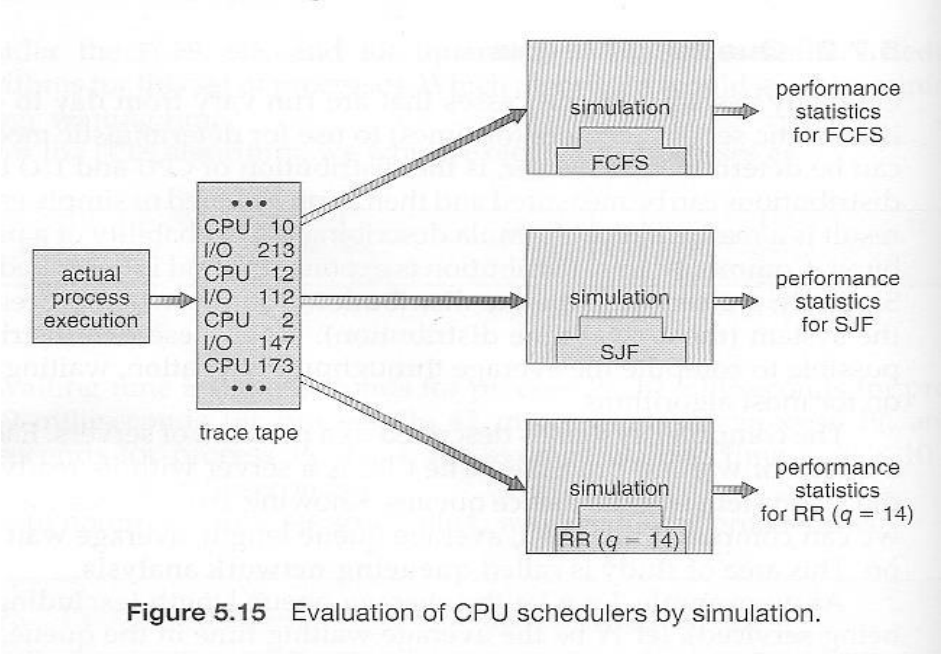


Figure 5.15 Evaluation of CPU schedulers by simulation.

Implementation

- An Implementation is actual realization of an concept. The output of an Implementation is real time.
- The only real way to determine how a proposed scheduling algorithm is going to operate is to implement it on a real system.
- For experimental algorithms and those under development, this can cause difficulties and resistance among users who don't care about developing OSes and are only trying to get their daily work done.
- Even in this case, the measured results may not be definitive, for at least two major reasons: (1) System work loads are not static, but change over time as new programs are installed, new users are added to the system, new hardware becomes available, new work projects get started, and even societal changes. (2) As mentioned above, changing the scheduling system may have an impact on the work load and the ways in which users use the system.
- Most modern systems provide some capability for the system administrator to adjust scheduling parameters, either on the fly or as the result of a reboot or a kernel rebuild.

Important Questions from Previous Papers

1. Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1, P2, P3, P4, and P5 all at time 0.

(a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non preemptive priority (a smaller priority number implies a high priority) and RR (Quantum = 1).

(b) What are the turnaround and waiting processes for each process for each of the scheduling algorithms in part a?

2. What is meant by 'convoy effect' in the context of FCFS scheduling algorithm? Explain with an example.

3. List five services provided by an operating system that are designed to make it more convenient for users to use the computer system. In what cases it would be impossible for user-level programs to provide these services. Explain

4. Describe the actions taken by a thread library to context switch between user-level threads.

5.Explain in detail the analytical method for evaluating scheduling algorithms.

6. Give a note on CPU scheduling in Linux.

7. State the criteria for evaluating CPU scheduling algorithms. Also state whether the criteria is to be optimized for minimal or maximum value.

8. Explain the following scheduling algorithms with examples:
   (a) First-come, First served.
   (b) Shortest-job first

9. a) What is a thread? Illustrate the difference between a traditional single-threaded process and a multithreaded process.
   b) Explain the following scheduling algorithms with examples:
   (i) Shortest-remaining-time-first. (ii) Round Robin.

10. Explain multilevel queue scheduling.

11. **a)** Solve dining-philosophers problem using monitors.
b) Give the definition of Swap ( ) instruction.

12. What is a semaphore? Explain the usage and implementation of semaphores.

13. What is meant by busy waiting? Modify the semaphore operations to overcome the need for busy waiting.

14. **a)** Explain the usage of a monitor.
b) What are the requirements for solving critical-section problem?

15. Give an instance where the producer and consumer routines may not function correctly when executed concurrently.

16. Explain critical section problem.

17. What is race condition? Explain with an example.

18. What is a monitor? Give the syntax of a monitor. Explain how conditional construct provides additional synchronization mechanisms.

19. Explain the mechanisms adopted in Solaris to control access to critical sections.

20. a)What are the various types of errors generated when programmers use semaphores incorrectly to solve the critical section problem?
b) Explain Peterson's solution to critical section problem.

21. With an example explain how two processes accessing two semaphores enter into deadlock state? Suggest a solution.

22. a) What is readers-writers problem? Solve the problem using semaphores.
b) What is priority inversion? What is the solution?

23.a) Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the TestAndSet() instruction. The solution should exhibit minimal busy waiting.
b) Why do Solaries, Linux and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems and not on single-processor systems?

24. Implement counting semaphores with binary semaphores.

UNIT-III

Memory Management: Swapping, Contiguous Memory Allocation, Segmentation, Paging, Structure of Page Table

As a result of CPU scheduling, we can improve both the utilization of the CPU and the speed of the computer's response to its users. To realize this increase in performance, however, we must keep several processes in memory—that is, we must share memory. We discuss various ways to manage memory.

Background:

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

Basic Hardware

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. If the data are not in memory, they must be moved there before the CPU can operate on them.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock. Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick

Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction that it is executing. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access.

For proper system operation we must protect the operating system from access by user processes. This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses.

We can provide this protection by using two registers, usually a base and a limit, as illustrated in below Figure. The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

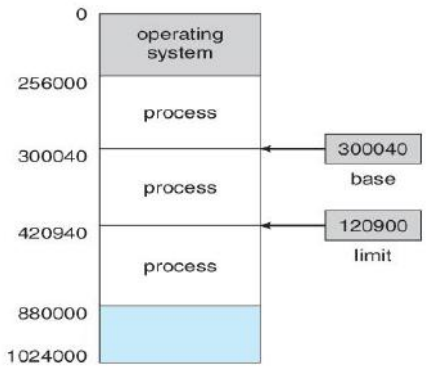


Figure: A base and a limit register define a logical addresss space

Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-

system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error(below figure)

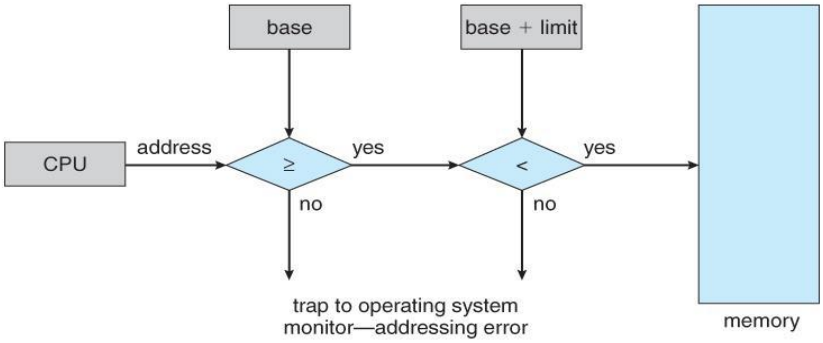


Figure: Hardware address protection with base and limit registers

The operating system, executing in kernel mode, is given unrestricted access to both operating-system memory and users' memory.

Address Binding

- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:
  - **Compile Time** - If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled. DOS .COM programs use compile time binding.
  - **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  - **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time. This requires special hardware, and is the method implemented by most modern OSes.
- Figure shows the various stages of the binding processes and the units involved in each stage:

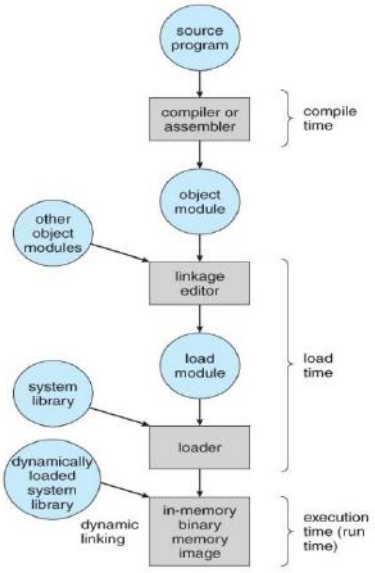


Figure- Multistep processing of a user program

Logical Versus Physical Address Space

- The address generated by the CPU is a ***logical address***, whereas the address actually seen by the memory hardware is a ***physical address***.
- Addresses bound at compile time or load time have identical logical and physical addresses.
- Addresses created at execution time, however, have different logical and physical addresses.
    - In this case the logical address is also known as a ***virtual address***, and the two terms are used interchangeably by our text.
    - The set of all logical addresses used by a program composes the ***logical address space***, and the set of all corresponding physical addresses composes the ***physical address space.***
- The run time mapping of logical to physical addresses is handled by the ***memory-management unit, MMU***.
    - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
    - The base register is now termed a ***relocation register***, whose value is added to every memory request at the hardware level.
- Note that user programs never see physical addresses. User programs work entirely in logical address space, and any memory references or manipulations are done using purely logical addresses. Only when the address gets sent to the physical memory chips is the physical memory address generated.

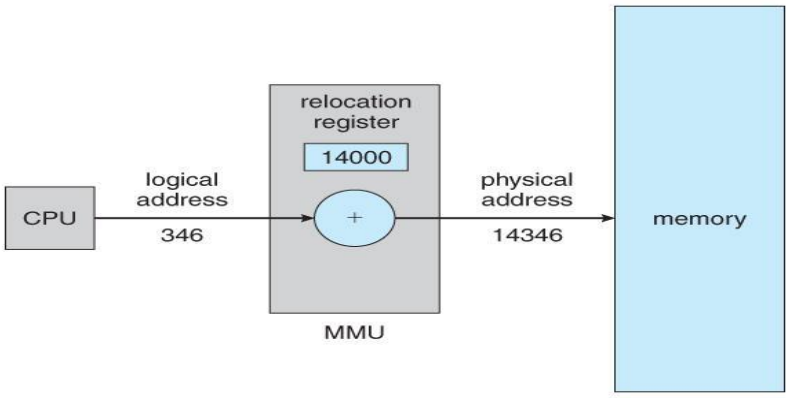


Figure- Dynamic relocation using a relocation register

Dynamic Loading

- Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need never be loaded, reducing total memory usage and generating faster program startup times. The downside is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then then loading it up if it is not already loaded.

Dynamic Linking and Shared Libraries

- With ***static linking*** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With ***dynamic linking***, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

- This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
- We will also learn that if the code section of the library routines is ***reentrant***, (meaning it does not modify the code while it runs, making it safe to re-enter it), then main memory can be saved by loading only one copy of dynamically linked routines into memory and sharing the code amongst all processes that are concurrently using it.
- An added benefit of ***dynamically linked libraries*** (***DLLs***, also known as ***shared libraries*** or ***shared objects*** on UNIX systems) involves easy upgrades and updates.
- In practice, the first time a program calls a DLL routine, the stub will recognize the fact and will replace itself with the actual routine from the DLL library.

Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes who are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***

Standard Swapping

- If compile-time or load-time address binding is used, then processes must be swapped back into the same memory location from which they were swapped out. If execution time binding is used, then the processes can be swapped back into any available location.
- Swapping is a very slow process compared to other operations. For example, if a user process occupied 10 MB and the transfer rate for the backing store were 40 MB per second, then it would take 1/4 second (250 milliseconds) just to do the data transfer. Adding in a latency lag of 8 milliseconds and ignoring head seek time for the moment, and further recognizing that swapping involves moving old data out as well as new data in, the overall transfer time required for this swap is 512 milliseconds, or over half a second.
- To reduce swapping transfer overhead, it is desired to transfer as little information as possible, which requires that the system know how much memory a process ***is*** using, as opposed to how much it ***might*** use. Programmers can help with this by freeing up dynamic memory that they are no longer using.
- It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.

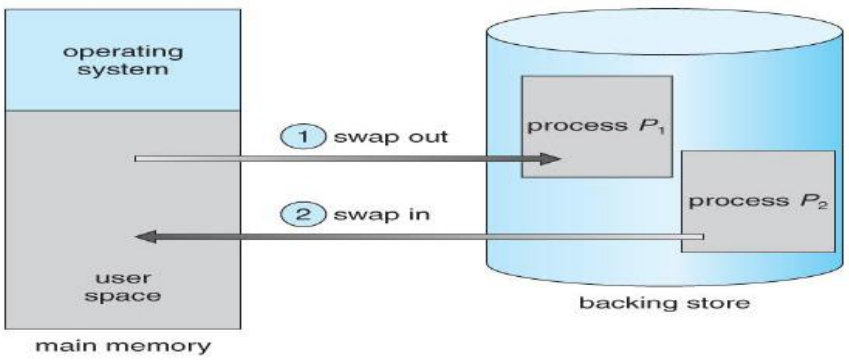


Figure - Swapping of two processes using a disk as a backing store

- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again.

Swapping on Mobile Systems

- Swapping is typically not supported on mobile platforms, for several reasons:
    - Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.
    - Flash memory can only be written to a limited number of times before it becomes unreliable.
    - The bandwidth to flash memory is also lower.
- Apple's IOS asks applications to voluntarily free up memory
    - Read-only data, e.g. code, is simply removed, and reloaded later if needed.
    - Modified data, e.g. the stack, is never removed, but . . .
    - Apps that fail to free up sufficient memory can be removed by the OS
- Android follows a similar strategy.
    - Prior to terminating a process, Android writes its ***application state*** to flash memory for quick restarting.

Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is where the interrupt vectors are located, but on older systems part of the OS was loaded high to make more room in low memory (within the 640K barrier) for user process

Memory Protection

- The system shown in Figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

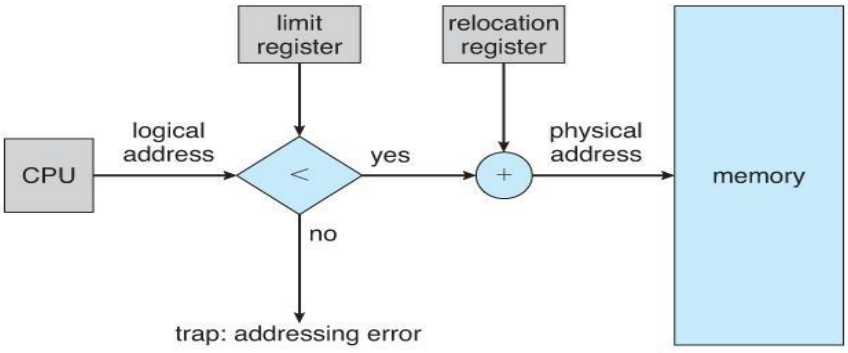


Figure-Hardware support for relocation and limit registers

Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into equal sized partitions, and to assign each process to their own partition.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a suitable size whenever a process needs to be loaded into memory. There are many different

strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:

1. **First fit** - Search the list of holes until one is found that is big enough to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
2. **Best fit** - Allocate the *smallest* hole that is big enough to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
3. **Worst fit** - Allocate the largest hole available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.

- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

Fragmentation

- All the memory allocation strategies suffer from ***external fragmentation***, though first and best fits experience the problems more so than worst fit. External fragmentation means that the available memory is broken up into lots of little pieces, none of which is big enough to satisfy the next memory requirement, although the sum total could.
- The amount of memory lost to fragmentation may vary with algorithm, usage patterns, and some design decisions such as which end of a hole to allocate and which end to save on the free list.
- Statistical analysis of first fit, for example, shows that for N blocks of allocated memory, another 0.5 N will be lost to fragmentation.
- ***Internal fragmentation*** also occurs, with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size. For a random distribution of memory requests, on the average 1/2 block will be wasted per memory request, because on the average the last allocated block will be only half full.
    - Note that the same effect happens with hard drives, and that modern hardware gives us increasingly larger drives and memory at the expense of ever larger block sizes, which translates to more memory lost to internal fragmentation.
    - Some systems use variable size blocks to minimize losses due to internal fragmentation.
- If the programs in memory are relocatable, (using execution-time address binding), then the external fragmentation problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution as we will see in upcoming sections is to allow processes to use non-contiguous blocks of physical memory, with a separate relocation register for each block.

Segmentation

Basic Method

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.

- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure:

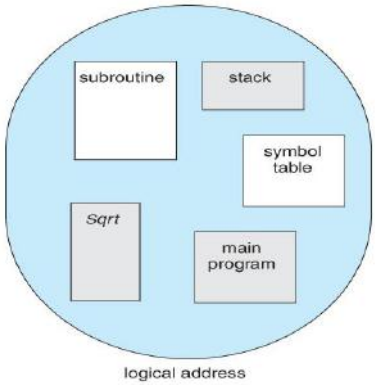


Figure-Programmer's view of a program.

Segmentation Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses, using a system similar to the page tables and relocation base registers.

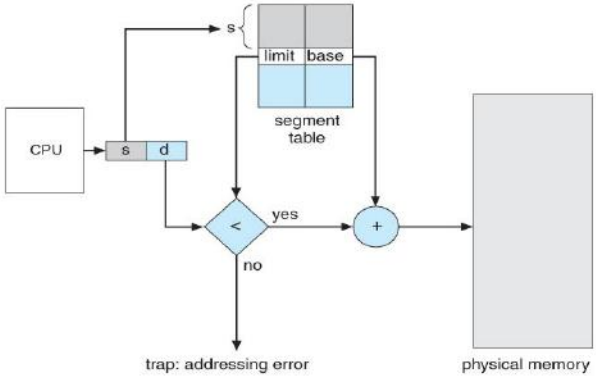


Figure - Segmentation hardware

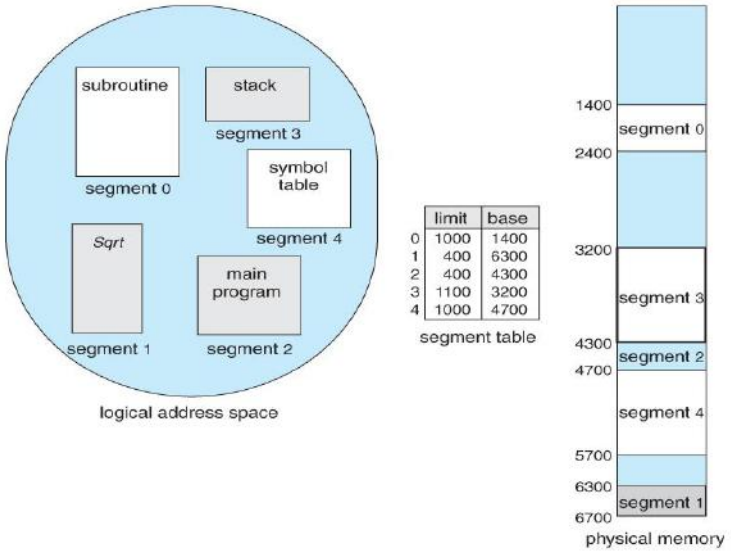


Figure - Example of segmentation

Paging

- Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a programs logical memory space into blocks of the same size called ***pages.***
- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up what frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory:

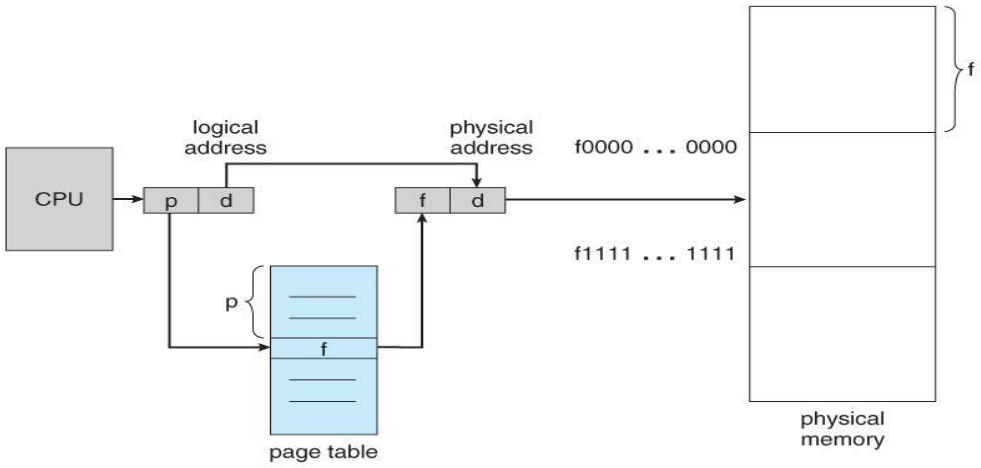


Figure - Paging hardware

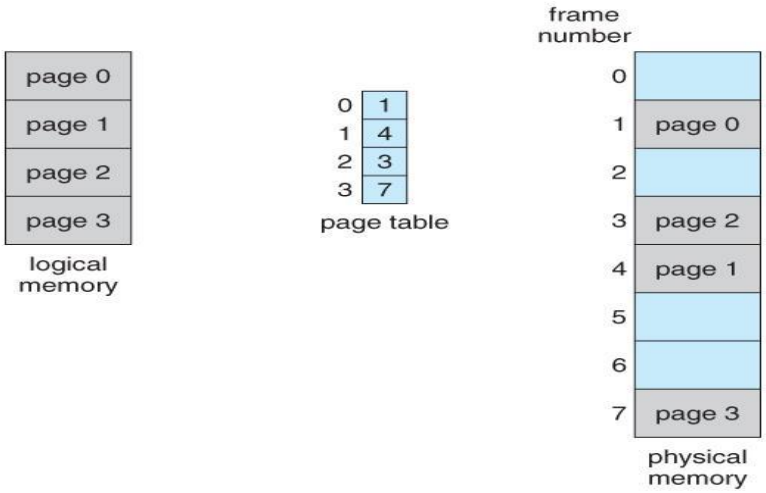


Figure - Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a

single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)

- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory.

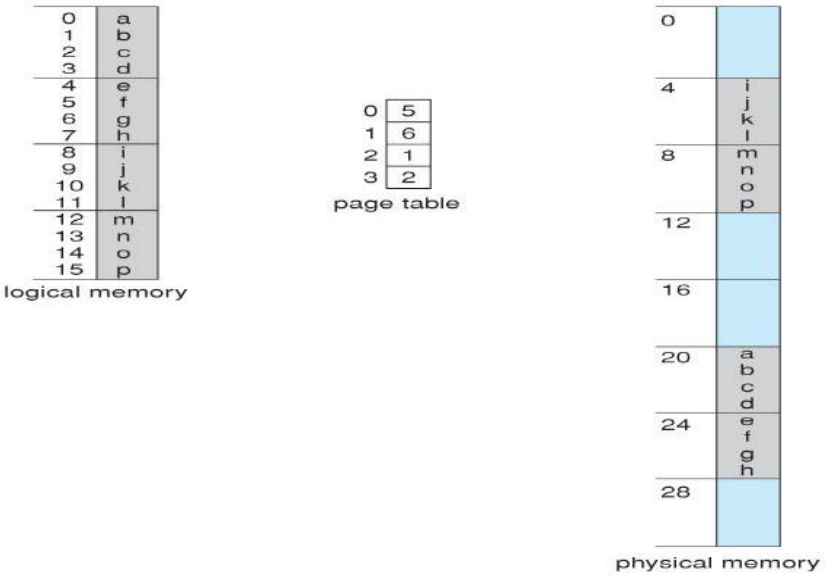


Figure - Paging example for a 32-byte memory with 4-byte pages

- There is **no external fragmentation** with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, **internal fragmentation**. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- Larger page sizes waste more memory, but are more efficient in terms of overhead. Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^{32} physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. (32 + 12 = 44 bits of physical address space.)
- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.

- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU.

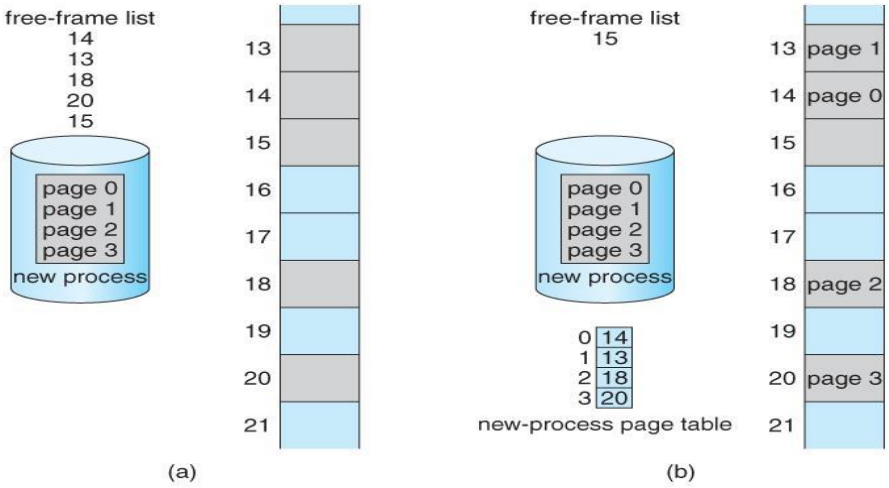


Figure - Free frames (a) before allocation and (b) after allocation

Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use a set of registers for the page table. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)
- An alternate option is to store the page table in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record where in memory the page table is located.
    - Process switching is fast, because only the single register needs to be changed.
    - However memory access just got half as fast, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.
    - The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***

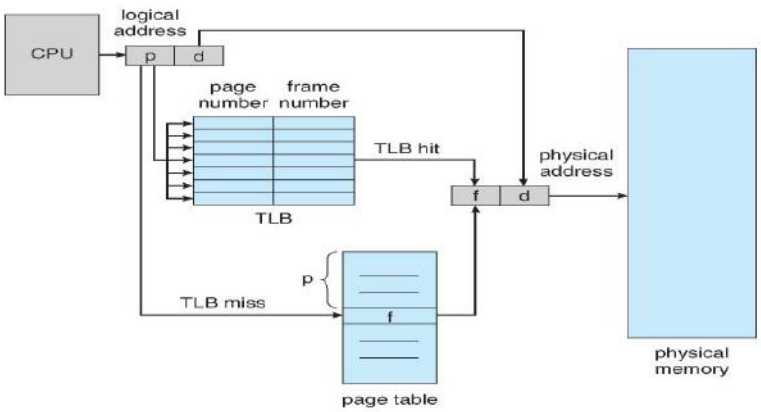


Figure - Paging hardware with TLB

- The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.
- The TLB is very expensive, however, and therefore very small. It is therefore used as a cache device.
- Addresses are first checked against the TLB, and if the info is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
- If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
- Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically these would be kernel frames.
- Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.
- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

  $$0.80 * 120 + 0.20 * 220 = 140 \text{ nanoseconds}$$

  for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

- The ninth edition ignores the 20 nanoseconds required to search the TLB, yielding

  $$0.80 * 100 + 0.20 * 200 = 120 \text{ nanoseconds}$$

  for a 20% slowdown to get the frame number. A 99% hit rate would yield 101 nanoseconds average access time (you should verify this), for a 1% slowdown.

Protection

- The page table can also help to protect processes from accessing memory that they shouldn't, or their own memory in ways that they shouldn't.
- A bit or bits can be added to the page table to classify a page as read-write, read-only, read-write-execute, or some combination of these sorts of things. Then each memory reference can be checked to ensure it is accessing the memory in the appropriate mode.
- Valid / invalid bits can be added to "mask off" entries in the page table that are not in use by the current process, as shown by example in Figure below.
- Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. (Areas of memory in the last page that are not entirely filled by the process, and may contain data left over by whoever used that frame last.)
- Many processes do not use all of the page table available to them, particularly in modern systems with very large potential page tables. Rather than waste memory by creating a full-size page table for every process, some systems use a ***page-table length register, PTLR***, to specify the length of the page table.

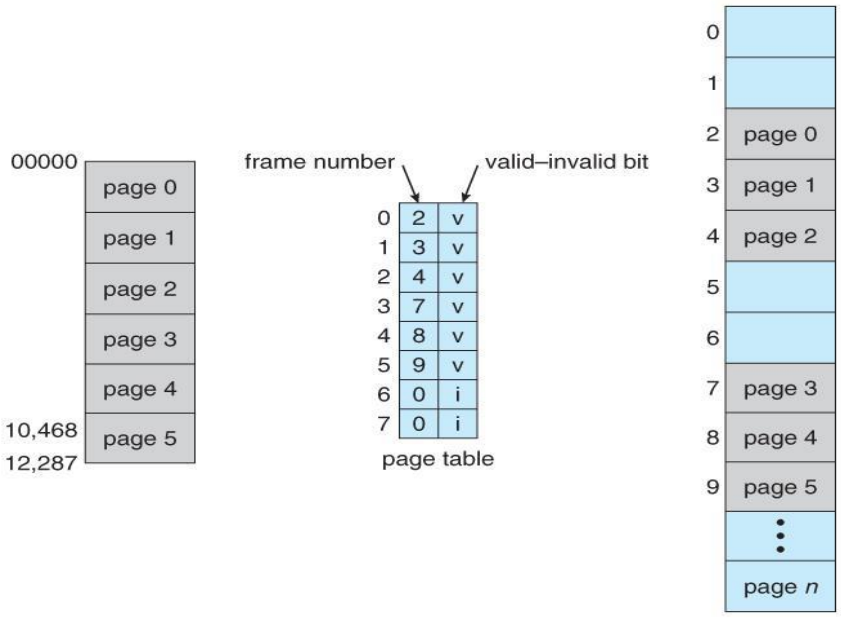


Figure - Valid (v) or invalid (i) bit in page table

Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- If code is ***re-entrant*** or ***pure code*** that means that it does not write to or change the code in any way (it is non self-modifying), and it is therefore safe to re-enter it. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.

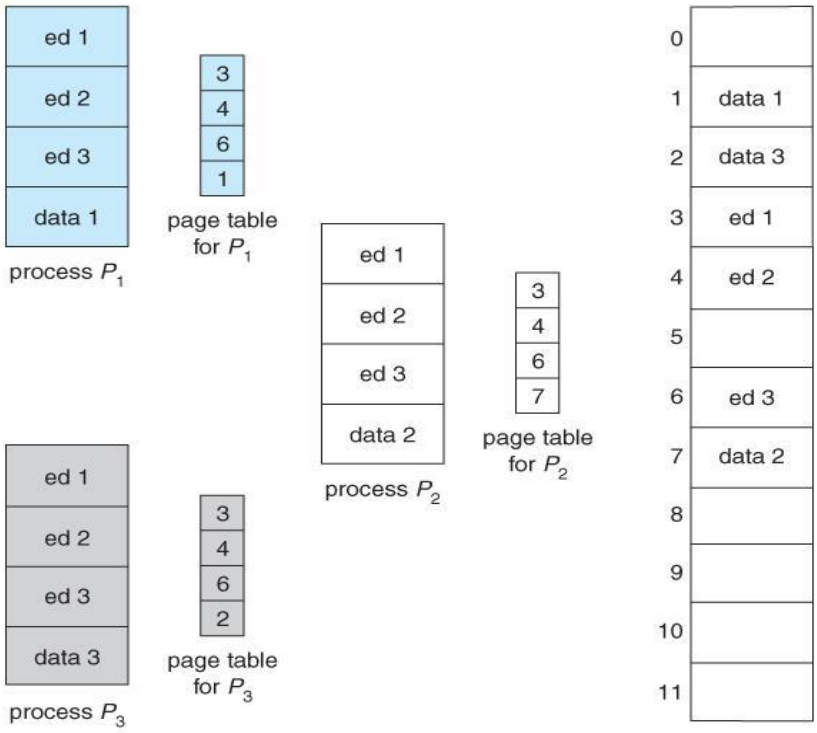


Figure - Sharing of code in a paging environment

Structure of the Page Table

Hierarchical Paging

- Most modern computer systems support logical address spaces of 2^{32} to 2^{64}.
- With a 2^{32} address space and 4K (2^{12}) page sizes, this leave 2^{20} entries in the page table. At 4 bytes per entry, this amounts to a 4 MB page table, which is too large to reasonably keep in contiguous memory. (And to swap in and out of memory with each process switch.) Note that with 4K pages, this would take 1024 pages just to hold the page table!
- One option is to use a two-tier paging system, i.e. to page the page table.
- For example, the 20 bits described above could be broken down into two 10-bit page numbers. The first identifies an entry in the outer page table, which identifies where in memory to find one page of an inner page table. The second 10 bits finds a specific entry in that inner page table, which in turn identifies a particular frame in physical memory. (The remaining 12 bits of the 32 bit logical address are the offset within the 4K frame.)

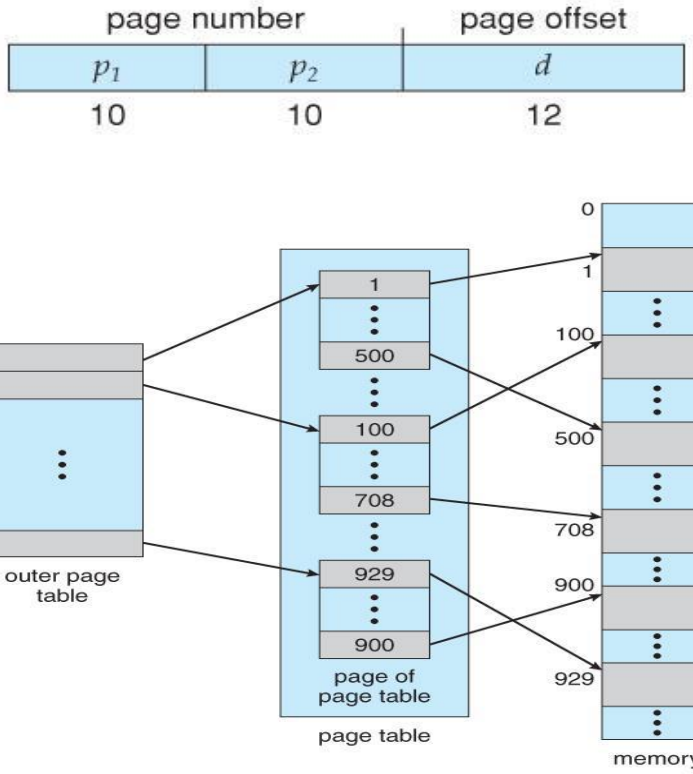


Figure- A two-level page-table scheme

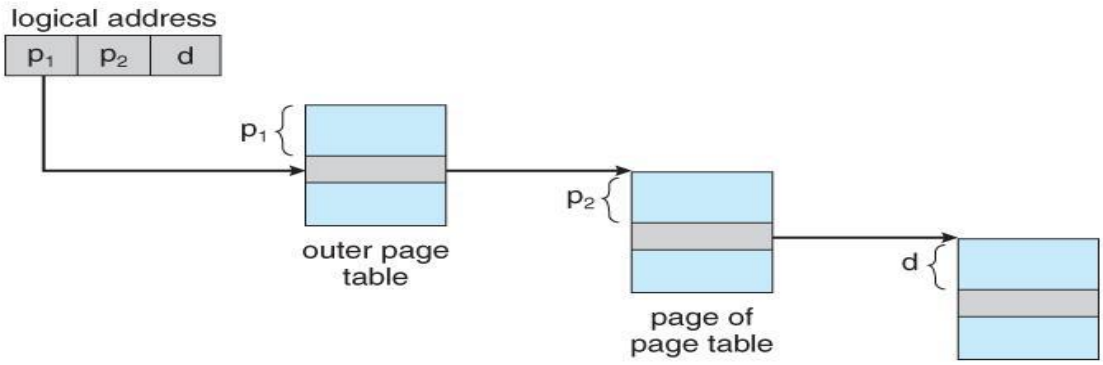
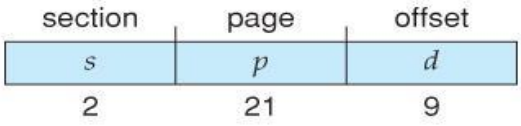
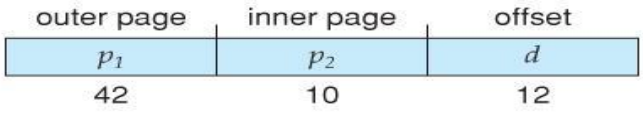


Figure - Address translation for a two-level 32-bit paging architecture

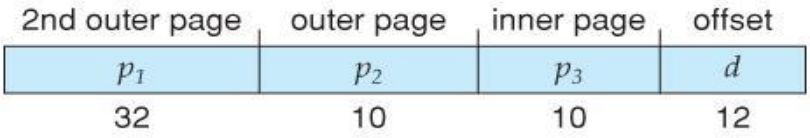
- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form of:



- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging. One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively slow memory access. So some other approach must be used.



64-bits Two-tiered leaves 42 bits in outer table



Going to a fourth level still leaves 32 bits in the outer table.

The 64-bit UltraSPARC would require seven levels of paging—a prohibitive number of memory accesses—to translate each logical address.

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.

A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for **sparse** address spaces, where memory references are noncontiguous and scattered throughout the address space.

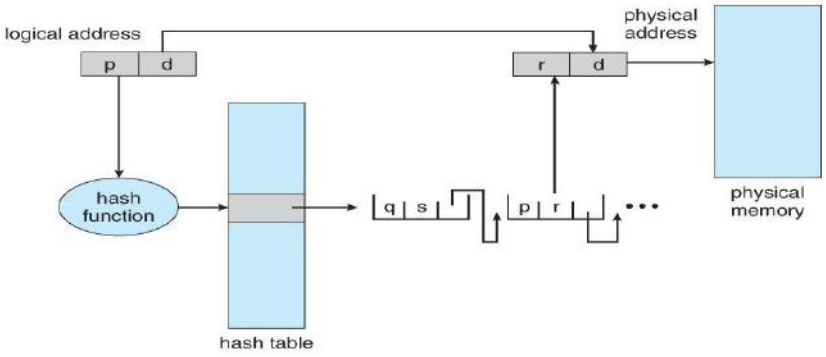


Figure - Hashed page table

Inverted Page Tables

Page 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. Figure shows the operation of an inverted page table.

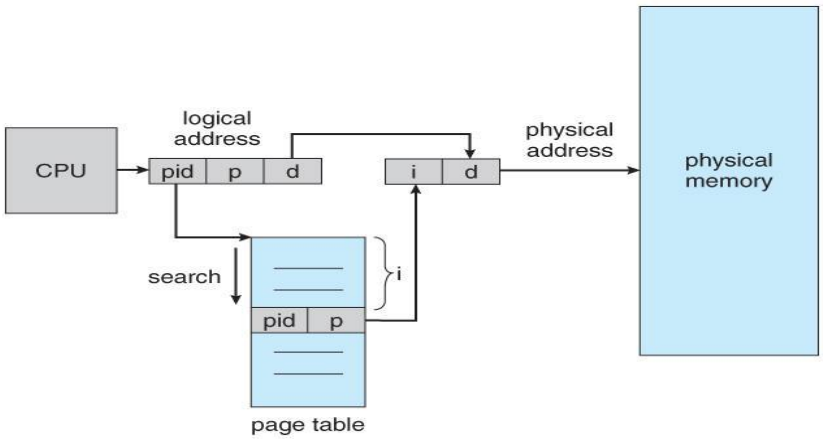


Figure - Inverted page table

Inverted page tables often require that an **address-space identifier** be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory.

Storing the **address-space identifier** ensures that a logical page for a particular process is mapped to the corresponding physical page frame. Examples of systems using inverted page tables include the 64-bit UltraSPARC and PowerPC.

A simplified version of the inverted page table used in the ***IBM RT***. For the IBM RT, each virtual address in the system consists of a triple:

<process-id, page-number, offset>.

Each inverted page-table entry is a pair <process-id, page-number> where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry *i*—then the physical address <*i*, offset> is generated. If no match is found, then an illegal address access has been attempted.

address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long. To alleviate this problem, we use a hash table, to limit the search to one—or at most a few—page-table entries.

Oracle SPARC Solaris

Solaris running on the **SPARC** CPU is a fully 64-bit operating system and as such has to solve the problem of virtual memory without using up all of its physical memory by keeping multiple levels of page tables. Its approach is a bit complex but solves the problem efficiently using hashed page tables.

There are two hash tables—one for the kernel and one for all user processes. Each maps memory addresses from virtual to physical memory. Each hash-table entry represents a contiguous area of mapped virtual memory, which is more efficient than having a separate hash-table entry for each page. Each entry has a base address and a span indicating the number of pages the entry represents.

Virtual-to-physical translation would take too long if each address required searching through a hash table, so the CPU implements a TLB that holds translation table entries (TTEs) for fast hardware lookups. A cache of these TTEs reside in a translation storage buffer (TSB), which includes an entry per recently accessed page. When a virtual address reference occurs, the hardware searches the TLB for a translation. If none is found, the hardware walks through the in-memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup.

This **TLB walk** functionality is found on many modern CPUs. If a match is found in the TSB, the CPU copies the TSB entry into the TLB, and the memory translation completes. If no match is found in the TSB, the kernel is interrupted to search the hash table. The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit. Finally, the interrupt handler returns control to the MMU, which completes the address translation and retrieves the requested byte or word from main memory.

Virtual Memory: Demand Paging, Page Replacement, Allocation of Frames, Thrashing, Memory Mapped Files, Allocating kernel memory

Background

- Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.
- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
    1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
    2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
    3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-)
- The ability to load only the portions of processes that were actually needed (and only *when* they were needed) has several benefits:
    - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
    - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
    - Less I/O is needed for swapping processes in and out of RAM, speeding things up.
- Figure shows the general layout of ***virtual memory***, which can be much larger than physical memory:

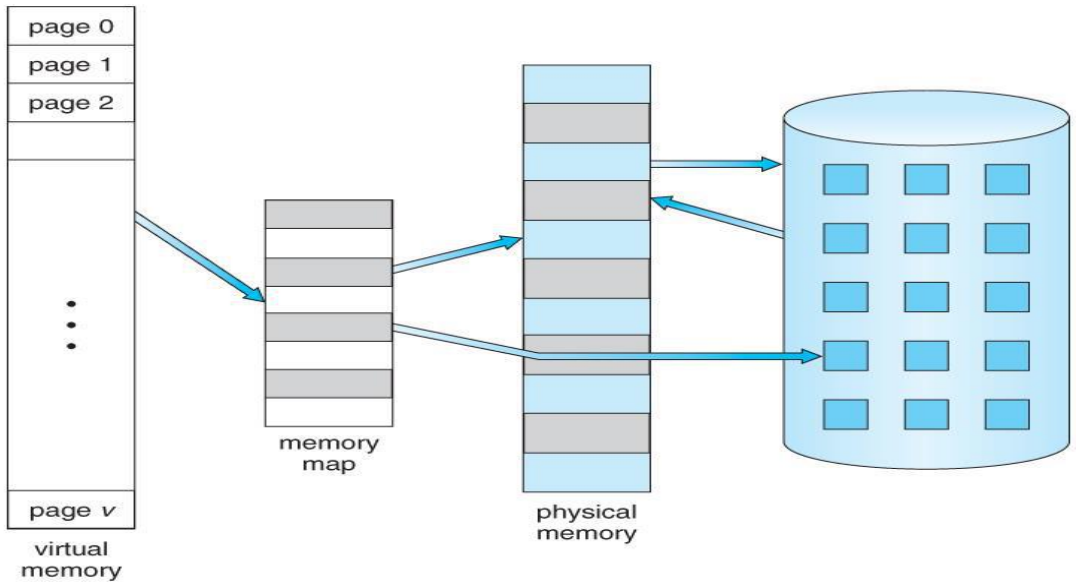


Figure - Diagram showing virtual memory that is larger than physical memory

- The following figure shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure below is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

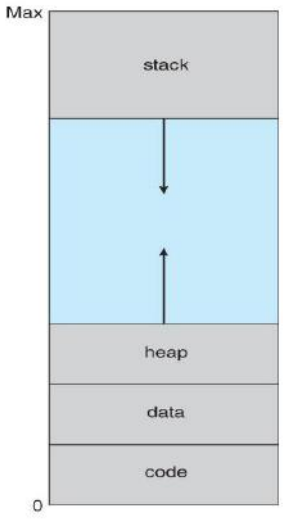


Figure - Virtual address space

- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
  - System libraries can be shared by mapping them into the virtual address space of more than one process.
  - Processes can also share virtual memory by mapping the same block of memory to more than one process.
  - Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

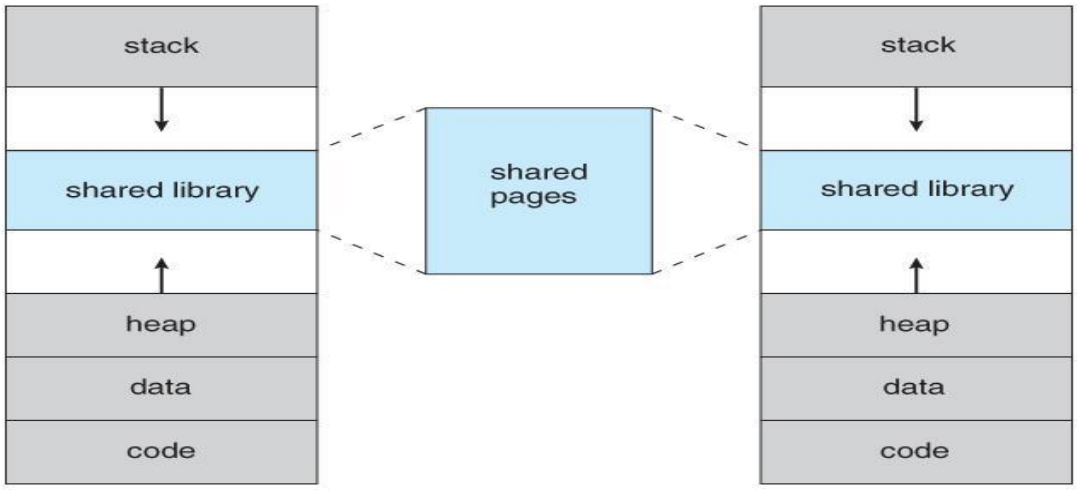


Figure - Shared library using virtual memory

Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a ***lazy swapper***, although a ***pager*** is a more accurate term.

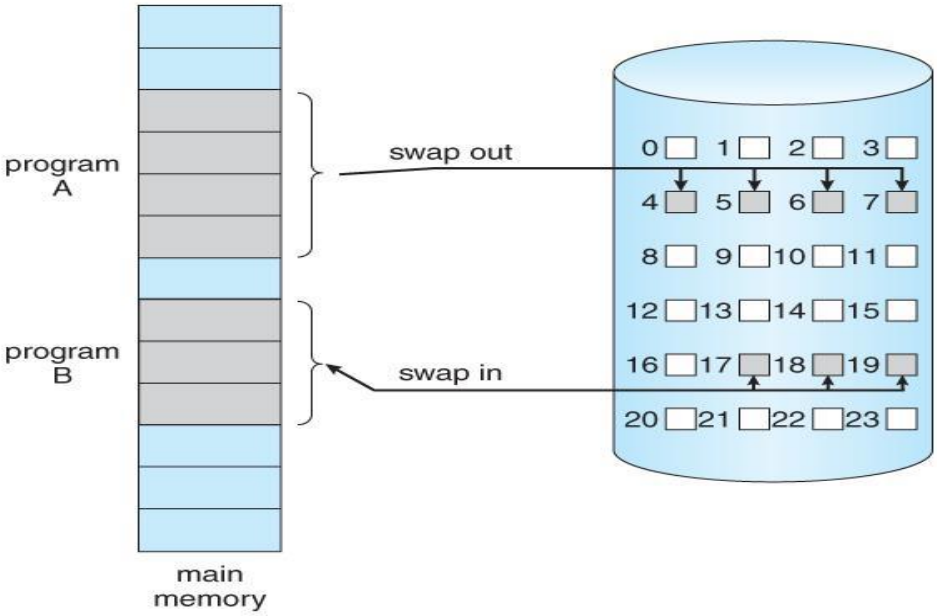


Figure - Transfer of a paged memory to contiguous disk space

Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit.
- If the process only ever accesses pages that are loaded in memory (*memory resident* pages), then the process runs exactly as if all the pages were loaded in to memory.

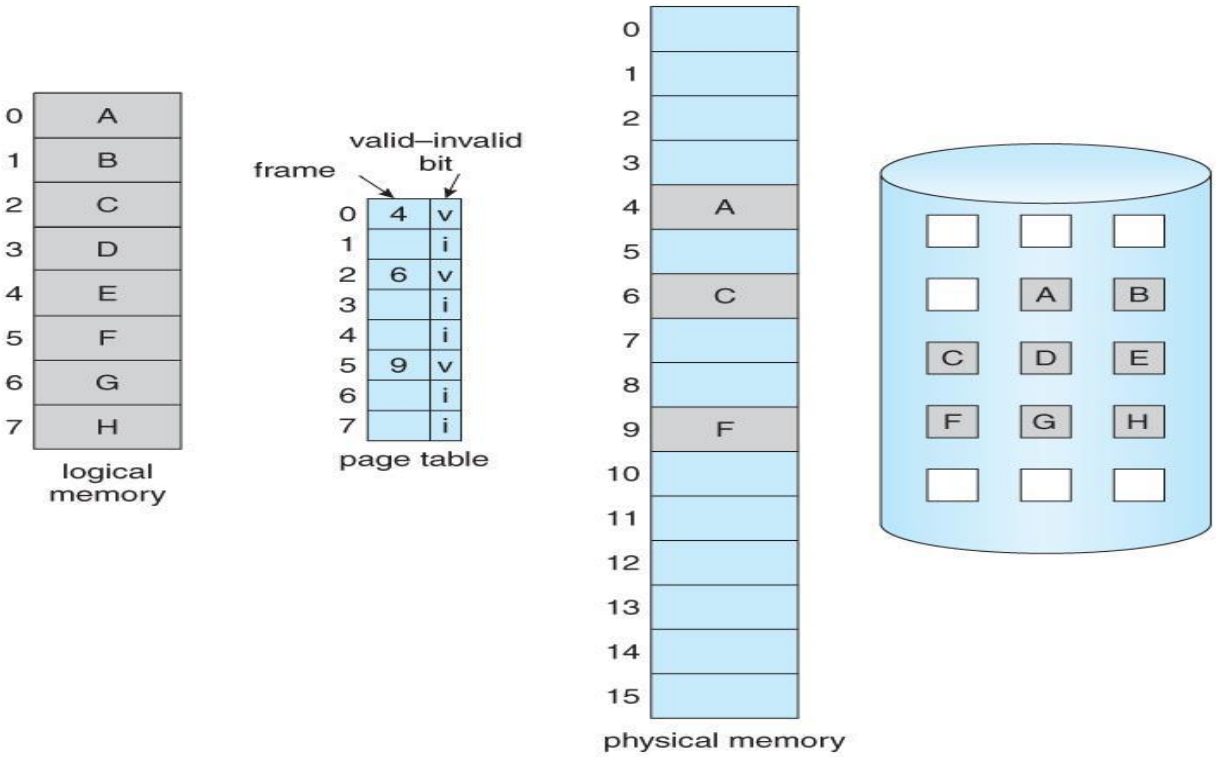


Figure - Page table when some pages are not in main memory.

- On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:
    1. The memory address requested is first checked, to make sure it was a valid memory request.

2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

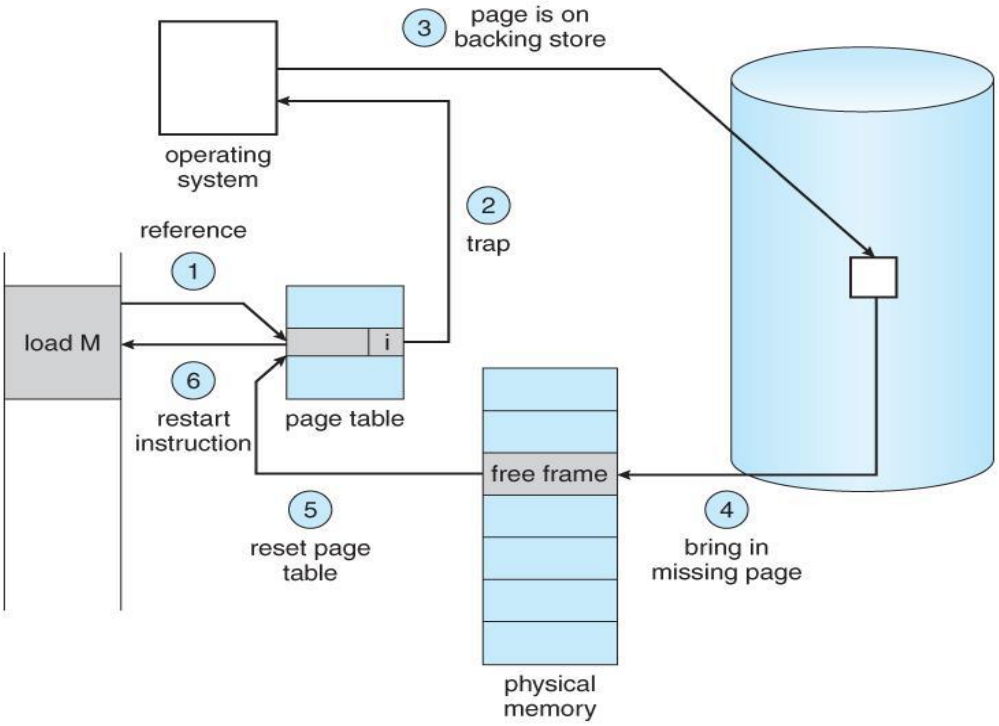


Figure - Steps in handling a page fault

- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- In theory each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory.
- A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory.
- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- There are many steps that occur when servicing a page fault (see book for full details), and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. (8,000,000 nanoseconds, or 40,000 times a normal memory access.) With a ***page fault rate*** of p, (on a scale from 0 to 1), the effective access time is now:

$$(1 - p) * (200) + p * 8000000$$

= 200 + 7,999,800 * p

which ***clearly*** depends heavily on p! Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.
- However memory is also needed for other purposes (such as I/O buffering), and what happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:
    1. Adjust the memory used by I/O buffering, etc., to free up some frames for user processes. The decision of how to allocate memory for I/O versus user processes is a complex one, yielding different policies on different systems.
    2. Put the process requesting more pages into a wait queue until some free frames become available.
    3. Swap some process out of memory completely, freeing up its page frames.
    4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as ***page replacement***, and is the most common solution. There are many different algorithms for page replacement, which is the subject of the remainder of this section.

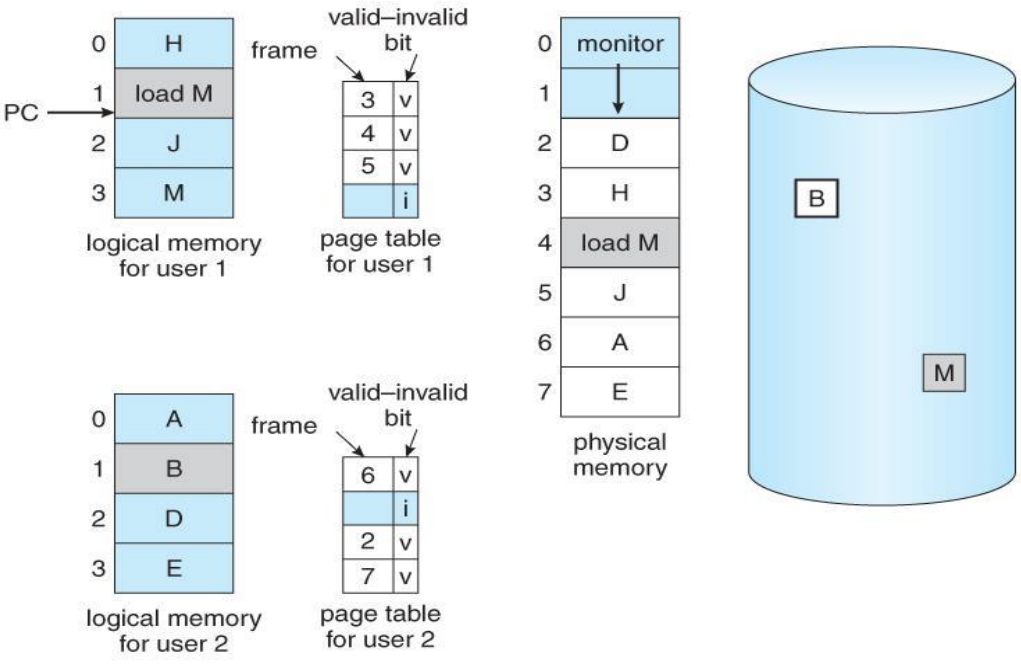


Figure - Ned for page replacement.

Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:

    1. Find the location of the desired page on the disk, either in swap space or in the file system.
    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 an existing frame to be replaced, known as the ***victim frame***.
        c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
    3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
    4. Restart the process that was waiting for this page.

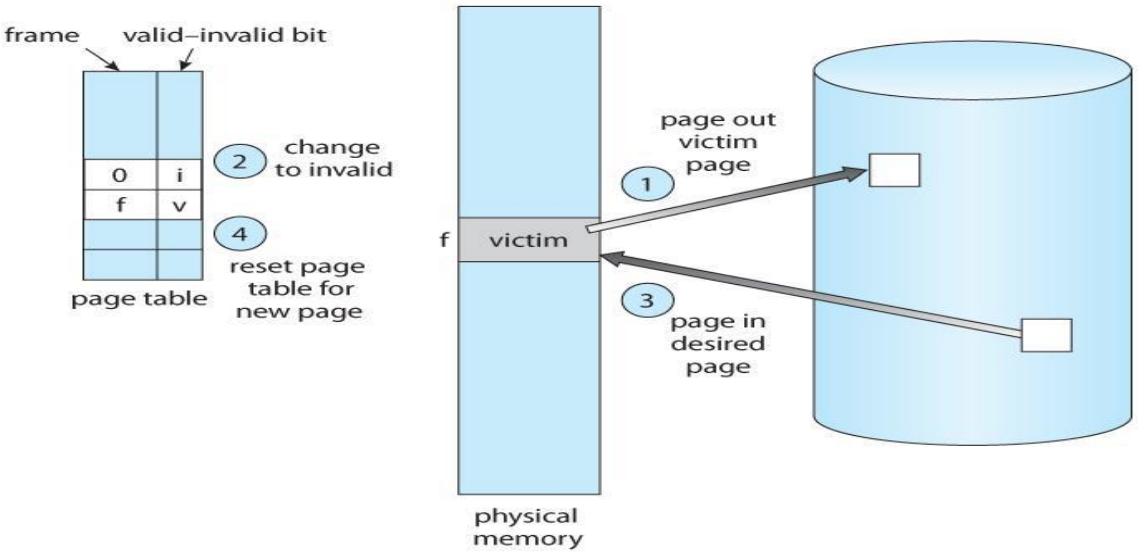


Figure - Page replacement.

- Note that step 3c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a ***modify bit,*** or ***dirty bit*** to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required
- There are two major requirements to implement a successful demand paging system. We must develop a ***frame-allocation algorithm*** and a ***page-replacement algorithm.*** The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available.
- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of memory accesses known as a ***reference string,*** which can be generated in one of (at least) three common ways:

1. Randomly generated, either evenly distributed or with some distribution curve based on observed system behavior. This is the fastest and easiest approach, but may not reflect real performance well, as it ignores locality of reference.
2. Specifically designed sequences. These are useful for illustrating the properties of comparative algorithms in published papers and textbooks, (and also for homework and exam problems. :-))
3. Recorded memory references from a live system. This may be the best approach, but the amount of data collected can be enormous, on the order of a million addresses per second. The volume of collected data can be reduced by making two important observations:
    1. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations.
    2. Successive accesses within the same page can be treated as a single page request, because all requests after the first are guaranteed to be page hits. (Since there are no intervening requests for other pages that could remove this page from the page table.)
    - So for example, if pages were of size 100 bytes, then the sequence of address requests (0100, 0432, 0101, 0612, 0634, 0688, 0132, 0038, 0420) would reduce to page requests (1, 4, 1, 6, 1, 0, 4)

- As the number of available frames increases, the number of page faults should decrease, as shown in Figure :

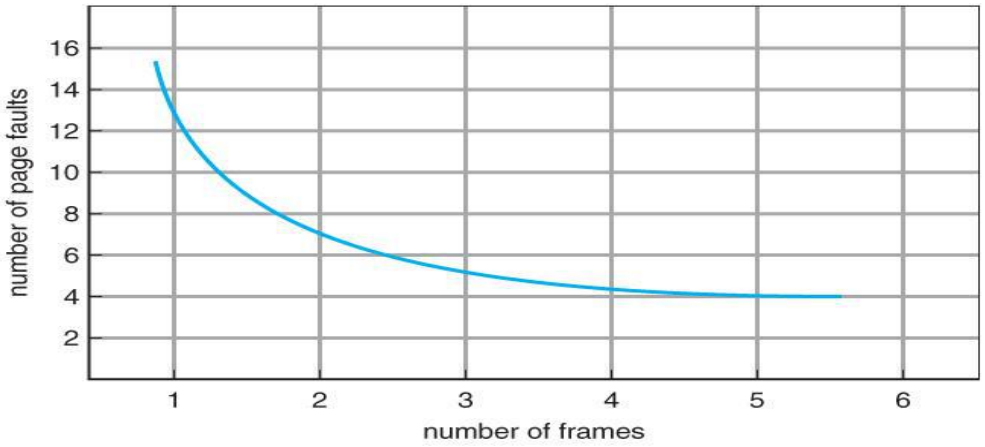


Figure - Graph of page faults versus number of frames.

FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:

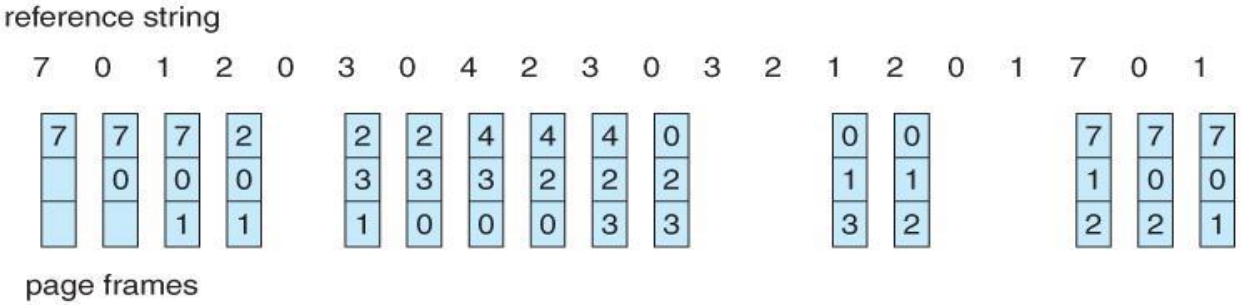


Figure - FIFO page-replacement algorithm.

- Although FIFO is simple and easy, it is not always optimal, or even efficient.
- An interesting effect that can occur with FIFO is ***Belady's anomaly***, in which increasing the number of frames available can actually ***increase*** the number of page faults that occur! Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames. Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:

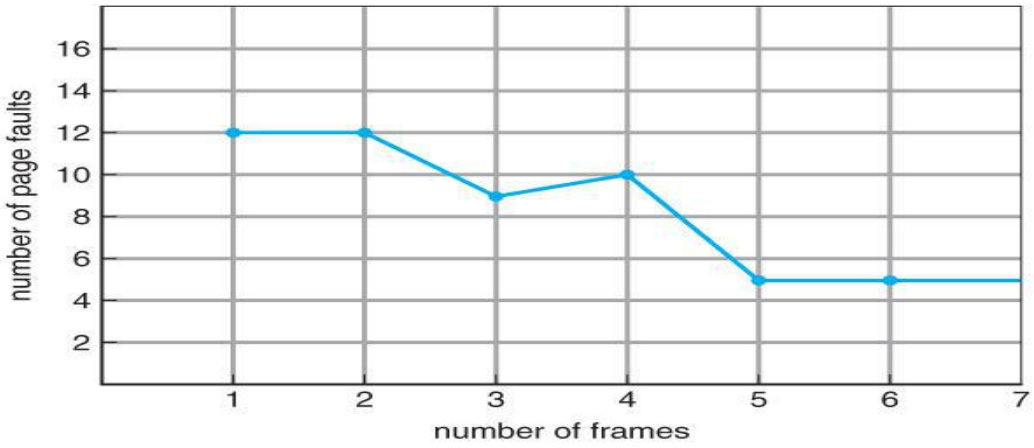


Figure - Page-fault curve for FIFO replacement on a reference string.

Optimal Page Replacement

- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT or MIN.*** This algorithm is simply "Replace the page that will not be used for the longest time in the future."
- For example, the below figure shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9. Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm.
- Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.

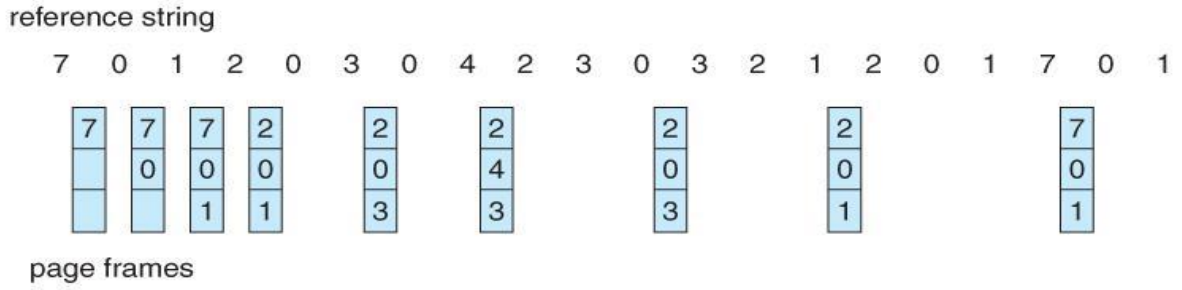


Figure - Optimal page-replacement algorithm

LRU Page Replacement

- The prediction behind ***LRU,*** the ***Least Recently Used,*** algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest ***load*** time, and the latter looks at the oldest ***use*** time.)

- Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)
- Figure illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

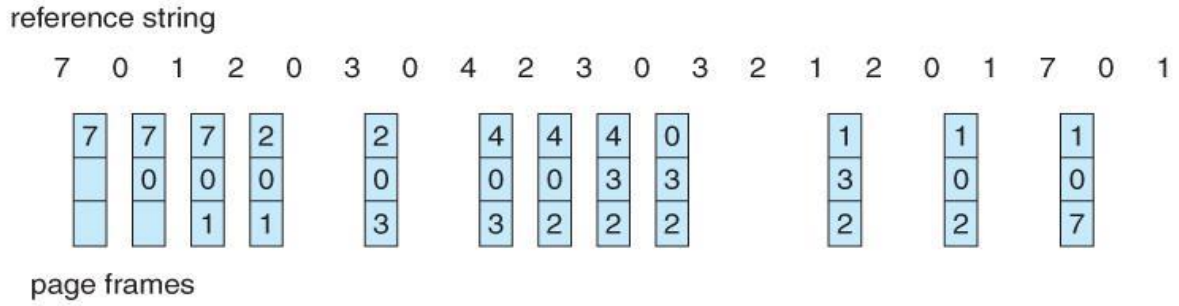


Figure - LRU page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:
  1. **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.
  2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.
- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called ***stack algorithms,*** which can never exhibit Belady's anomaly.

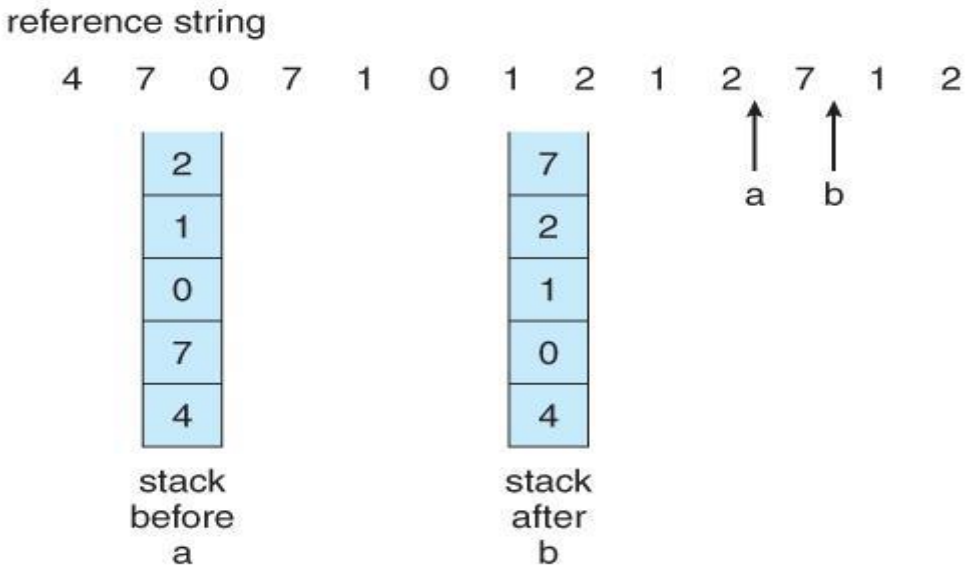


Figure - Use of a stack to record the most recent page references.

LRU-Approximation Page Replacement

- Unfortunately full implementation of LRU requires hardware support, and few systems provide the full hardware support necessary.
- However many systems offer some degree of HW support, enough to approximate LRU fairly well.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to distinguish pages that have been accessed since the last clear from those that have not, but does not provide any finer grain of detail.

Additional-Reference-Bits Algorithm

- Finer grain is possible by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry, which is interpreted as an unsigned int.
    - At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.
    - The high-order (leftmost) bit is then filled in with the current value of the reference bit, and the reference bits are cleared.
    - At any given time, the page with the smallest value for the reference byte is the LRU page.
- Obviously the specific number of bits used and the frequency with which the reference byte is updated are adjustable, and are tuned to give the fastest performance on a given hardware platform.

Second-Chance Algorithm

- The ***second chance algorithm*** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.
    - When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
    - If a page is found with its reference bit not set, then that page is selected as the next victim.
    - If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:
        - The reference bit is cleared, and the FIFO search continues.
        - If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table.
        - If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.

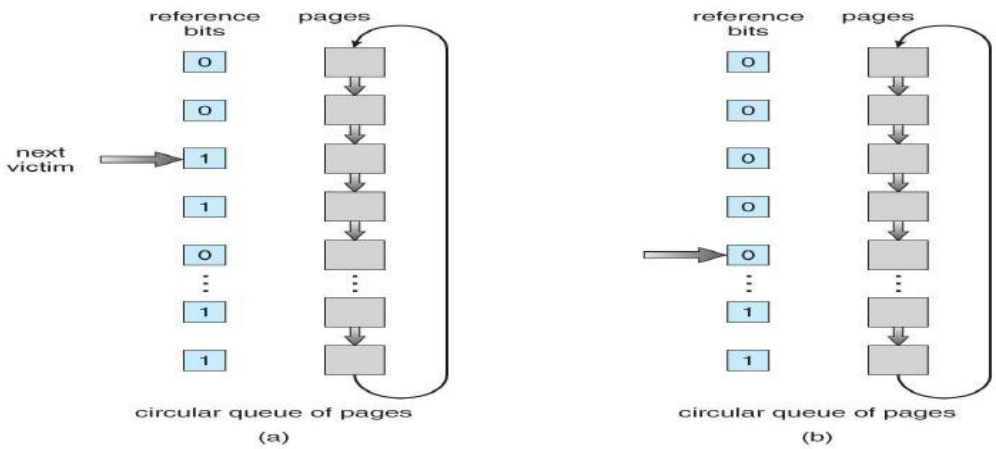


Figure - Second-chance (clock) page-replacement algEnhanced Second-Chance Algorithm

The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:

1. (0, 0) - Neither recently used nor modified.
2. (0, 1) - Not recently used, but modified.
3. (1, 0) - Recently used, but clean.
4. (1, 1) - Recently used and modified.

- This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. I.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

Counting-Based Page Replacement

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
  - ***Least Frequently Used, LFU:*** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
  - ***Most Frequently Used, MFU:*** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.
- In general counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate OPT well.

Page-Buffering Algorithms

- There are a number of page-buffering algorithms that can be used in conjunction with the afore-mentioned algorithms, to improve overall performance and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms:
- Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and then select a victim page to write to disk and free up a frame as a second step.
- Keep a list of modified pages, and when the I/O system is otherwise idle, have it write these pages out to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim.
- Keep a pool of free frames, but remember what page was in it before it was made free. Since the data in the page is not actually cleared out when the page is freed, it can be made an active page again without having to load in any new data from disk. This is useful when an algorithm mistakenly replaces a page that in fact is needed again soon.

Applications and Page Replacement

- Some applications (most notably database programs) understand their data accessing and caching needs better than the general-purpose OS, and should therefore be given reign to do their own memory management.

- Sometimes such programs are given a ***raw disk partition*** to work with, containing raw data blocks and no file system structure. It is then up to the application to use this disk partition as extended memory or for whatever other reasons it sees fit.

Allocation of Frames

We said earlier that there were two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy. This section covers the second part of that pair.

Minimum Number of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture, and corresponds to the worst-case scenario of the number of pages that could be touched by a single (machine) instruction.
- If an instruction (and its operands) spans a page boundary, then multiple pages could be needed just for the instruction fetch.
- Memory references in an instruction touch more pages, and if those memory locations can span page boundaries, then multiple pages could be needed for operand access also.
- The worst case involves indirect addressing, particularly where multiple levels of indirect addressing are allowed. Left unchecked, a pointer to a pointer to a pointer to a pointer to a . . . could theoretically touch every page in the virtual address space in a single machine instruction, requiring every virtual page be loaded into physical memory simultaneously.

Allocation Algorithms

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is s_i, and S is the sum of all s_i, then the allocation for process Pi is $a_i = m * s_i / S$.
- Variations on proportional allocation could consider priority of process rather than just their size.
- Obviously all allocations fluctuate over time as the number of available free frames, m, fluctuates, and all are also subject to the constraints of minimum allocation. (If the minimum allocations cannot be met, then processes must either be swapped out or not allowed to start until more free frames become available.)

Global versus Local Allocation

- One big question is whether frame allocation (page replacement) occurs on a local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

Non-Uniform Memory Access

The above arguments all assume that all memory is equivalent, or at least has equivalent access times.

- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In these latter systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.
- The presence of threads complicates the picture, especially when the threads get loaded onto different processors.
- Solaris uses an ***lgroup*** as a solution, in a hierarchical fashion based on relative latency.

Thrashing

- If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes. This is an intermediate level of CPU scheduling.
- But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.
- A process that is spending more time paging than executing is said to be ***thrashing.***

Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the schedule to add in even more processes and exacerbating the problem! Eventually the system would essentially grind to a halt.
- Local page replacement policies can prevent one thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue, thereby slowing down any other process that needs to do even a little bit of paging (or any other I/O for that matter.)

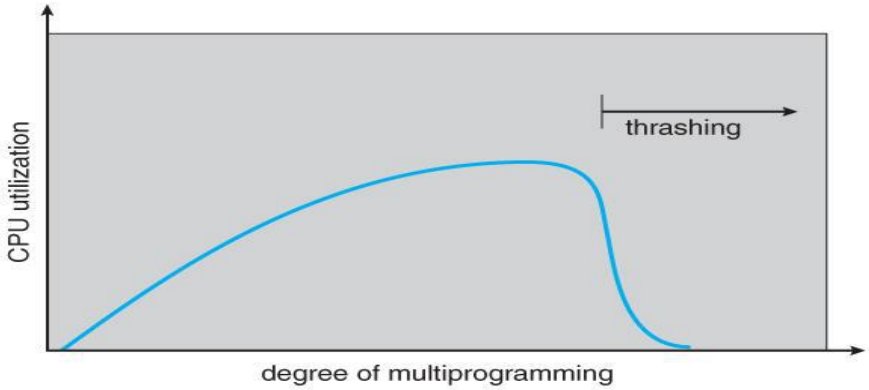


Figure - Thrashing

- To prevent thrashing we must provide processes with as many frames as they really need "right now", but how do we know what that is?
- The ***locality model*** notes that processes typically access memory references in a given ***locality,*** making lots of references to the same general area of memory before moving periodically to a new locality, as shown in Figure 9.19 below. If we could just keep as many frames

as are involved in the current locality, then page faulting would occur primarily on switches from one locality to another. (E.g. when one function exits and another is called.)

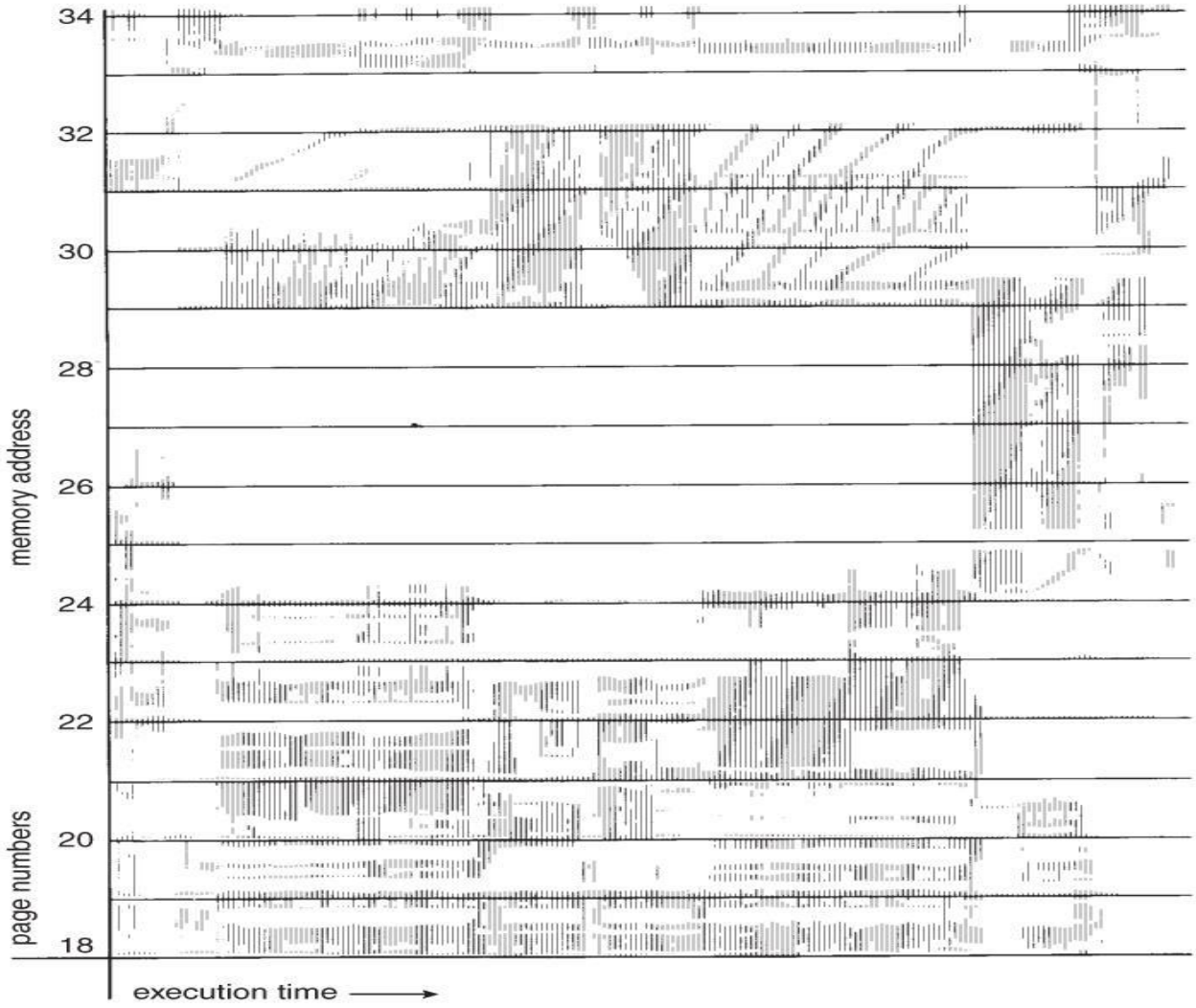


Figure - Locality in a memory-reference pattern.

Working-Set Model

- The ***working set model*** is based on the concept of locality, and defines a ***working set window***, of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure:

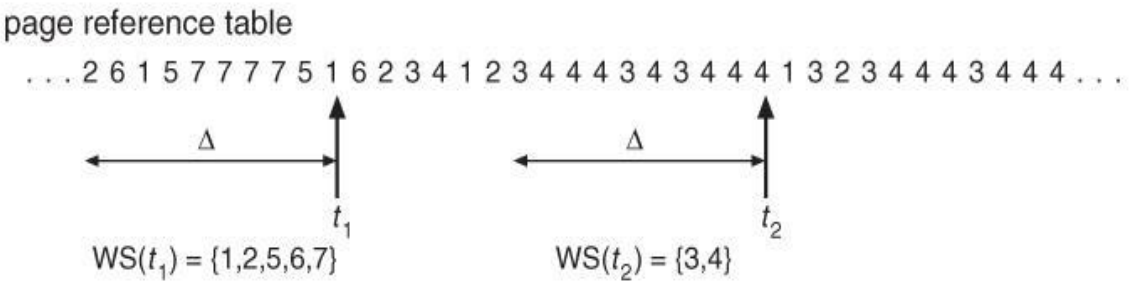


Figure - Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.

- The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.
- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references:

Page-Fault Frequency

- A more direct approach is to recognize that what we really want to control is the page-fault rate, and to allocate frames based on this directly measurable value. If the page-fault rate exceeds a certain upper bound then that process needs more frames, and if it is below a given lower bound, then it can afford to give up some of its frames to other processes.

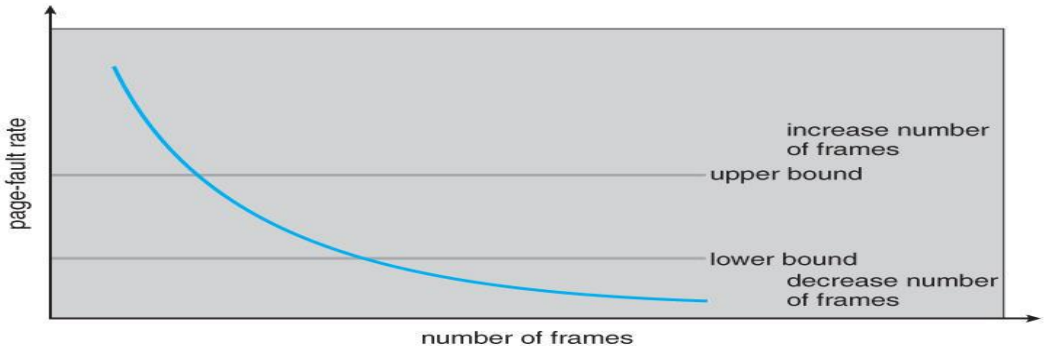


Figure - Page-fault frequency.

Memory-Mapped Files

Basic Mechanism

- Memory mapping a file is accomplished by mapping a disk block to a page (or pages) in memory. Initial access to the file proceeds through ordinary demand paging, resulting in a page fault. However, a page-sized portion of the file is read from the file system into a physical.
- Subsequent reads and writes to the file are handled as routine memory accesses. Manipulating files through memory rather than incurring the overhead of using the read() and write() system calls simplifies and speeds up file access and usage.
- Note that writes to the file mapped in memory are not necessarily immediate (synchronous) writes to the file on disk. When the file is closed, all the memory-mapped data are written back to disk and removed from the virtual memory of the process.
- Some operating systems provide memory mapping only through a specific system call and use the standard system calls to perform all other file I/O. Let's take Solaris as an example. If a file is specified as memory-mapped (using the mmap() system call), Solaris maps the file into the address space of the process.
- If a file is opened and accessed using ordinary system calls, such as open(), read(), and write(),Solaris still memory-maps the file; however, the file is mapped to the kernel address space. Regardless of how the file is opened, then, Solaris treats all file I/O as memory-mapped, allowing file access to take place via the efficient memory subsystem.
- Multiple processes may be allowed to map the same file concurrently, to allow sharing of data. Writes by any of the processes modify the data in virtual memory and can be seen by all others that map the same section of the file.

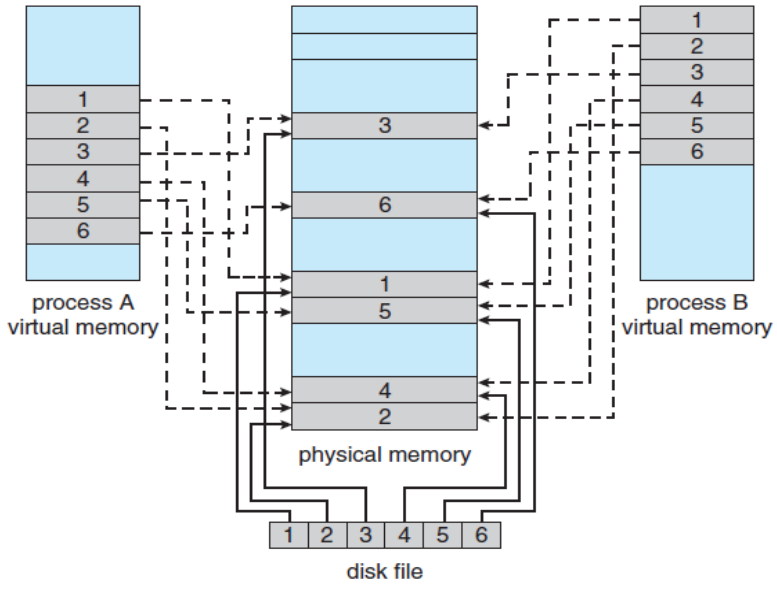


Figure 9.22 Memory-mapped files.

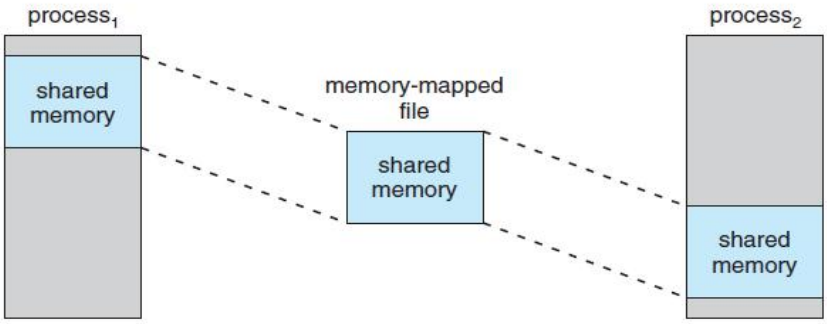


Figure 9.23 Shared memory using memory-mapped I/O.

Shared Memory in the Windows API

- The general outline for creating a region of shared memory using memory mapped files in the Windows API involves first creating a file mapping for the file to be mapped and then establishing a view of the mapped file in a process's virtual address space.
- A second process can then open and create a view of the mapped file in its virtual address space. The mapped file represents the shared-memory object that will enable communication to take place between the processes.
- A producer process first creates a shared-memory object using the memory-mapping features available in the Windows API. The producer then writes a message to shared memory. After that, a consumer process opens a mapping to the shared-memory object and reads the message written by the consumer.
- To establish a memory-mapped file, a process first opens the file to be mapped with the CreateFile() function, which returns a HANDLE to the opened file. The process then creates a mapping of this file HANDLE using the CreateFileMapping() function.
- Once the file mapping is established, the process then establishes a view of the mapped file in its virtual address space with the MapViewOfFile() function. The view of the mapped file represents the portion of the file being mapped in the virtual address space of the process —the entire file or only a portion of it may be mapped.

Memory-Mapped I/O

- To allow more convenient access to I/O devices, many computer architectures provide memory-mapped I/O. In this case, ranges of memory addresses are set aside and are mapped to the device registers. Reads and writes to these memory addresses cause the data to be transferred to and from the device registers. This method is appropriate for devices that have fast response times, such as video controllers.
- Memory-mapped I/O is also convenient for other devices, such as the serial and parallel ports used to connect modems and printers to a computer. The CPU transfers data through these kinds of devices by reading and writing a few device registers, called an I/O **port**.
- To send out a long string of bytes through a memory-mapped serial port, the CPU writes one data byte to the data register and sets a bit in the control register to signal that the byte is available. The device takes the data byte and then clears the bit in the control register to signal that it is ready for the next byte.
- Then the CPU can transfer the next byte. If the CPU uses polling to watch the control bit, constantly looping to see whether the device is ready, this method of operation is called **programmed I/O (PIO)**. If the CPU does not poll the control bit, but instead receives an interrupt when the device is ready for the next byte, the data transfer is said to be **interrupt driven**.

Allocating Kernel Memory

- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

  1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation.
  2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

- Two strategies for managing free memory that is assigned to kernel processes: the "buddy system" and slab allocation.

Buddy System

- The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2.
- For example, a request for 11 KB is satisfied with a 16-KB segment. Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call *AL* and *AR*—each 128 KB in size.
- One of these buddies is further divided into two 64-KB buddies—*BL* and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure , where *CL* is the segment allocated to the 21-KB request.
- An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**. In Figure , for example, when the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment. This segment, *BL*, can in turn be coalesced with its buddy *BR* to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

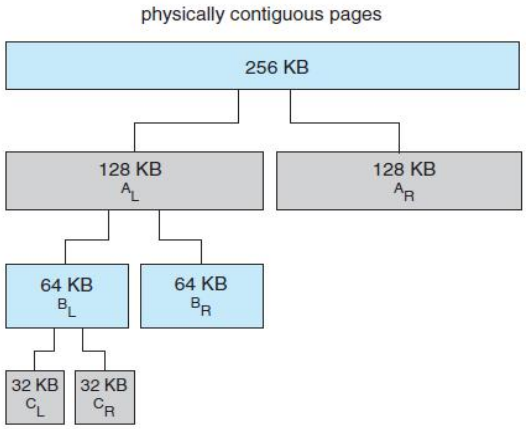


Figure 9.26 Buddy system allocation.

- The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64-KB segment. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation.

Slab Allocation

- A **slab** is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure —for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents.
- The relationship among slabs, caches, and objects is shown in Figure. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

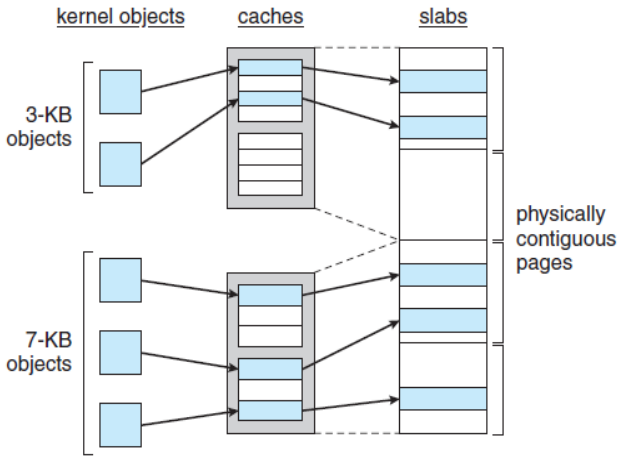


Figure 9.27 Slab allocation.

- In Linux, a slab may be in one of three possible states:
  1. **Full**. All objects in the slab are marked as used.
  2. **Empty**. All objects in the slab are marked as free.
  3. **Partial**. The slab consists of both used and free objects.
- The slab allocator provides two main benefits:
  1. No memory is wasted due to fragmentation.
  2. Memory requests can be satisfied quickly.

Deadlocks: System Model, Deadlock Characterization, Methods to Handling deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

■ Example
- System has 2 tape drives.
- P0 and P1 each hold one tape drive and each needs another one.

■ Example
- semaphores A and B, initialized to 1

P0	P1
wait (A);	wait(B)
wait (B);	wait(A)

System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
  1. **Request** - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
  2. **Use** - The process uses the resource, e.g. prints to the printer or reads from the file.
  3. **Release** - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- For all kernel-managed resources, the kernel keeps track of what resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, (i.e. binary or counting semaphores.)
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

Deadlock Characterization

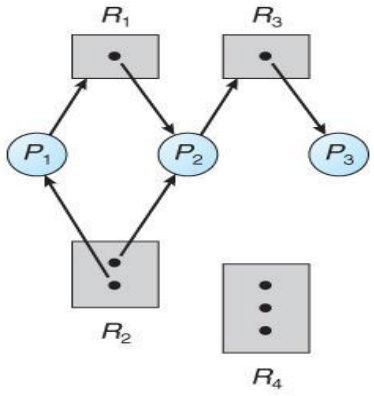
Necessary Conditions
- There are four conditions that are necessary to achieve deadlock:

  1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
  2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
4. **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

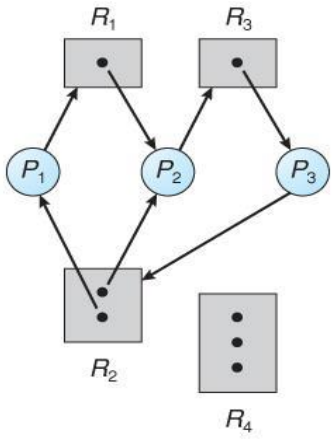
Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties: A set of vertices V and a set of edges E. V is partitioned into two types:
  - A set of resource categories, R={ R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)
  - A set of processes, P={ P1, P2, P3, . . ., PN }
  - **Request Edges** - A set of directed arcs from (Pi → Rj) Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
  - **Assignment Edges** - A set of directed arcs from (Rj → Pi) Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
  - Note that a **request edge** can be converted into an assignment edge by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas **assignment edges** emanate from a particular instance dot within the box.)
  - For example:



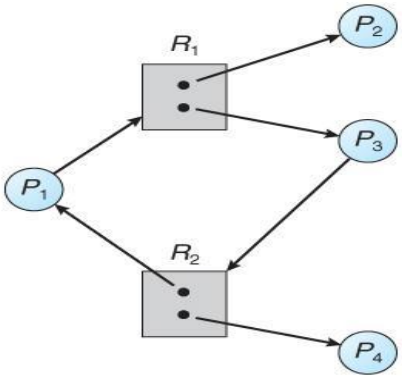
Resource allocation graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are *directed* graphs.) See the example in Figure above.
- If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures below:
  - the possible cycles are P1→R1→P2→R3→P3→R2→P1,
    P2→R3→P3→R2→P2



Resource allocation graph with a deadlock

The possible cycle in the below graph is P1→R1→P3→R2→P1



Resource allocation graph with a cycle but no deadlock

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
    1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
    2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
    3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.
- In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future
- Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

Deadlock Prevention

- Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
    - Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
    - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
    - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
    - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
    - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources ***and*** are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
    - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

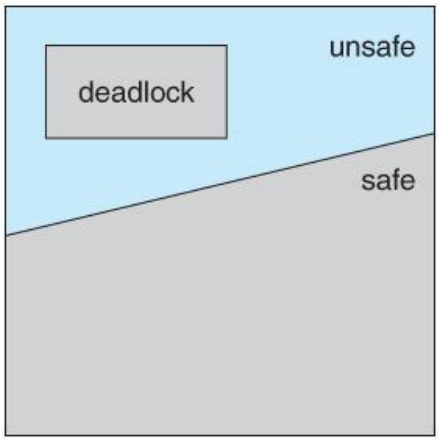
Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)

- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)



Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

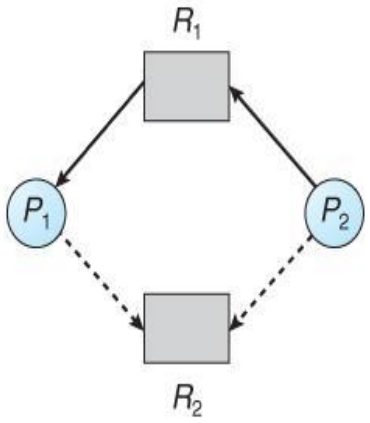
	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

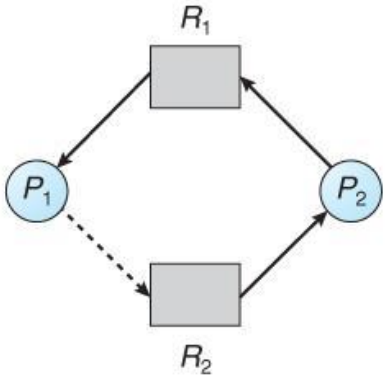
- If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges,*** noted by dashed lines, which point from a process to a resource that it may request in the future.

- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:



Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.



An unsafe state in a resource allocation graph

Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.

- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
    - **Available[m]** indicates how many resources are currently available of each type.
    - **Max[n][m]** indicates the maximum demand of each process of each resource.
    - **Allocation[n][m]** indicates the number of each resource category allocated to each process.
    - **Need[n][m]** indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
    - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
    - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

Safety Algorithm

- In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
- This algorithm determines if the current state of a system is safe, according to the following steps:
    1. Let Work and Finish be vectors of length m and n respectively.
        - Work is a working copy of the available resources, which will be modified during the analysis.
        - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
        - Initialize Work to Available, and Finish to false for all elements.
    2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
    3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
    4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.

- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:

    1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
    2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
    3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely. The procedure for granting a request (or pretending to for testing purposes) is:
        - Available = Available - Request
        - Allocation = Allocation + Request
        - Need = Need - Request

An Illustrative Example

- Consider the following situation:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

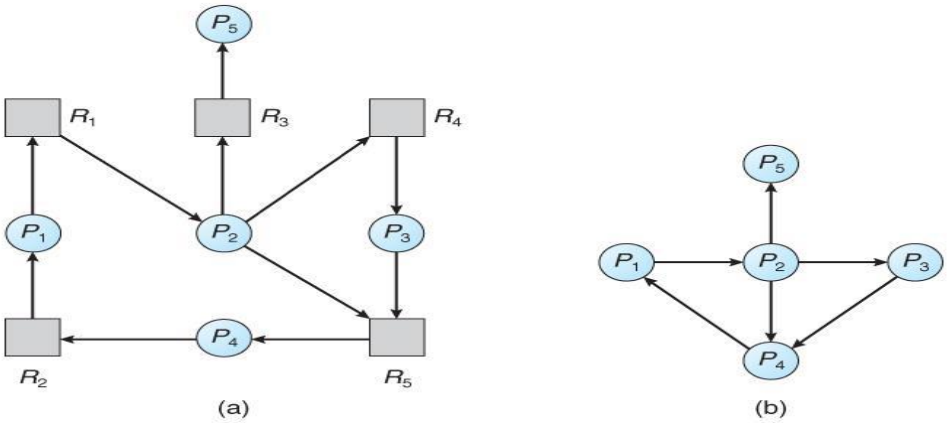
- What about requests of (3, 3,0) by P4? or (0, 2, 0) by P0? Can these be safely granted? Why or why not?

Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.



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

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

Several Instances of a Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
    - In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
    - Steps 2 and 3 are unchanged
    - In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

- (Note: An alternative method was presented above, in which Finish held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes.)
- Consider, for example, the following state, and determine if it is currently deadlocked:

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

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?

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

Detection-Algorithm Usage

- When should the deadlock detection be done? Frequently, or infrequently?
- The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. (If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes.)
- There are two obvious approaches, each with trade-offs:

    1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. (One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock.) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.

    2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.

Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
    1. Inform the system operator, and allow him/her to take manual intervention.
    2. Terminate one or more processes involved in the deadlock
    3. Preempt resources.

Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
    - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
    - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
- In the latter case there are many factors that can go into deciding which processes to terminate next:
    1. Process priorities.
    2. How long the process has been running, and how close it is to finishing.
    3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
    4. How many more resources does the process need to complete.
    5. How many processes will need to be terminated
    6. Whether the process is interactive or batch.
    7. (Whether or not the process has made non-restorable changes to any resource.)

Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

PREVIOUS PAPERS QUESTIONS

1.a) What is compaction? What are its advantages and disadvantages?
b) Explain enhanced second-chance page replacement algorithm with an example. What is the major difference between this algorithm and the simpler clock algorithm?

2.a) With a neat diagram explain paging hardware with TLB.
b) Explain memory management in UNIX.

3.Explain in detail paging with an example. Give the paging hardware and explain page table implementation.

4. Explain the most common techniques for structuring the page table.

5.a) Explain why sharing a reentrant module is easier when segmentation is used than pure paging is used.
b) Compare the main memory organization schemes of contiguous memory allocation, pure segmentation and pure paging with respect to the following issues:
(a) External fragmentation.
(b) Internal fragmentation.
(c) Ability to share code across processes.

6.a) What is 50 percent rule of fragmentation?
b) Compare and contrast paging and segmentation.

7.a) What is the purpose of swapping? Explain the variants of swapping.
b) Explain optimal page replacement algorithm with an example.

8.a) Explain the concept of swapping.
b) Discuss LRU Page replacement algorithm.

9.a) What is meant by relocation? Give the necessary hardware for implementing dynamic relocation and explain.
b) Explain FIFO page replacement algorithm with an example.

10.a) Why is it that the size of the page is typically a power of 2?
b) Explain the different hardware implementations of page table.

11.a) What is meant by memory protection? Explain how memory is protected by using base and limit register.

b) Discuss hardware support required to support demand paging.

12. a) What is the purpose of base and limit register?

b) What is segmentation? Explain with an example . also give the segmentation hardware.

13. Why can a deadlock not be prevented easily? Discuss this with respect to the necessary conditions for deadlock, considering each of them one by one.

14. a). Consider the dining philosopher's problem when the chopsticks are placed at the center of the table and any two of them could be used by philosopher. Assume that the requests for chopsticks

are made one at a time. Describe a simple rule for determining whether a particular request could be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.
b). Illustrate resource allocation graph.

15. Explain the different strategies that operating system designers can adopt vis-à-vis the problem of deadlock

16.a). Consider a system consisting of 'm' resources of the same type being shared by 'n' processes, resources can be requested and released by processes only one at a time. Show that the system is deadlock free if the following two conditions hold:
(i) The maximum need of each process is between 1 and 'm' resources.
(ii) The sum of all needs is less than m + n.
b). Under the normal mode of operation, what is the sequence of operations a process performs while utilizing a resource?

17.a). Define deadlock prevention and deadlock avoidance.
b). Explain in detail the deadlock recovery techniques.

18. Describe Banker's algorithm to avoid a deadlock. What are the problems in its implementation?

19.Consider the deadlock situation that could occur in the dining philosopher's problem when the philosophers obtain the chopsticks one at a time. Discuss the four conditions for deadlocks indeed hold in the setting. Discuss how deadlocks could be avoided by eliminating any one of the four conditions

20. Explain Banker's algorithm for deadlock avoidance with an example

21. Discuss the methods for handling deadlocks.

UNIT-IV

File System Interface: File Concept, Access Methods, Disk and Directory Structure, File System Mounting, File Sharing

It provides the mechanism for online storage of and access to both data and programs of OS 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

- Computers can store information on various storage media such as magnetic disks, magnetic tapes and optical disks.
- OS abstracts from the physical properties of its storage devices to define a logical storage unit called a **file**. Files are mapped by OS onto physical devices. These storage devices are non volatile so the contents are persistent through power failures and system reboots.
- A file is a named collection of related information that is recorded on secondary storage. 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.
- Files represent programs 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. A file is a sequence of bits, bytes, lines or records.
- Information in a file is defined by its creator. Many different types of information may be stored in a file – source programs, object programs, executable programs, numeric data, text etc. A file has a certain defined structure which depends on its type.

- Text file –sequence of characters organized into lines
- Source file – sequence of sub routines and functions each of which is further organized as declarations followed by executable statements.
- Object file – sequence of bytes organized into blocks understandable by the system's linker
- Executable file – series of code sections that the loader can bring into memory and execute.

File Attributes

- A file is referred to by its name. A name is usually a string of characters. When a file is named, it becomes independent of the process, the user and even the system that created it.
- A file's attributes vary from one OS to another but consist of these –

Name: symbolic file name is the only information kept in human readable form.

Identifier: number which identifies the file within the file system; it is the non human readable name for the file.

Type: 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

Protection: Access control information determines who can do reading, writing, executing etc.

Time, date and user identification: This information may be kept for creation, last modification and last use.

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

File Operations

- A file is an abstract data type. OS can provide system calls to create, write, read, reposition, delete and truncate files.

Creating 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, specify both the name of the file and the information to be written to the file. The system must keep a write pointer to the location in the file where the next write is to take place.
Reading a file – To read from a file, 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. Because a process is either reading from or writing to a file, the current operation location can be kept as a per process current file position pointer.
Repositioning within a file – Directory is searched for the appropriate entry and the current file position pointer is repositioned to a given value. This operation is also known as file seek.
Deleting a file – To delete a file, search the directory for the named file. When found, release all file space and erase the directory entry.
Truncating a file – User may want to erase the contents of a file but keep its attributes. This function allows all attributes to remain unchanged except for file length.

- Other common operations include appending new information to the end of an existing file and renaming an existing file. We may also need operations that allow the user to get and set the various attributes of a file.
- Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant search, many systems require that an open () system call be made before a file is first used actively.
- OS keeps a small table called the **open file table** containing information about all open files. When a file operation is requested, the file is specified via an index into this table so no searching is required.

To summarize, **several pieces of information are associated with an open file.**

- File pointer – System must keep track of the last read – write location as a current file position pointer.
- File open count – As files are closed, OS must reuse its open file entries or it could run out of space in the table. File open counter tracks the number of opens and closes and reaches zero on the last close.
- Disk location of the file – The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- Access rights – Each process opens a file a file in an access mode. This information is stored on the per process table so the OS can allow or deny subsequent I/O requests.

- Some OS's provide facilities for locking an open file. File locks allow one process to lock a file and prevent other processes from gaining access to it.
- File locks are useful for files that are shared by several processes. A **shared lock** is where several processes can acquire the lock concurrently. An **exclusive lock** is where only one process at a time can acquire such a lock.
- Also some OS's may provide either **mandatory or advisory** file locking mechanisms. If a lock is mandatory, then once a process acquires an exclusive lock, the OS will prevent any other process from accessing the locked file. If the lock scheme is mandatory, OS ensures locking integrity.

File types

- 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 separated by a period character. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

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, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

File structure

- File types can be used to indicate the internal structure of the file. Source and object files have structures that match the expectations of the programs that read them. Certain files conform to a required structure that is understood by OS.
- But the disadvantage of having the OS support multiple file structures is that the resulting size of the OS is cumbersome. If the OS contains five different file structures, it needs to contain the code to support these file structures.
- Hence some OS's impose a minimal number of file structures. MAC OS also supports a minimal number of file structures. It expects files to contain two parts – a resource fork and a data fork. The resource fork contains information of interest to the user. The data fork contains program code or data – traditional file contents.

Internal file structure

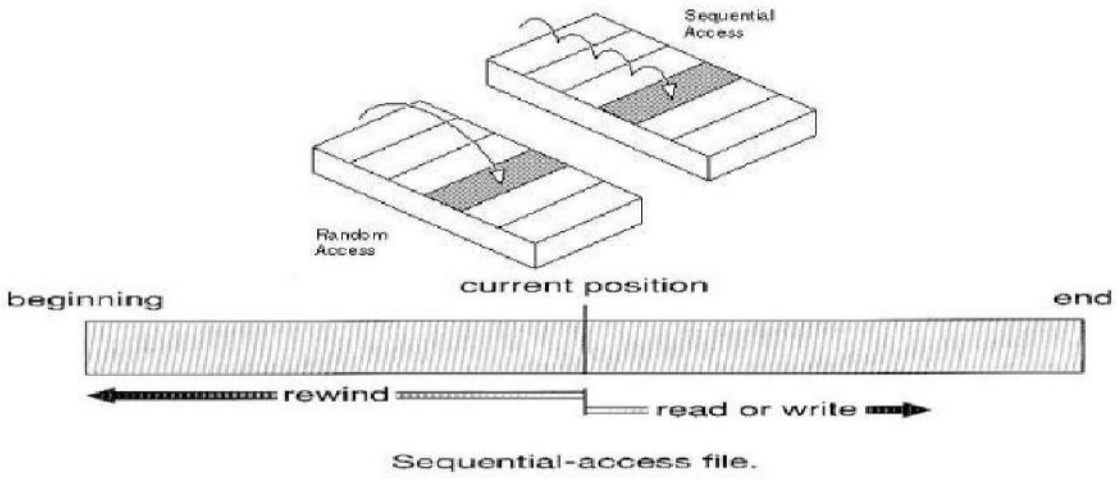
- Internally locating an offset within a file can be complicated for the OS. Disk systems have a well defined block size determined by the size of the sector. All disk I/O is performed in units of one block and all blocks are the same size.
- Since it is unlikely that the physical record size will exactly match the length of the desired logical record, and then logical records may even vary in length, packing a number of logical records into physical blocks is a solution.
- The logical record size, physical block size and packing technique determine how many logical

records are in each physical block. The packing can be done either by the user's application program or by the OS. Hence the file may be considered to be a sequence of blocks. All the basic I/O functions operate in terms of blocks.

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. They are –

Sequential access: Simplest method. Information in the file is processed in order that is one record after the other. This method is based on a tape model of a file and works as well on sequential access devices as it does on random access



Sequential-access file.

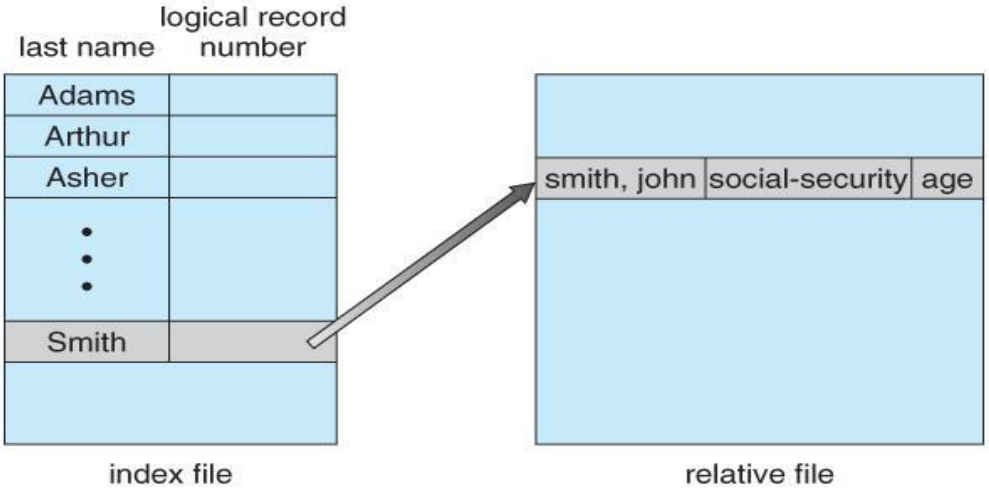
Direct access: Another method is direct access or relative access. A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct access method is based on a disk model of a file since disks allow random access to any file block.

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

The block number provided by the user to the OS is a relative block number. A relative block number is an index relative to the beginning of the file. The use of relative block numbers allows the OS to decide where the file should be placed and helps to prevent the user from accessing portions of the file system that may not be a part of the file.

Some systems allow only sequential file access; others allow only direct access.

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. This index contains pointers to the various blocks. To find a record in the file, first search the index and then use the pointer to access the file directly and to find the desired record.



But 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 would contain pointers to secondary index files which would point to actual data items.

Directory Structure

- Systems may have zero or more file systems and the file systems may be of varying types. Organizing millions of files involves use of directories.

Storage Structure

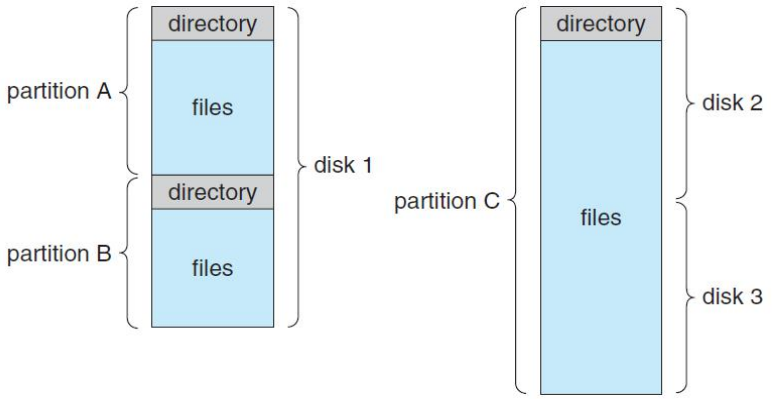


Figure 11.7 A typical file-system organization.

- A disk can be used in its entirety for a file system. But at times, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things. These parts are known variously as **partitions, slices or minidisks**.
- A file system can be created on each of these parts of the disk. These parts can be combined together to form larger structures known as **volumes** and file systems can be created on these too. Each volume can be thought of as a virtual disk. Volumes can also store multiple OS's allowing a system to boot and run more than one.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory or volume table of contents**. The device directory/directory records information for all files on that volume.

Directory Overview

- The directory can be viewed as a symbol table that translates file names into their directory entries. The operations that can be performed on the directory are:
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system

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. But this implementation has limitations when the number of files increases or when the system has more than one user.
- Since all files are in same directory, all files names must be unique. Keeping track of so many files is a difficult task. 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.

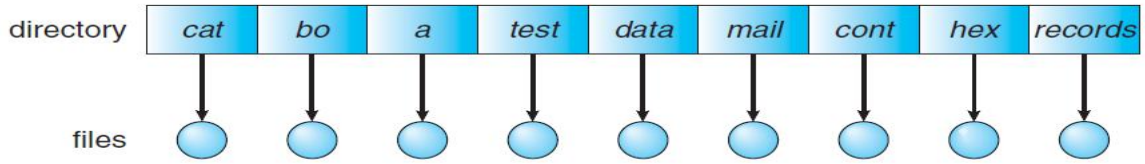


Figure 11.9 Single-level directory.

Two level directory

- In the two level directory structure, each user has his own **user file directory** (UFD). The UFD's 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. When a user refers to a particular file, only his own UFD is searched. Different users may have files with the same name as long as all the files names within each UFD are unique.
- Root of the tree is MFD. Its direct descendants are UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree.
- The sequence of directories searched when a file is names is called the **search path**.

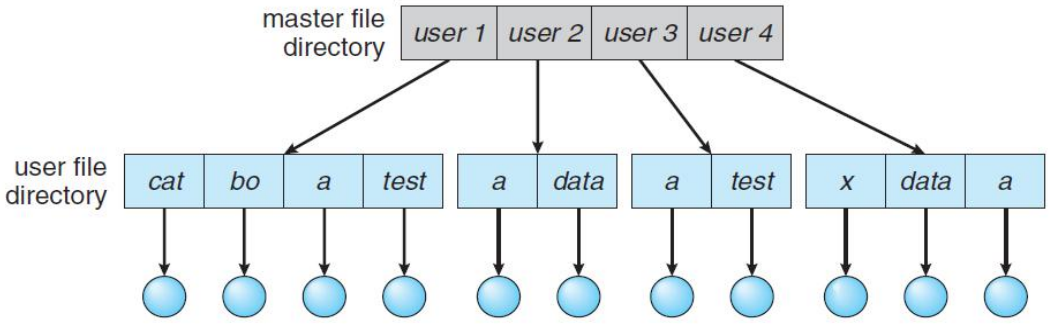


Figure 11.10 Two-level directory structure.

- Although the two level directory structure solves the name collision problem, it still has disadvantages. This structure isolates on user from another. Isolation is an advantage when the users are completely independent but a disadvantage when the users want to cooperate on some task and to access one another's files.

Tree Structured Directories

- Here, we extend the two level directory to a tree of arbitrary height. This generalization allows users to create their own subdirectories and to organize their files accordingly. 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 contains a set of files or sub directories. 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).

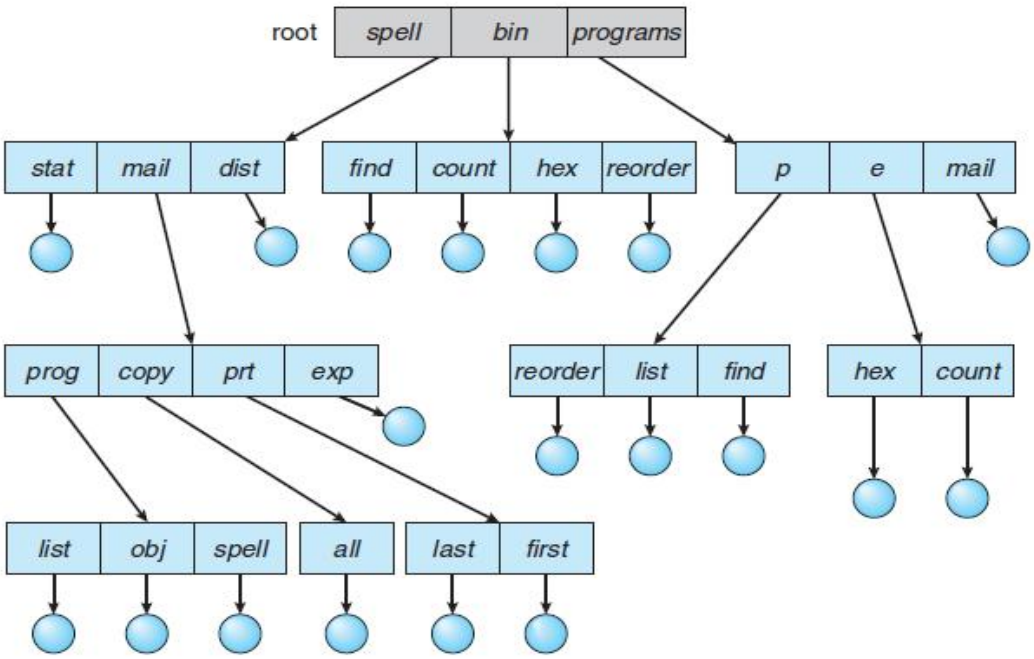


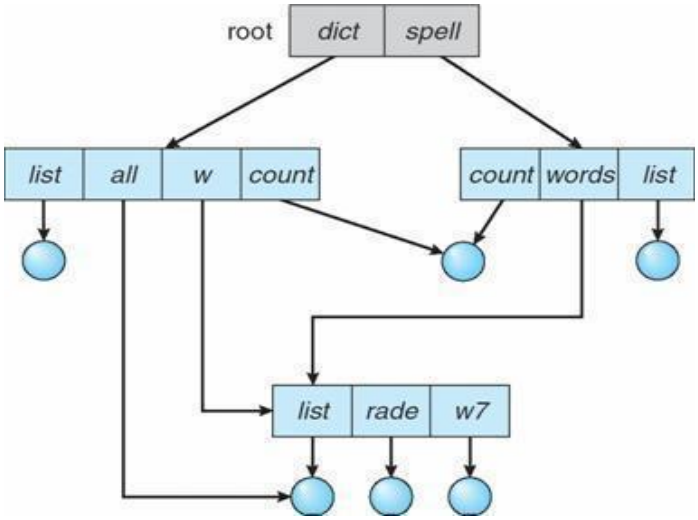
Figure 11.11 Tree-structured directory structure.

- Each process has a current directory. The current directory should contain most of the files that are of current interest to the process.
- 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.
- Deletion of directory under tree structured directory – If a directory is empty, its entry in the directory that contains it can simply be deleted. If the directory to be deleted is not empty, then use one of the two approaches –
  - ✓ User must first delete all the files in that directory
  - If a request is made to delete a directory, all the directory's files and sub directories are also to be deleted. A path to a file in a tree structured directory can be longer than a path in a two level directory.

Acyclic graph directories

- A tree structure prohibits the sharing of files and directories. An acyclic graph i.e. a graph with no cycles allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories.
- With a shared file, only one actual file exists. Sharing is particularly important for subdirectories. Shared files and subdirectories can be implemented in several ways.
- One way is to create a new directory entry called a link. A link is a pointer to another file or subdirectory. Another approach in implementing shared files is to duplicate all information about

them in both sharing directories.



- An acyclic graph directory structure is flexible than a tree structure but it is more complex. Several problems may exist such as multiple absolute path names or deletion.
- Another problem involves deletion. When can the space allocated to a shared file be deallocated and reused? One possibility is to remove the file whenever anyone deletes it, but this action may leave **dangling pointers** to the now-non existent file. Worse, if the remaining file pointers contain actual disk addresses, and the space is subsequently reused for other files, these dangling pointers may point into the middle of other files.
- In a system where sharing is implemented by symbolic links, this situation is somewhat easier to handle. The deletion of a link need not affect the original file; only the link is removed. If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive.

General graph directory

- A problem with using an acyclic graph structure is ensuring that there are no cycles.

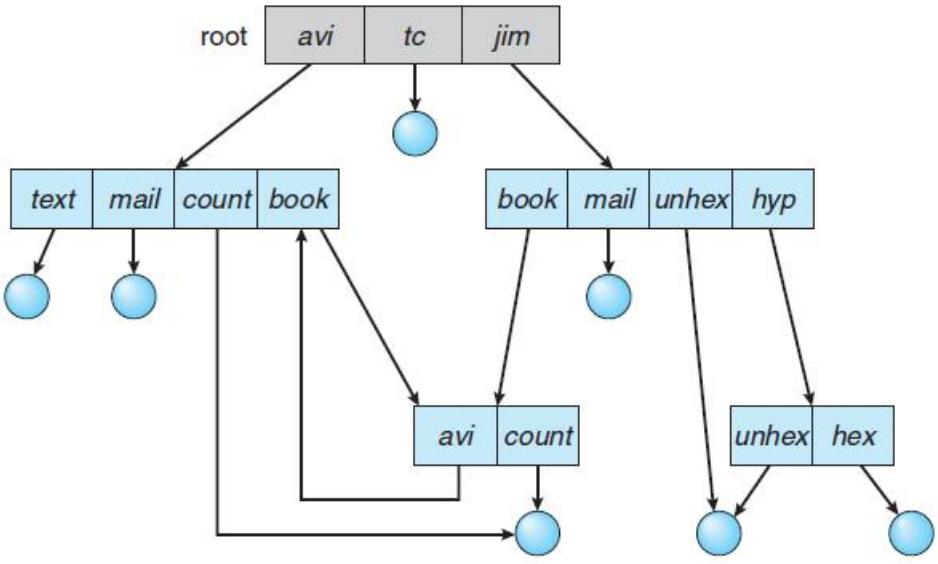
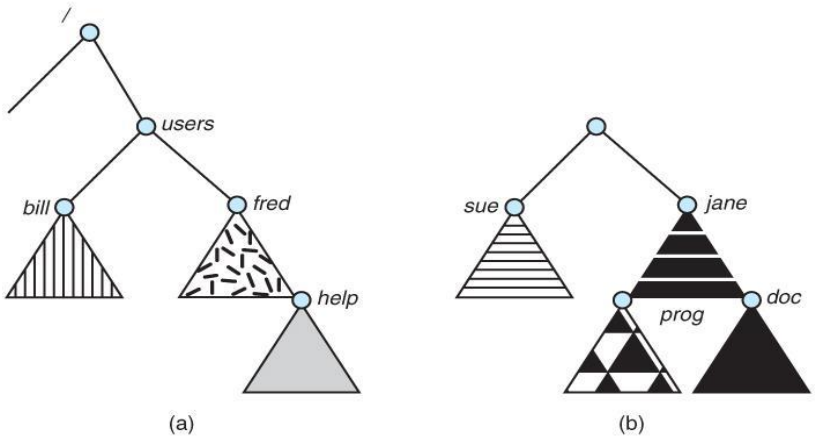


Figure 11.13 General graph directory.

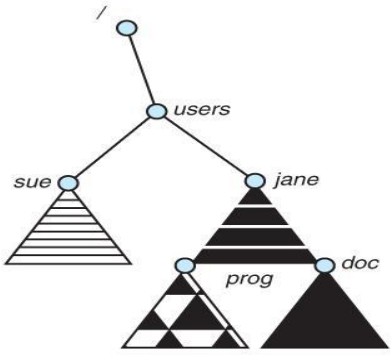
- The primary advantage of an acyclic graph is the relative simplicity of the algorithms to traverse the graph and to determine when there are no more references to a file. If cycles are allowed to exist in the directory, avoid searching any component twice.
- A similar problem exists when we are trying to determine when a file can be deleted. The difficulty is to avoid cycles as new links are added to the structure.
- A similar problem exists when we are trying to determine when a file can be deleted. With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted. However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. This anomaly results from the possibility of self-referencing (or a cycle) in the directory structure.
- In this case, we generally need to use a **garbage collection** scheme to determine when the last reference has been deleted and the disk space can be reallocated. Garbage collection involves traversing the entire file system, marking everything that can be accessed. Then, a second pass collects everything that is not marked onto a list of free space. Garbage collection for a disk-based file system, however, is extremely time consuming and is thus seldom attempted. Garbage collection is necessary only because of possible cycles in the graph.

File System Mounting

- A file system must be mounted before it can be available to processes on the system. OS is given the name of the device and a mount point – the location within the file structure where the file system is to be attached. This 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, OS 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 OS notes in its directory structure that a file system is mounted at the specified mount point.



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



Mount point.

- Consider the actions of the Mac OS X operating system. Whenever the system encounters a disk for the first time (either at boot time or while the system is running), the Mac OS X operating system searches for a file system on the device.
- If it finds one, it automatically mounts the file system under the /Volumes directory, adding a folder icon labeled with the name of the file system (as stored in the device directory). The user is then able to click on the icon and thus display the newly mounted file system.
- The Microsoft Windows family of operating systems maintains an extended two-level directory structure, with devices and volumes assigned drive letters. The path to a specific file takes the form of drive-letter:\path\to\file.

File Sharing

- File sharing is desirable for users who want to collaborate and to reduce the effort required to achieve a computing goal.

Multiple users

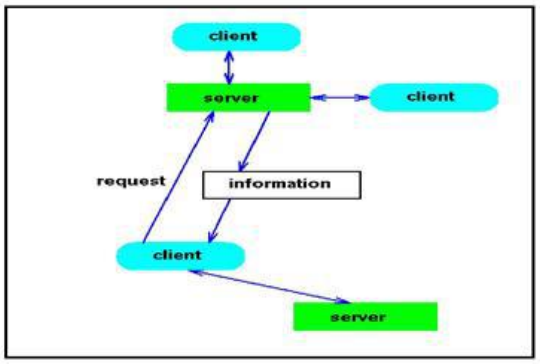
- When an OS accommodates multiple users, the issues of file sharing, file naming and file protection become preeminent. System mediates file sharing. The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

Remote File Systems

- Networking allows sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files.
- The first implemented file sharing is method involves manually transferring files between machines via programs like ftp. The second major method uses a distributed file system in which remote directories are visible from a local machine. The third method is through WWW.
- ftp is used for both anonymous and authenticated access. Anonymous access allows a user to transfer files without having an account on the remote system.
- WWW uses anonymous files exchange almost exclusively. DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

Client Server Model

- Remote file systems allow a computer to mount one or more file systems from one or more remote machines. Here the machine containing the files is the server and the machine seeking access to the files is the client.
- A server can serve multiple clients and a client can use multiple servers depending on the implementation details of a given client server facility. Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol.



Distributed Information Systems

- To make client server systems easier to manage, distributed information systems also known as distributed naming services provide unified access to the information needed for remote computing. The domain name system provides host name to network address translations for the entire Internet.
- Distributed information systems used by some companies –

  Sun Microsystems – Network Information Service or NIS

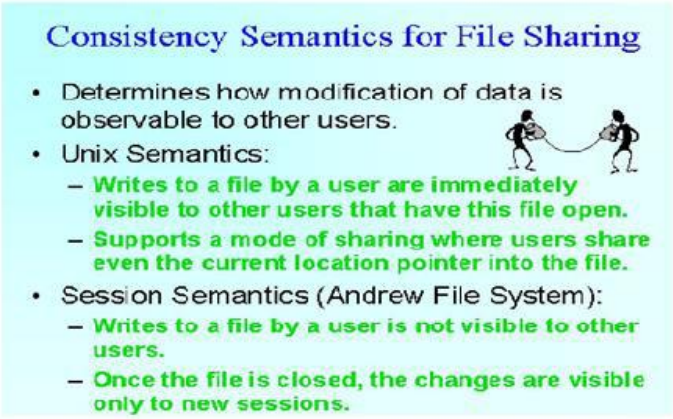
  Microsoft – Common internet file system or CIFS

Failure Modes

- Local file systems can fail for a variety of reasons including failure of the disk containing the file system, corruption of the delivery structure or other disk management information, disk controller failure, cable failure and host adapter failure.
- User or system administrator failure can also cause files to be lost or entire directories or volumes to be deleted. Many of these failures will cause a host to crash and an error condition to be displayed and human intervention will be required to repair the damage.
- Remote fail systems have even more failure modes. In the case of networks, the network can be interrupted between two hosts. Such interruption can result from hardware failure, poor hardware configuration or networking implementation issues.
- For a recovery from a failure, some kind of state information may be maintained on both the client and server.

Consistency semantics

- These represent an important criterion for evaluating any file system that supports file sharing. These semantics specify how multiple users of a system are to access a shared file simultaneously. These are typically implemented as code with the file system.



- **Immutable-Shared-Files SemanticsProtection**
  Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: its name may not be reused, and its contents may not be altered. Thus, the name of an immutable file signifies that the contents of the file are fixed.

- When information is stored in a computer system, it should be kept safe from physical damage (reliability) and improper access (protection). Reliability is provided by duplicate copies of files.
- Protection can be provided in many ways such as physically removing the floppy disks and locking them up.

Types of Access

- Complete protection to files can be provided by prohibiting access. Systems that do not permit access to the files of other users do not need protection. Both these approaches are extreme. Hence **controlled access** is required.
- Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on many factors. Several different types of operations may be controlled –

  i. Read
  ii. Write
  iii. Execute
  iv. Append
  v. Delete
  vi. List

- Other operations such as renaming, copying etc., may also be controlled.

Access Control

- The most common approach to the protection problem is to make access dependent on the identity of the user. The most general scheme to implement identity- dependent access is to associate with each file and directory an access- control list (ACL) specifying user names and the types of access allowed for each user.
- This approach has the advantage of enabling complex access methodologies. The main problem with access lists is their length. To condense the length of the access control list, many systems recognize three classifications of users in connection with each file:

a) Owner – user who created the file
b) Group – set of users who are sharing the file and need similar access
c) Universe – all other users in the system
d) With the more limited protection classification, only three fields are needed to define protection. Each field is a collection of bits and each bit either allows or prevents the access associated with it. A separate field is kept for the file owner for the file's group and for all the other users.

Other Protection Approaches

- Another approach to protection problem is to associate a password with each file. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.
- Use of passwords has certain disadvantages –

1. The number of passwords that a user needs to remember may become large making the scheme impractical.
2. If only one password is used for all the files, then once it is discovered, all files are accessible.

Implementing File Systems: File System Structure, File System Implementation, Directory Implementation, Allocation Methods, Free-Space Management

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 the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:
  - ✓ 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.
  - ✓ A disk an access directly any given block of information it contains.
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors. Sector size varies from 32 bytes to 4,096 bytes; the usual size is 512 bytes.
- To provide efficient and convenient access to the disk, OS imposes one or more file systems to allow the data to be stored, located and retrieved easily. The file system is composed of many different levels –

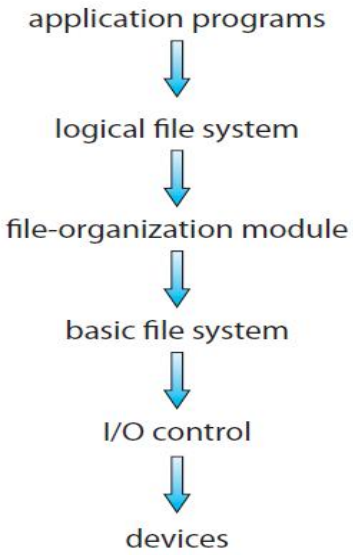


Figure 12.1 Layered file system.

- Each level in the design uses the features of lower levels to create new features for use by higher levels.
- The lowest level, I/O control consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system.
- The basic file system needs to issue generic commands to 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.
- The logical file system manages metadata information. Metadata includes all of the file system structure except the actual data. A **file control block** contains information about the file including ownership, permissions and location of the file contents.
- When a layered structure is used for file-system implementation, duplication of code is minimized.

- UNIX uses the **UNIX file system (UFS)**, which is based on the Berkeley Fast File System (FFS).Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats. Although Linux supports over forty different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4.

File System Implementation

OS's implement open() and close() system calls for processes to request access to file contents.

Overview

- Several on disk and in memory structures are used to implement a file system. These structures vary depending on the OS and the file system. File system may contain information such as:

- **Boot control block -** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty.In UFS, it is called the **boot block**; in NTFS it is **partition boot sector**.
- **Volume control block** – (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, it is called a **super block**; in NTFS it is stored in the **master file table**
- A directory structure per file system is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in master file table.
- A per fie FCB contains many details about the file, including file permissions, ownership, size and location of data blocks. In UFS, it is called the inode. In NTFS this is stored within the master file table which uses a relational database structure.

The structures may include the ones described below –

- ✓ An in memory **mount table** contains information about each mounted volume
- ✓ An in memory directory structure cache holds the directory information of recently accessed directories.
- ✓ The **system wide open file table** contains a copy of the FCB of each open file
- ✓ The **per process open file table** contains a pointer to the appropriate entry in the system wide open file table.

- To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures.
- To create a new file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.)
- The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical FCB is shown in Figure

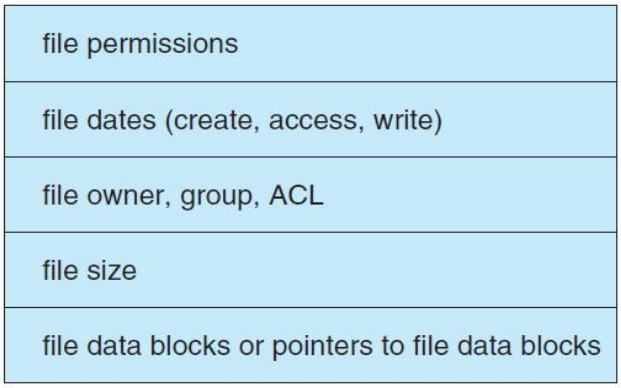
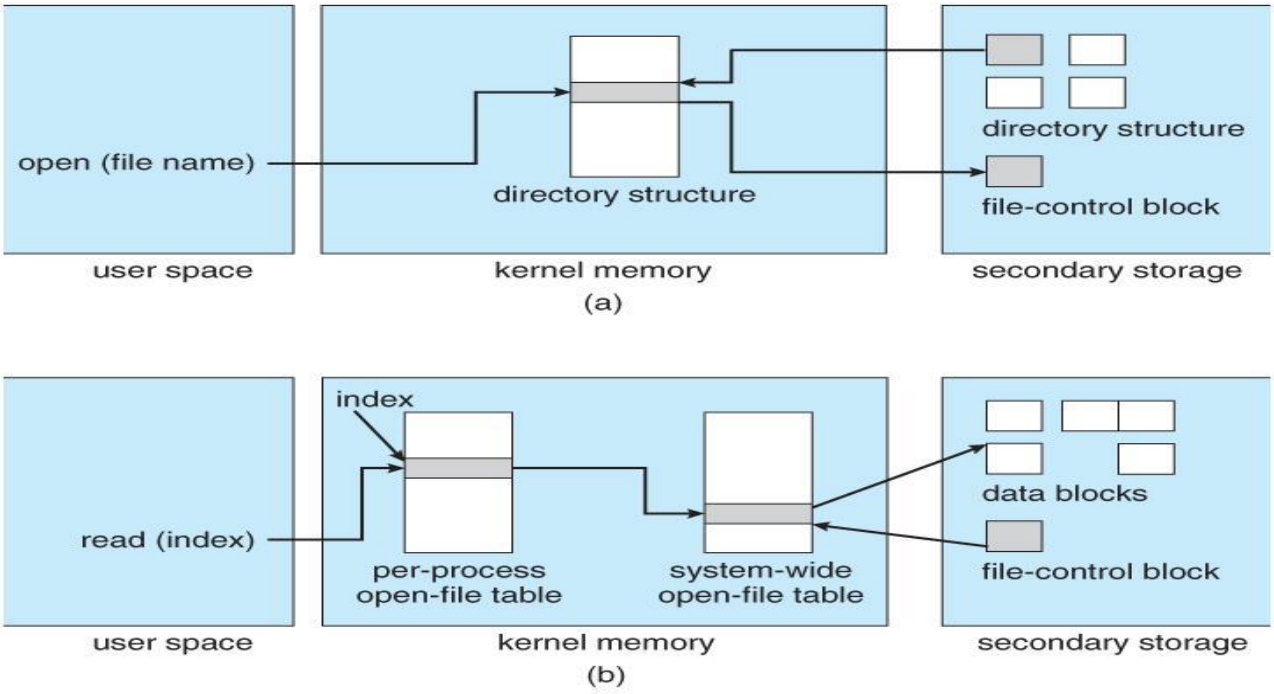


Figure 12.2 A typical file-control block.

- Once a file has been created, it can be used for I/O. First, though, it must be opened. The open() call passes a file name to the logical file system. The open() system call first searches the system-wide open-file table to see if the file is already in use by another process.
- If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. If the file is not already open, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory.
- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open.
- When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.



In-memory file-system structures. (a) File open. (b) File read.

Partitions and Mounting

- A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks. Each partition can be either "raw," containing no file system, or "cooked," containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format. Rather, boot information is usually a sequential series of blocks, loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte.
- This **boot loader** in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing. It can contain more than the instructions for how to boot a specific operating system. For instance, many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the disk.
- The **root partition**, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system. As part of a successful mount operation, 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.

Virtual File Systems

- An optimal method of implementing multiple types of file systems is to write directory and file routines for each type. Most operating systems use object oriented techniques to simplify, organize and modularize the implementation. Data structures and procedures are used to isolate the basic system call functionality from the implementation details. Thus, file system implementation consists of three major layers –

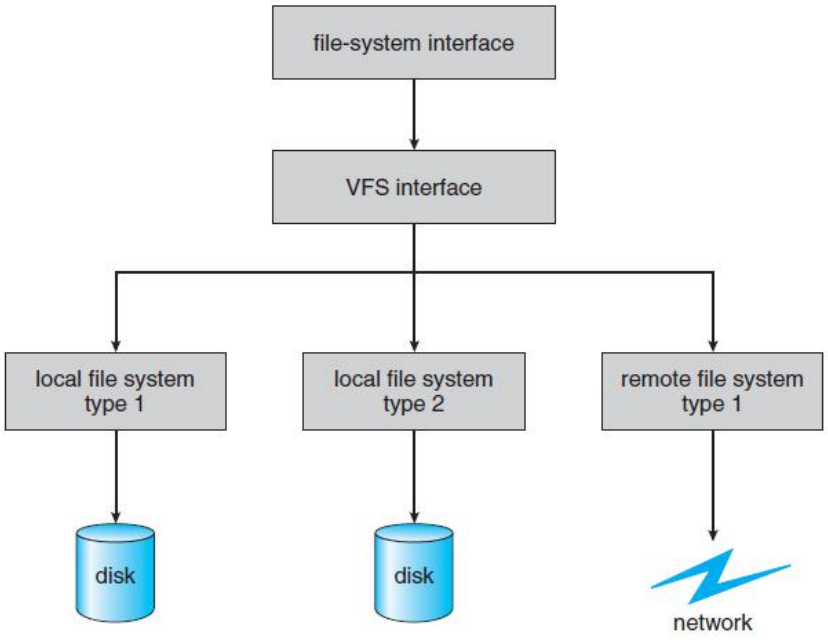


Figure 12.4 Schematic view of a virtual file system.

- The first layer is the file system interface based on system calls and on file descriptors.
- The second layer is called virtual file system layer which serves two important functions:
  - Separates file system generic operations from their implementation by defining a clean VFS interface.
  - VFS provides a mechanism for uniquely representing a file throughout a network. VFS is based on a file representation structure called vnode that contains a numerical designator for a network wide unique file.

    Thus, VFS distinguishes local files from remote ones and local files are further distinguished according to their file system types.
- The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

The four main object types defined by the Linux VFS are:

- The **inode object**, which represents an individual file
- The **file object**, which represents an open file
- The **superblock object**, which represents an entire file system
- The **dentry object**, which represents an individual directory entry

API for some of the operations for the file object includes:

- int open(. . .)—Open a file.
- int close(...)—Close an already-open file.
- ssize t read(. . .)—Read from a file.
- ssize t write(. . .)—Write to a file.
- int mmap(. . .)—Memory-map a file.

Directory Implementation

The selection of directory allocation and directory management algorithms significantly affects the efficiency, performance and reliability of the file system.

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time consuming to execute. The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.

Hash Table

- Another data structure used for a file directory is a **hash table.** With this method, a linear list stores the directory entries but a hash data structure is also used. The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

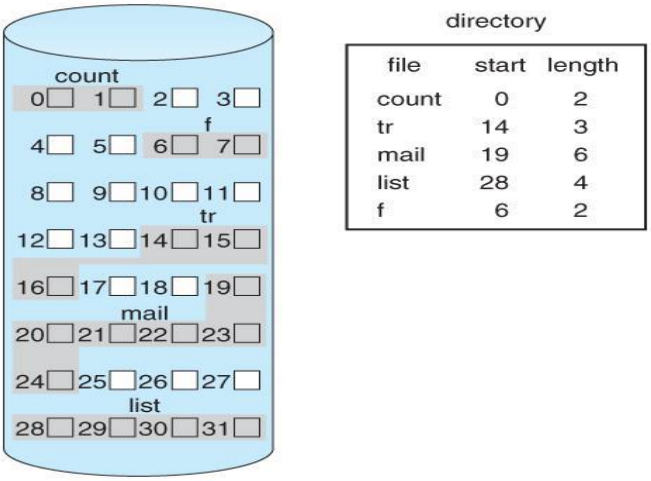
Allocation Methods

The direct access nature of disks allows flexibility in the implementation of files. The main problem here 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:

i. Contiguous
ii. Linked
iii. Indexed

Contiguous Allocation

- This allocation requires that each file occupy a set of contiguous blocks on the disk. The number of disk seeks required for accessing contiguously allocated files is minimal. Contiguous allocation of a file is defined by the disk address and length of the first block.
- Accessing a file that has been contiguously allocated is easy. Both sequential and direct access can be supported by contiguous allocation.



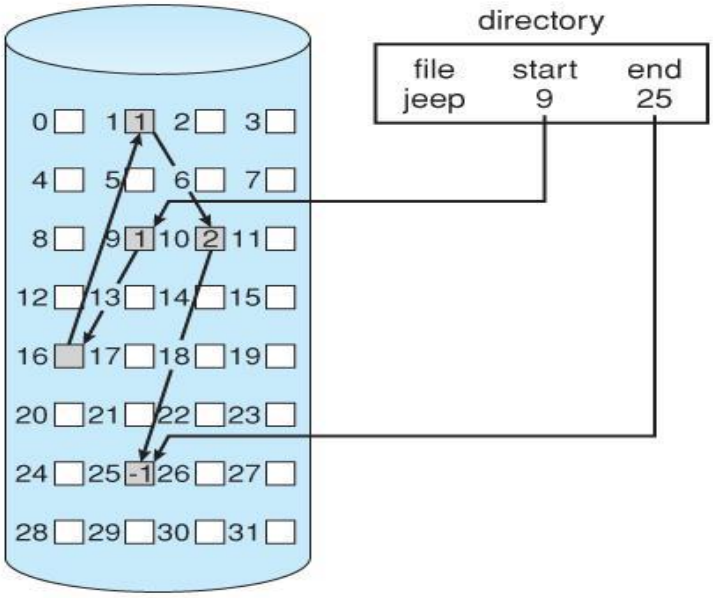
- Disadvantage is finding space for a new file. This problem can be seen as a particular application of general dynamic storage allocation problem which involves how to satisfy a request of size n form a list of free holes. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
- These algorithms suffer from **external fragmentation**. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. For solving the fragmentation problem, **compact** all free space into one contiguous space.
- Another problem with contiguous allocation is determining how much space is needed for a file. Pre allocation of memory space to a file may be insufficient. A file may be allocated space for its final size but large amount of that space will remain unused for a long time. The file therefore has a large amount of **internal fragmentation.**
- To minimize these drawbacks, some operating systems use a modified contiguous allocation scheme. Here a contiguous chunk of space is allocated initially and if that amount proves not to be large enough another chunk of contiguous space called **extent** is added. Internal fragmentation can still be a problem if the extents are too large and external fragmentation can become a problem as extents of varying sizes are allocated and de allocated.

Linked Allocation

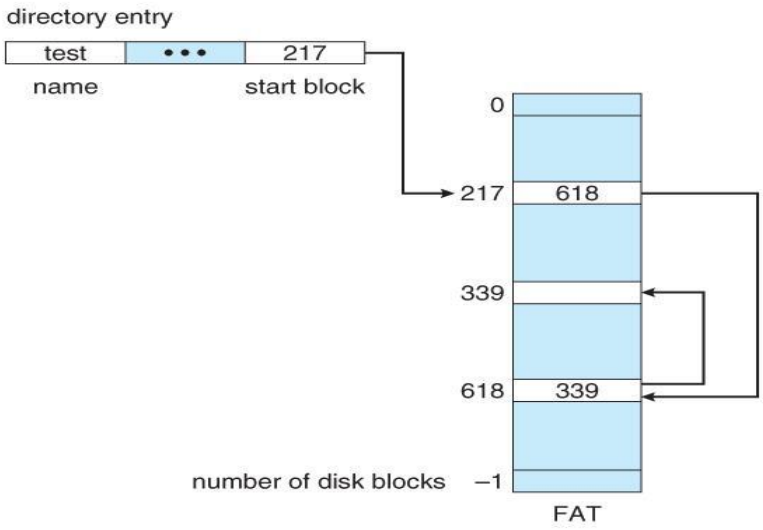
- This solves all problems of contiguous allocation. Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4

bytes, then the user sees blocks of 508 bytes.

- There is no external fragmentation with linked allocation and any free block on the free space list can be used to satisfy a request.
- But the major problem is that it can be used effectively only for sequential access files. It is inefficient to support a direct access capability for linked allocation files. Another disadvantage is space required for pointers.



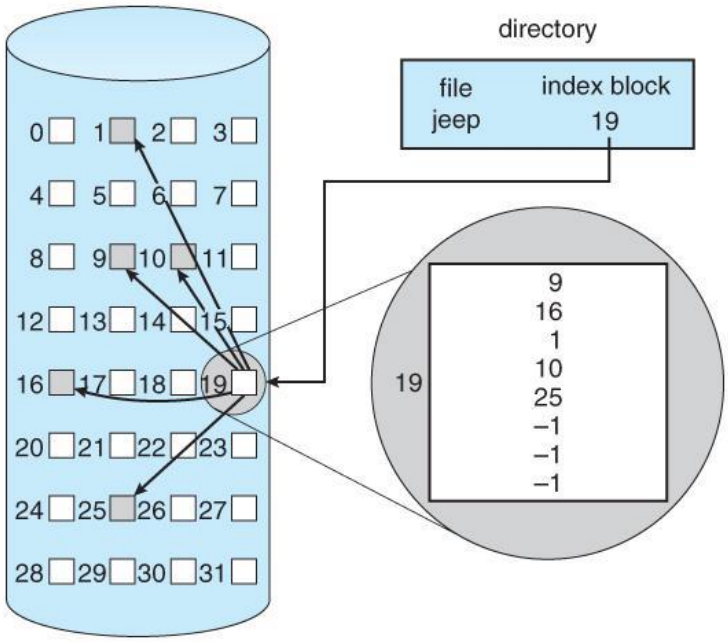
- Solution to this problem is to collect blocks into multiples called **clusters** and to allocate clusters rather than blocks. This method allows logical to physical block mapping to remain simple but improved disk through put and decreases the space needed for block allocation and free list management. This increases internal fragmentation because more space is wasted when a cluster is partially full than when a block is partially full.
- Another problem of linked allocation is reliability. 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. One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block.
- An important variation of 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 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.

Indexed Allocation

- Linked allocation solves external fragmentation and size declaration problems of contiguous allocation. In the absence of 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 pointers together into one location – index block. Each file has its own index block which is an array of disk block addresses.



- Indexed allocation supports direct access without suffering from external fragmentation because any free block on the disk can satisfy a request for more space.
- But indexed allocation suffers from wasted space. Every file must have an index block so it should be as small as possible. But if it is too small, it will not be able to hold enough pointers for a large file and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include –

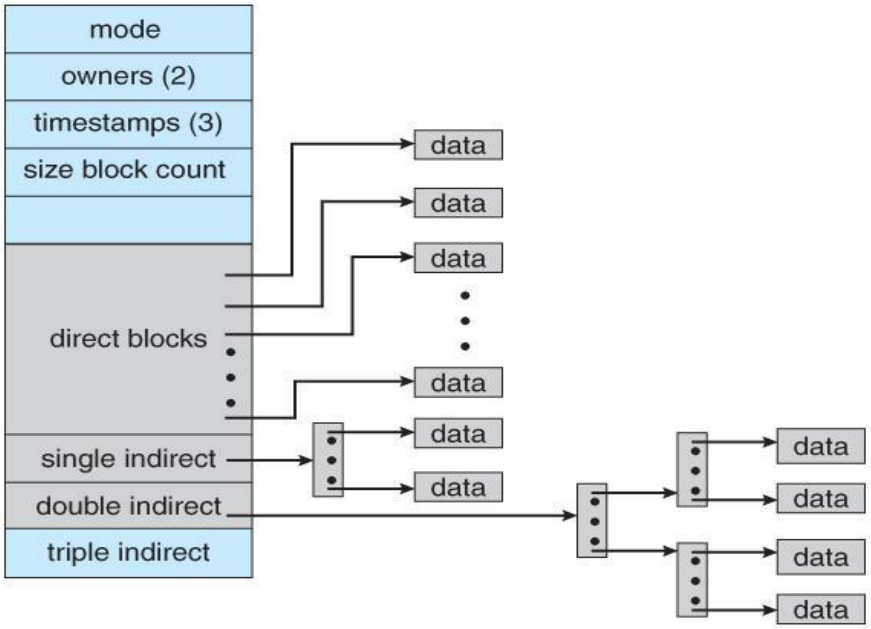
1. Linked scheme 2. Multilevel index 3. Combined scheme

Linked Scheme - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

Multi-Level Index - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.

Combined Scheme - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below.) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)

- Indexed allocation scheme suffers from some of the same performance problems as does linked allocation.



Performance

- The allocation methods vary in their storage efficiency and data block access times. Both are important in selecting the proper method for an operating system to implement. Before selecting an allocation method, determine how systems will be used.
- For any type of access, contiguous allocation requires only one access to get a disk block. For linked allocation, we can keep the address of the next block in memory and read it directly. This method is fine for sequential access. Hence some systems support direct access files by using contiguous allocation and sequential access by linked allocation.

Free Space Management

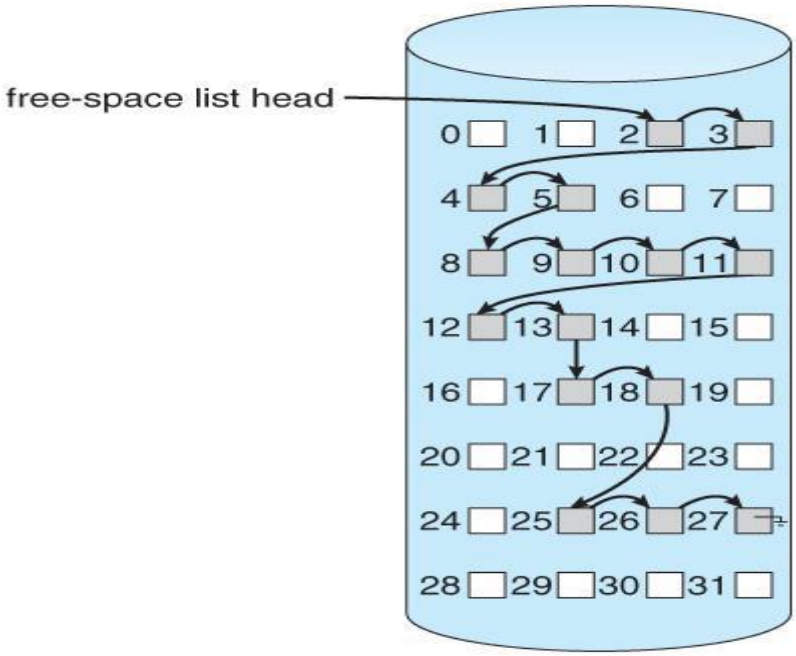
- Since disk space is limited, we should reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a **free space list**. The free space list records all free disk blocks – those not allocated to some file or directory. This free space list can be implemented as one of the following:

a) **Bit vector** – free space list is implemented as a bit map or a bit vector. Each block is represented by one bit. If the block is free, bit is 1, if the block is allocated, bit is 0.

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. The calculation of the block number is

(Number of bits per word) * (number of 0-value words) + offset of first 1 bit

b) **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. The first block contains a pointer to the next free disk block.



c) **Grouping** – A modification of the free list approach is to store the addresses of n free blocks in the first free block.

d) **Counting** – Another approach is to take advantage of the fact that several contiguous blocks may be allocated or freed simultaneously when space is allocated with the contiguous allocation algorithm or clustering.

e) **Space Maps -** Oracle's **ZFS** file system (found in Solaris and other operating systems) was designed to encompass huge numbers of files, directories, and even file systems. In its management of free space, ZFS uses a combination of techniques to control the size of data structures and minimize the I/O needed to manage those structures. First, ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks. The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log into that structure.

Questions from Previous Papers

1 A) What is file? What are the different types of files?
B) What are the attributes of a file? What is their significance?
2.Explain directory structure.
3.A) What are the different types of operations that can be performed on a directory?
B) Compare single level directory with two level directories.
4. Explain the tree structured directories along with their merits and demerits.
5.A) What are the approaches for sharing of files? Give their relative merits and demerits.
B) How dangling pointer problem occurs when deleting a file? What is the solution for it?
C) How garbage collection is used in the context of file system?
6.A) What is file structure? How file structure is supported by different operating systems?
B) What are the different types of file access methods?
7.A) What are the various operations that can be performed on the file?
B) What are the various pieces of information associated with an open file?

8. A) File system is the most visible aspect of an operating system. Discuss.
B) A file is an abstract data type. Discuss.

9. a) What is dual booting?
b) How multiple file systems are interested into a directory structures?
10. Explain different allocation methods for disk space.
11. What are the structures and operations that are used to implement file system operations?
12. What is mounting of a file system, how mounting takes place in different operating systems.
13. a) What is search path, absolute path and relative path?
b) What is directory structure which is suitable for sharing of files? Justify your answer. Illustrate with an example.
14. Explain file sharing in different types of systems.
15. Explain the following terms with respect to disk:
(i) Seek time (ii) Rotational latency (iii) Bandwidth

16. Give a detailed note on RAID levels.
17. Explain the following terms with respect to a magnetic disk.
(i) Transfer rate.

(ii) Random access time.

(iii) Head crash.
18. Discuss the problems with RAID.
19. Give a note on selection of a disk scheduling algorithm.

20. Compare the performance of write operations achieved by a RAID level 5 organization with that achieved by a RAID level 1 organization.
21. What are the goals of achieving parallelism in a disk system through stripping?
22. Explain the differences between SCAN, C-SCAN, LOOK, and C-LOOK disk scheduling algorithms with example.

Protection

14.1 Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

14.2 Principles of Protection

- The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

14.3 Domain of Protection

- A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).
- The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.

14.3.1 Domain Structure

- A ***protection domain*** specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An ***access right*** is the ability to execute an operation on an object.
- A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.

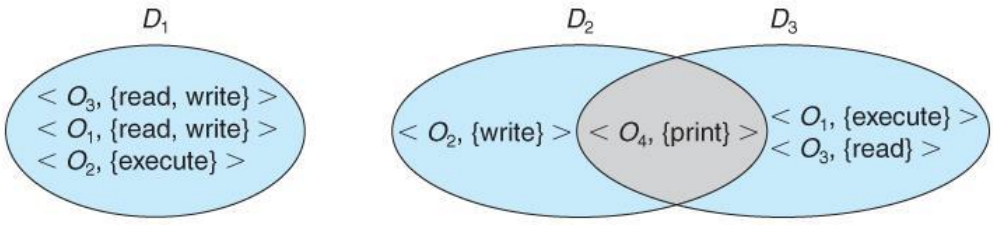


Figure 14.1 - System with three protection domains.

- The association between a process and a domain may be *static* or *dynamic*.

    - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
    - If the association is dynamic, then there needs to be a mechanism for ***domain switching*.**
- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

14.3.2 An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.
- Yet another alternative is to not allow the changing of ID at all. Instead, special privileged daemons are launched at boot time, and user processes send messages to these daemons when they need special tasks performed.

14.3.3 An Example: MULTICS

- The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:

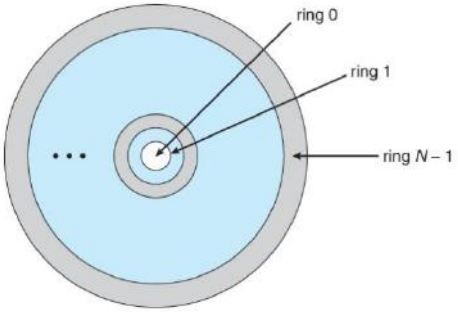


Figure 14.2 - MULTICS ring structure.

- Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
- Each file is a memory segment, and each segment description includes an entry that indicates the ring number associated with that segment, as well as read, write, and execute privileges.
- Each process runs in a ring, according to the *current-ring-number*, a counter associated with each process.
- A process operating in one ring can only access segments associated with higher (farther out) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.
- Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:
    - An ***access bracket***, defined by integers b1 <= b2.
    - A ***limit*** b3 > b2
    - A ***list of gates,*** identifying the entry points at which the segments may be called.
- If a process operating in ring i calls a segment whose bracket is such that b1 <= i <= b2, then the call succeeds and the process remains in ring i.
- Otherwise a trap to the OS occurs, and is handled as follows:

    - If i < b1, then the call is allowed, because we are transferring to a procedure with fewer privileges. However if any of the parameters being passed are of segments below b1, then they must be copied to an area accessible by the called procedure.
    - If i > b2, then the call is allowed only if i <= b3 and the call is directed to one of the entries on the list of gates.
- Overall this approach is more complex and less efficient than other protection schemes.

14.4 Access Matrix

- The model of protection that we have been discussing can be viewed as an ***access matrix,*** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure 14.3 - Access matrix.

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure 14.4 - Access matrix of Figure 14.3 with domains as objects.

- The ability to ***copy*** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:
    - If the asterisk is removed from the original access right, then the right is ***transferred,*** rather than being copied. This may be termed a ***transfer*** right as opposed to a ***copy*** right.
    - If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a ***limited copy*** right, as shown in Figure 14.5 below:

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure 14.5 - Access matrix with *copy* rights.

- The ***owner*** right adds the privilege of adding new rights or removing existing ones:

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure 14.6 - Access matrix with *owner* rights.

- Copy and owner rights only allow the modification of rights within a column. The addition of ***control rights***, which only apply to domain objects, allow a process operating in one domain to affect

the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure 14.7 - Modified access matrix of Figure 14.4

14.5 Implementation of Access Matrix

14.5.1 Global Table

- The simplest approach is one big global table with < domain, object, rights > entries.
- Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques.)
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

14.5.2 Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

14.5.3 Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
    - A ***tag,*** possibly hardware implemented, distinguishing this special type of data. (other types may be floats, pointers, booleans, etc.)
    - The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by the operating system for maintaining the process's access right capability list.

14.5.4 A Lock-Key Mechanism

- Each resource has a list of unique bit patterns, termed locks.
- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

14.5.5 Comparison

- Each of the methods here has certain advantages or disadvantages, depending on the particular situation and task at hand.
- Many systems employ some combination of the listed methods.

14.6 Access Control

- ***Role-Based Access Control, RBAC,*** assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls.
- RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to SUID or SGID programs.

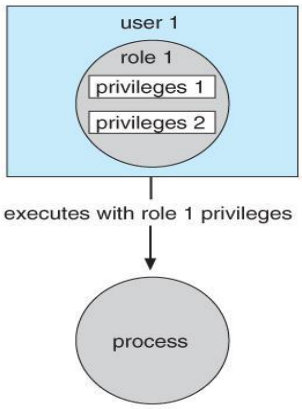


Figure 14.8 - Role-based access control in Solaris 10.

14.7 Revocation of Access Rights

- The need to revoke access rights dynamically raises several questions:
  - Immediate versus delayed - If delayed, can we determine when the revocation will take place?
  - Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
  - Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
  - Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?
- With an access list scheme revocation is easy, immediate, and can be selective, general, partial, total, temporary, or permanent, as desired.
- With capabilities lists the problem is more complicated, because access rights are distributed throughout the system. A few schemes that have been developed include:
  - Reacquisition - Capabilities are periodically revoked from each domain, which must then re-acquire them.
  - Back-pointers - A list of pointers is maintained from each object to each capability which is held for that object.
  - Indirection - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry, which may affect multiple processes, which must then re-acquire access rights to continue.
  - Keys - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process.
    - A master key is associated with each object.
    - When a capability is created, its key is set to the object's master key.
    - As long as the capability's key matches the object's key, then the capabilities remain valid.

        - The object master key can be changed with the set-key command, thereby invalidating all current capabilities.
        - More flexibility can be added to this scheme by implementing a ***list*** of keys for each object, possibly in a global table.

14.8 Capability-Based Systems (Optional)

14.8.1 An Example: Hydra

- Hydra is a capability-based system that includes both system-defined ***rights*** and user-defined rights. The interpretation of user-defined rights is up to the specific user programs, but the OS provides support for protecting access to those rights, whatever they may be
- Operations on objects are defined procedurally, and those procedures are themselves protected objects, accessed indirectly through capabilities.
- The names of user-defined procedures must be identified to the protection system if it is to deal with user-defined rights.
- When an object is created, the names of operations defined on that object become ***auxiliary rights,*** described in a capability for an ***instance*** of the type. For a process to act on an object, the capabilities it holds for that object must contain the name of the operation being invoked. This allows access to be controlled on an instance-by-instance and process-by-process basis.
- Hydra also allows ***rights amplification,*** in which a process is deemed to be ***trustworthy,*** and thereby allowed to act on any object corresponding to its parameters.
- Programmers can make direct use of the Hydra protection system, using suitable libraries which are documented in appropriate reference manuals.

14.8.2 An Example: Cambridge CAP System

- The CAP system has two kinds of capabilities:
    - ***Data capability,*** used to provide read, write, and execute access to objects. These capabilities are interpreted by microcode in the CAP machine.
    - ***Software capability,*** is protected but not interpreted by the CAP microcode.
        - Software capabilities are interpreted by protected (privileged) procedures, possibly written by application programmers.
        - When a process executes a protected procedure, it temporarily gains the ability to read or write the contents of a software capability.
        - This leaves the interpretation of the software capabilities up to the individual subsystems, and limits the potential damage that could be caused by a faulty privileged procedure.
        - Note, however, that protected procedures only get access to software capabilities for the subsystem of which they are a part. Checks are made when passing software capabilities to protected procedures that they are of the correct type.
        - Unfortunately the CAP system does not provide libraries, making it harder for an individual programmer to use than the Hydra system.

14.9 Language-Based Protection (Optional)

- As systems have developed, protection systems have become more powerful, and also more specific and specialized.
- To refine protection even further requires putting protection capabilities into the hands of individual programmers, so that protection policies can be implemented on the application level, i.e. to protect resources in ways that are known to the specific applications but not to the more general operating system.

14.9.1 Compiler-Based Enforcement

- In a compiler-based approach to protection enforcement, programmers directly specify the protection needed for different resources at the time the resources are declared.
- This approach has several advantages:
    1. Protection needs are simply declared, as opposed to a complex series of procedure calls.
    2. Protection requirements can be stated independently of the support provided by a particular OS.
    3. The means of enforcement need not be provided directly by the developer.
    4. Declarative notation is natural, because access privileges are closely related to the concept of data types.
- Regardless of the means of implementation, compiler-based protection relies upon the underlying protection mechanisms provided by the underlying OS, such as the Cambridge CAP or Hydra systems.
- Even if the underlying OS does not provide advanced protection mechanisms, the compiler can still offer some protection, such as treating memory accesses differently in code versus data segments. (E.g. code segments cant be modified, data segments can't be executed.)
- There are several areas in which compiler-based protection can be compared to kernel-enforced protection:
    - **Security.** Security provided by the kernel offers better protection than that provided by a compiler. The security of the compiler-based enforcement is dependent upon the integrity of the compiler itself, as well as requiring that files not be modified after they are compiled. The kernel is in a better position to protect itself from modification, as well as protecting access to specific files. Where hardware support of individual memory accesses is available, the protection is stronger still.
    - **Flexibility.** A kernel-based protection system is not as flexible to provide the specific protection needed by an individual programmer, though it may provide support which the programmer may make use of. Compilers are more easily changed and updated when necessary to change the protection services offered or their implementation.
    - **Efficiency.** The most efficient protection mechanism is one supported by hardware and microcode. Insofar as software based protection is concerned, compiler-based systems have the advantage that many checks can be made off-line, at compile time, rather that during execution.
- The concept of incorporating protection mechanisms into programming languages is in its infancy, and still remains to be fully developed. However the general goal is to provide mechanisms for three functions:
    0. Distributing capabilities safely and efficiently among customer processes. In particular a user process should only be able to access resources for which it was issued capabilities.
    1. Specifying the ***type*** of operations a process may execute on a resource, such as reading or writing.
    2. Specifying the ***order*** in which operations are performed on the resource, such as opening before reading.

14.9.2 Protection in Java

- Java was designed from the very beginning to operate in a distributed environment, where code would be executed from a variety of trusted and untrusted sources. As a result the Java Virtual Machine, JVM incorporates many protection mechanisms
- When a Java program runs, it load up classes dynamically, in response to requests to instantiates objects of particular types. These classes may come from a variety of different sources, some trusted and some not, which requires that the protection mechanism be implemented at the resolution of individual classes, something not supported by the basic operating system.

- As each class is loaded, it is placed into a separate protection domain. The capabilities of each domain depend upon whether the source URL is trusted or not, the presence or absence of any digital signatures on the class (Chapter 15), and a configurable policy file indicating which servers a particular user trusts, etc.
- When a request is made to access a restricted resource in Java, (e.g. open a local file), some process on the current ***call stack*** must specifically assert a privilege to perform the operation. In essence this method ***assumes responsibility*** for the restricted access. Naturally the method must be part of a class which resides in a protection domain that includes the capability for the requested operation. This approach is termed ***stack inspection,*** and works like this:
    - When a caller may not be trusted, a method executes an access request within a `doPrivileged( )` block, which is noted on the calling stack.
    - When access to a protected resource is requested, `checkPermissions( )` inspects the call stack to see if a method has asserted the privilege to access the protected resource.
        - If a suitable doPriveleged block is encountered on the stack before a domain in which the privilege is disallowed, then the request is granted.
        - If a domain in which the request is disallowed is encountered first, then the access is denied and a AccessControlException is thrown.
        - If neither is encountered, then the response is implementation dependent.
- In the example below the untrusted applet's call to `get( )` succeeds, because the trusted URL loader asserts the privilege of opening the specific URL lucent.com. However when the applet tries to make a direct call to `open( )` it fails, because it does not have privilege to access any sockets.

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission (a, connect);
			connect (a);
			...

Figure 14.9 - Stack inspection.

Security

15.1 The Security Problem

- This chapter (Security) deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.
- Some of the most common types of ***violations*** include:
    - ***Breach of Confidentiality*** - Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.
    - ***Breach of Integrity*** - Unauthorized ***modification*** of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
    - ***Breach of Availability*** - Unauthorized ***destruction*** of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.
    - ***Theft of Service*** - Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.
    - ***Denial of Service, DOS*** - Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.
- One common attack is ***masquerading,*** in which the attacker pretends to be a trusted third party. A variation of this is the ***man-in-the-middle,*** in which the attacker masquerades as both ends of the conversation to two targets.
- A ***replay attack*** involves repeating a valid transmission. Sometimes this can be the entire attack, (such as repeating a request for a money transfer), or other times the content of the original message is replaced with malicious content.

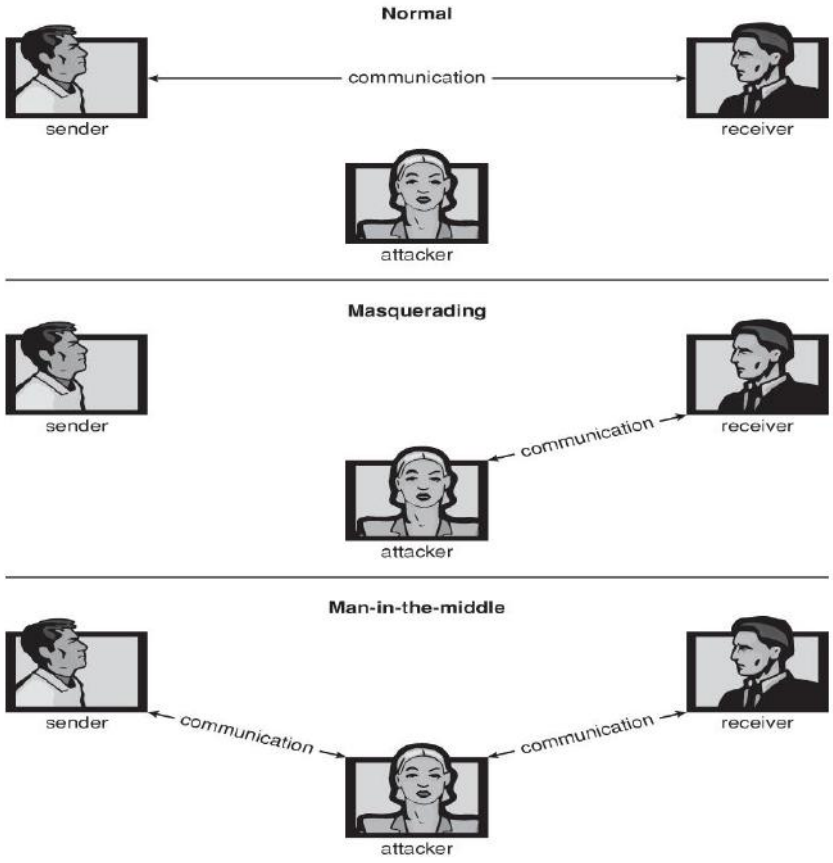


Figure 15.1 - Standard security attacks.

- There are four levels at which a system must be protected:
    1. **Physical -** The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
    2. **Human -** There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via ***social engineering,*** which basically means fooling trustworthy people into accidentally breaching security.
        - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.
        - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. (Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)
        - **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary (in any language), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password.)
    3. **Operating System -** The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.
    4. **Network -** As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

15.2 Program Threats

- There are many common threats to modern systems. Only a few are discussed here.

15.2.1 Trojan Horse

- A ***Trojan Horse*** is a program that secretly performs some maliciousness in addition to its visible actions.
- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with ***viruses,*** (see below.)
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory (".") as part of the path. If a dangerous program having the same name as a legitimate program (or a common mis-spelling, such as "sl" instead of "ls") is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again (with a proper login prompt), logs in successfully, and doesn't realize that their information has been stolen.
- **(Special Note to UIC students: Beware that someone has registered the domain name of uic.EU (without the "D"), and is running an ssh server which will accept requests to any machine in the domain, and gladly accept your login and password information,**

without, of course, actually logging you in. Access to this site is blocked from campus, but you are on your own off campus.)

- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such as Control-Alt-Delete in order to log in. (This is why modern Windows systems require the Control-Alt-Delete sequence to commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system.)
- ***Spyware*** is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. (This is an example of ***covert channels,*** in which surreptitious communications occur.) Another common task of spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

15.2.2 Trap Door

- A ***Trap Door*** is when a designer or a programmer (or hacker) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

15.2.3 Logic Bomb

- A ***Logic Bomb*** is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.
- A classic example is the ***Dead-Man Switch***, which is designed to check whether a certain person (e.g. the author) is logging in every day, and if they don't log in for a long time (presumably because they've been fired), then the logic bomb goes off and either opens up security holes or causes other problems.

15.2.4 Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[1] exceeds 256 characters:
    - The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
    - (The problem could be avoided using str***n***cpy, with a limit of 255 characters copied plus room for the null byte.)

```
#include
#define BUFFER_SIZE 256
      int main( int argc, char * argv[ ] )
{
    char buffer[ BUFFER_SIZE ];
    if( argc < 2 )
        return -1;
    else {
        strcpy( buffer, argv[ 1 ] );
        return 0;
    }
}
```

Figure 15.2 - C program with buffer-overflow condition.

- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
    - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
    - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses. (I.e. an array "grows" towards the bottom of the stack.
    - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.

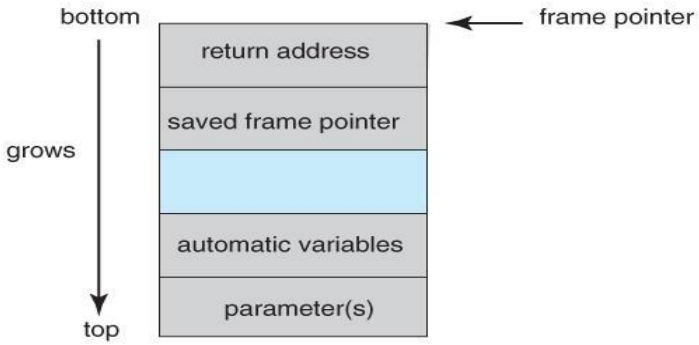


Figure 15.3 - The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec(/bin/sh)". To do this requires compiling a program that contains this instruction, and then using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.
- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location. (Note, however, that neither the bad code or the padding can contain null bytes, which would terminate the strcpy.)
- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.

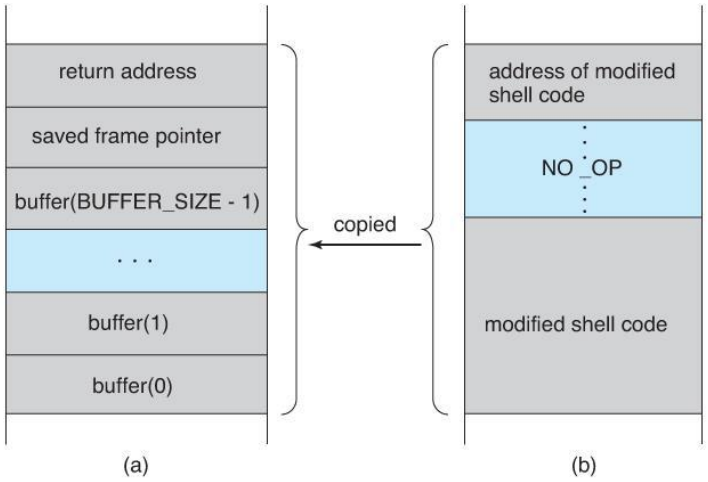


Figure 15.4 - Hypothetical stack frame for Figure 15.2, (a) before and (b) after.

- Unfortunately famous hacks such as the buffer overflow attack are well published and well known, and it doesn't take a lot of skill to follow the instructions and start attacking lots of systems until the law of averages eventually works out. (***Script Kiddies*** are those hackers with only rudimentary skills of their own but the ability to copy the efforts of others.)
- Fortunately modern hardware now includes a bit in the page tables to mark certain pages as non-executable. In this case the buffer-overflow attack would work up to a point, but as soon as it "returns" to an address in the data space and tries executing statements there, an exception would be thrown crashing the program.
- (More details about stack-overflow attacks are available on-line from http://www.insecure.org/stf/smashstack.txt)

15.2.5 Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a ***virus dropper,*** usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- Viruses take many forms (see below.) Figure 15.5 shows typical operation of a boot sector virus:

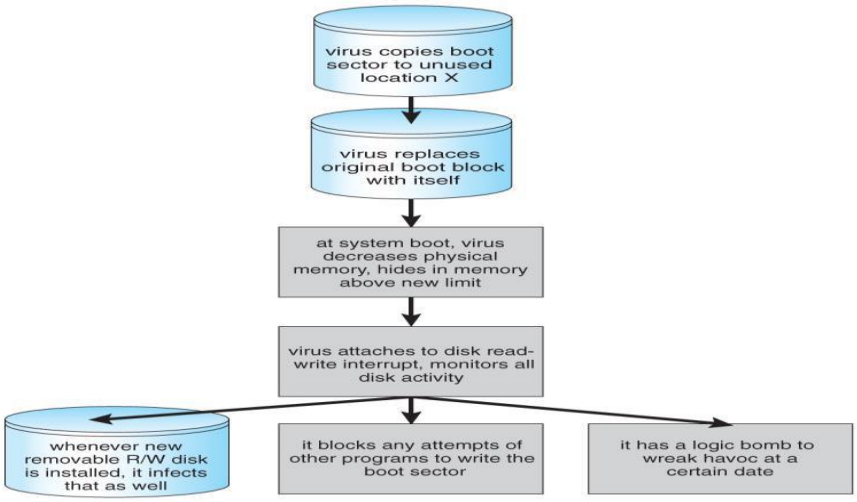


Figure 15.5 - A boot-sector computer virus.

- Some of the forms of viruses include:
  - **File -** A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed ***parasitic,*** because they do not leave any new files on the system, and the original program is still fully functional.
  - **Boot -** A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as ***memory viruses***, because in operation they reside in memory, and do not appear in the file system.
  - **Macro -** These viruses exist as a macro (script) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
  - **Source code** viruses look for source code and infect it in order to spread.
  - **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their ***signature,*** by which virus checkers recognize them.
  - **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.

    - **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read( ) system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.
    - **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
    - **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
    - **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".
- In 2004 a virus exploited three bugs in Microsoft products to infect hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server, which in turn infected any Microsoft Internet Explorer web browser that visited any of the infected server sites. One of the back-door programs it installed was a ***keystroke logger,*** which records users keystrokes, including passwords and other sensitive information.
- There is some debate in the computing community as to whether a ***monoculture,*** in which nearly all systems run the same hardware, operating system, and applications, increases the threat of viruses and the potential for harm caused by them.

15.3 System and Network Threats

- Most of the threats described above are termed ***program threats***, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

15.3.1 Worms

- A ***worm*** is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
    1. A small program called a ***grappling hook,*** which was deposited on the target system through one of three vulnerabilities, and
    2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

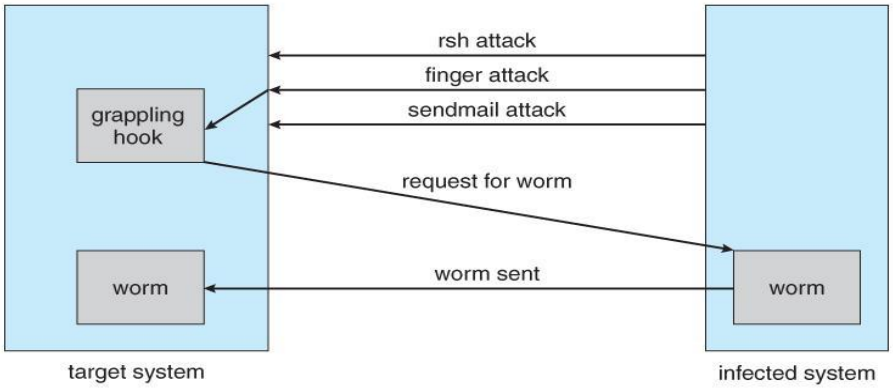


Figure 15.6 - The Morris Internet worm.

- The three vulnerabilities exploited by the Morris Internet worm were as follows:
    1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that ***any*** user (except root) on system A could access the same account on system B without providing a password.
    2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.
    3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.
- Once in place, the worm undertook systematic attacks to discover user passwords:
    1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
    2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
    3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.
- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

15.3.2 Port Scanning

- ***Port Scanning*** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.
- Because port scanning is easily detected and traced, it is usually launched from ***zombie systems,*** i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.

- There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are ***nmap*** (http://www.insecure.org/nmap) and ***nessus*** (http://www.nessus.org). The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

15.3.3 Denial of Service

- ***Denial of Service (DOS)*** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.
- Sometimes DOS is not the result of deliberate maliciousness. Consider for example:
    - A web site that sees a huge volume of hits as a result of a successful advertising campaign.
    - CNN.com occasionally gets overwhelmed on big news days, such as Sept 11, 2001.
    - CS students given their first programming assignment involving fork( ) often quickly fill up process tables or otherwise completely consume system resources. :-)
    - (Please use ipcs and ipcrm when working on the inter-process communications assignment !)

15.4 Cryptography as a Security Tool

- Within a given computer the transmittal of messages is safe, reliable and secure, because the OS knows exactly where each one is coming from and where it is going.
- On a network, however, things aren't so straightforward - A rogue computer (or e-mail sender) may spoof their identity, and outgoing packets are delivered to a lot of other computers besides their (intended) final destination, which brings up two big questions of security:
    - **Trust -** How can the system be sure that the messages received are really from the source that they say they are, and can that source be trusted?
    - **Confidentiality -** How can one ensure that the messages one is sending are received only by the intended recipient?
- Cryptography can help with both of these problems, through a system of **secrets** and **keys.** In the former case, the key is held by the sender, so that the recipient knows that only the authentic author could have sent the message; In the latter, the key is held by the recipient, so that only the intended recipient can receive the message accurately.
- Keys are designed so that they cannot be divined from any public information, and must be guarded carefully. (***Asymmetric encryption*** involve both a public and a private key.)

15.4.1 Encryption

- The basic idea of encryption is to encode a message so that only the desired recipient can decode and read it. Encryption has been around since before the days of Caesar, and is an entire field of study in itself. Only some of the more significant computer encryption schemes will be covered here.

- The basic process of encryption is shown in Figure 15.7, and will form the basis of most of our discussion on encryption. The steps in the procedure and some of the key terminology are as follows:
    1. The **sender** first creates a **message, m** in plaintext.
    2. The message is then entered into an **encryption algorithm, E,** along with the **encryption key, Ke.**
    3. The encryption algorithm generates the **ciphertext, c, = E(Ke)(m).** For any key k, E(k) is an algorithm for generating ciphertext from a message, and both E and E(k) should be efficiently computable functions.
    4. The ciphertext can then be sent over an unsecure network, where it may be received by **attackers.**
    5. The **recipient** enters the ciphertext into a **decryption algorithm, D,** along with the **decryption key, Kd.**
    6. The decryption algorithm re-generates the plaintext message, m, = D(Kd)(c). For any key k, D(k) is an algorithm for generating a clear text message from a ciphertext, and both D and D(k) should be efficiently computable functions.
    7. The algorithms described here must have this important property: Given a ciphertext c, a computer can only compute a message m such that c = E(k)(m) if it possesses D(k). (In other words, the messages can't be decoded unless you have the decryption algorithm and the decryption key.)

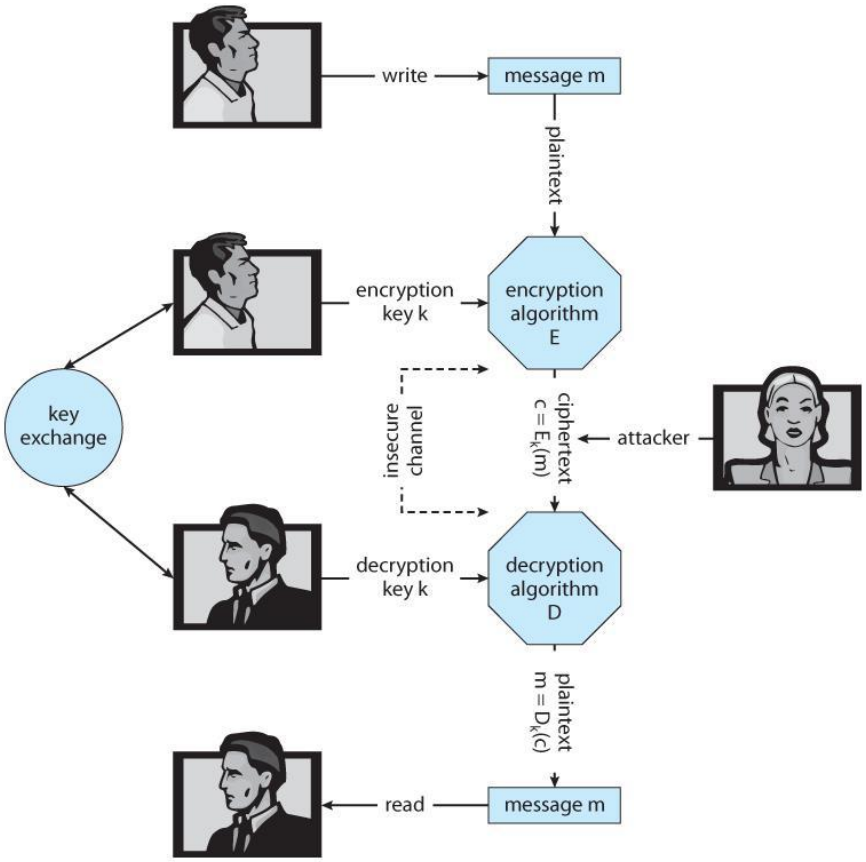


Figure 15.7 - A secure communication over an insecure medium.

15.4.1.1 Symmetric Encryption

- With ***symmetric encryption*** the same key is used for both encryption and decryption, and must be safely guarded. There are a number of well-known symmetric encryption algorithms that have been used for computer security:
    - The ***Data-Encryption Standard, DES,*** developed by the National Institute of Standards, NIST, has been a standard civilian encryption standard for over 20 years. Messages are broken down into 64-bit chunks, each of which are encrypted using a 56-bit key through a series of substitutions and

transformations. Some of the transformations are hidden (black boxes), and are classified by the U.S. government.
    - DES is known as a ***block cipher,*** because it works on blocks of data at a time. Unfortunately this is a vulnerability if the same key is used for an extended amount of data. Therefore an enhancement is to not only encrypt each block, but also to XOR it with the previous block, in a technique known as ***cipher-block chaining.***
    - As modern computers become faster and faster, the security of DES has decreased, to where it is now considered insecure because its keys can be exhaustively searched within a reasonable amount of computer time. An enhancement called ***triple DES*** encrypts the data three times using three separate keys (actually two encryptions and one decryption) for an effective key length of 168 bits. Triple DES is in widespread use today.
    - The ***Advanced Encryption Standard, AES,*** developed by NIST in 2001 to replace DES uses key lengths of 128, 192, or 256 bits, and encrypts in blocks of 128 bits using 10 to 14 rounds of transformations on a matrix formed from the block.
    - The ***twofish algorithm,*** uses variable key lengths up to 256 bits and works on 128 bit blocks.
    - ***RC5*** can vary in key length, block size, and the number of transformations, and runs on a wide variety of CPUs using only basic computations.
    - ***RC4*** is a ***stream cipher,*** meaning it acts on a stream of data rather than blocks. The key is used to seed a pseudo-random number generator, which generates a ***keystream*** of keys. RC4 is used in ***WEP,*** but has been found to be breakable in a reasonable amount of computer time.

15.4.1.2 Asymmetric Encryption

- With ***asymmetric encryption,*** the decryption key, Kd, is not the same as the encryption key, Ke, and more importantly cannot be derived from it, which means the encryption key can be made publicly available, and only the decryption key needs to be kept secret. (or vice-versa, depending on the application.)
- One of the most widely used asymmetric encryption algorithms is ***RSA,*** named after its developers - Rivest, Shamir, and Adleman.
- RSA is based on two large prime numbers, ***p*** and ***q,*** (on the order of 512 bits each), and their product ***N.***
    - Ke and Kd must satisfy the relationship:
      (Ke * Kd) % [(p - 1) * (q - 1)] = = 1
    - The encryption algorithm is:
      c = E(Ke)(m) = m^Ke % N
    - The decryption algorithm is:
      m = D(Kd)(c) = c^Kd % N
- An example using small numbers:
    - p = 7
    - q = 13
    - N = 7 * 13 = 91
    - (p - 1) * (q - 1) = 6 * 12 = 72
    - Select Ke < 72 and relatively prime to 72, say 5
    - Now select Kd, such that (Ke * Kd) % 72 = = 1, say 29
    - The public key is now (5, 91) and the private key is (29, 91)
    - Let the message, m = 42
    - Encrypt: c = 42^5 % 91 = 35
    - Decrypt: m = 35^29 % 91 = 42

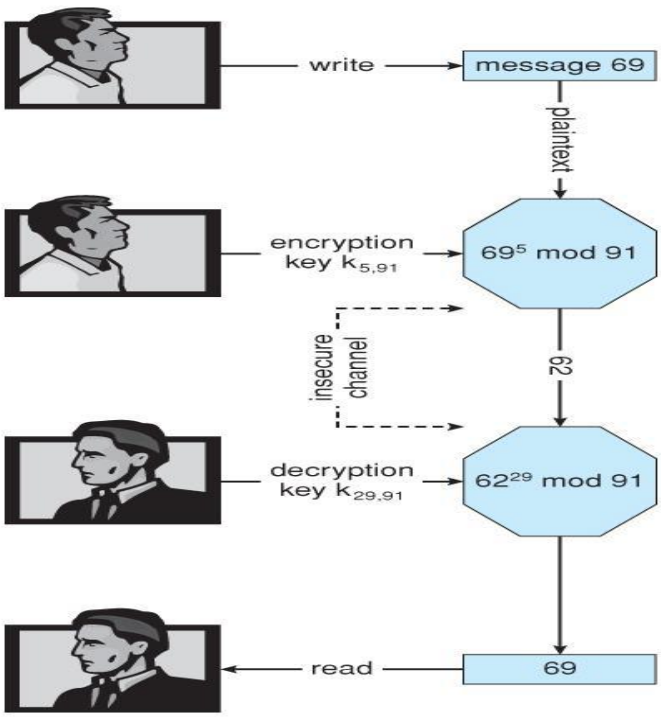


Figure 15.8 - Encryption and decryption using RSA asymmetric cryptography

- Note that asymmetric encryption is much more computationally expensive than symmetric encryption, and as such it is not normally used for large transmissions. Asymmetric encryption is suitable for small messages, authentication, and key distribution, as covered in the following sections.

15.4.1.3 Authentication

- Authentication involves verifying the identity of the entity who transmitted a message.
- For example, if D(Kd)(c) produces a valid message, then we know the sender was in possession of E(Ke).
- This form of authentication can also be used to verify that a message has not been modified
- Authentication revolves around two functions, used for ***signatures*** (or ***signing***), and ***verification:***
    - A signing function, ***S(Ks)*** that produces an ***authenticator, A,*** from any given message m.
    - A Verification function, ***V(Kv,m,A)*** that produces a value of "true" if A was created from m, and "false" otherwise.
    - Obviously S and V must both be computationally efficient.
    - More importantly, it must not be possible to generate a valid authenticator, A, without having possession of S(Ks).
    - Furthermore, it must not be possible to divine S(Ks) from the combination of (m and A), since both are sent visibly across networks.
- Understanding authenticators begins with an understanding of hash functions, which is the first step:
    - ***Hash functions, H(m)*** generate a small fixed-size block of data known as a ***message digest,*** or ***hash value*** from any given input data.
    - For authentication purposes, the hash function must be ***collision resistant on m.*** That is it should not be reasonably possible to find an alternate message m' such that H(m') = H(m).
    - Popular hash functions are **MD5,** which generates a 128-bit message digest, and **SHA-1,** which generates a 160-bit digest.

- Message digests are useful for detecting (accidentally) changed messages, but are not useful as authenticators, because if the hash function is known, then someone could easily change the message and then generate a new hash value for the modified message. Therefore authenticators take things one step further by encrypting the message digest.
- A ***message-authentication code, MAC,*** uses symmetric encryption and decryption of the message digest, which means that anyone capable of verifying an incoming message could also generate a new message.
- An asymmetric approach is the ***digital-signature algorithm,*** which produces authenticators called ***digital signatures.*** In this case Ks and Kv are separate, Kv is the public key, and it is not practical to determine S(Ks) from public information. In practice the sender of a message signs it (produces a digital signature using S(Ks)), and the receiver uses V(Kv) to verify that it did indeed come from a trusted source, and that it has not been modified.
- There are three good reasons for having separate algorithms for encryption of messages and authentication of messages:
    1. Authentication algorithms typically require fewer calculations, making verification a faster operation than encryption.
    2. Authenticators are almost always smaller than the messages, improving space efficiency. (?)
    3. Sometimes we want authentication only, and not confidentiality, such as when a vendor issues a new software patch.
- Another use of authentication is ***non-repudiation,*** in which a person filling out an electronic form cannot deny that they were the ones who did so.

15.4.1.4 Key Distribution

- Key distribution with symmetric cryptography is a major problem, because all keys must be kept secret, and they obviously can't be transmitted over unsecure channels. One option is to send them ***out-of-band,*** say via paper or a confidential conversation.
- Another problem with symmetric keys, is that a separate key must be maintained and used for each correspondent with whom one wishes to exchange confidential information.
- Asymmetric encryption solves some of these problems, because the public key can be freely transmitted through any channel, and the private key doesn't need to be transmitted anywhere. Recipients only need to maintain one private key for all incoming messages, though senders must maintain a separate public key for each recipient to which they might wish to send a message. Fortunately the public keys are not confidential, so this ***key-ring*** can be easily stored and managed.
- Unfortunately there are still some security concerns regarding the public keys used in asymmetric encryption. Consider for example the following man-in-the-middle attack involving phony public keys:

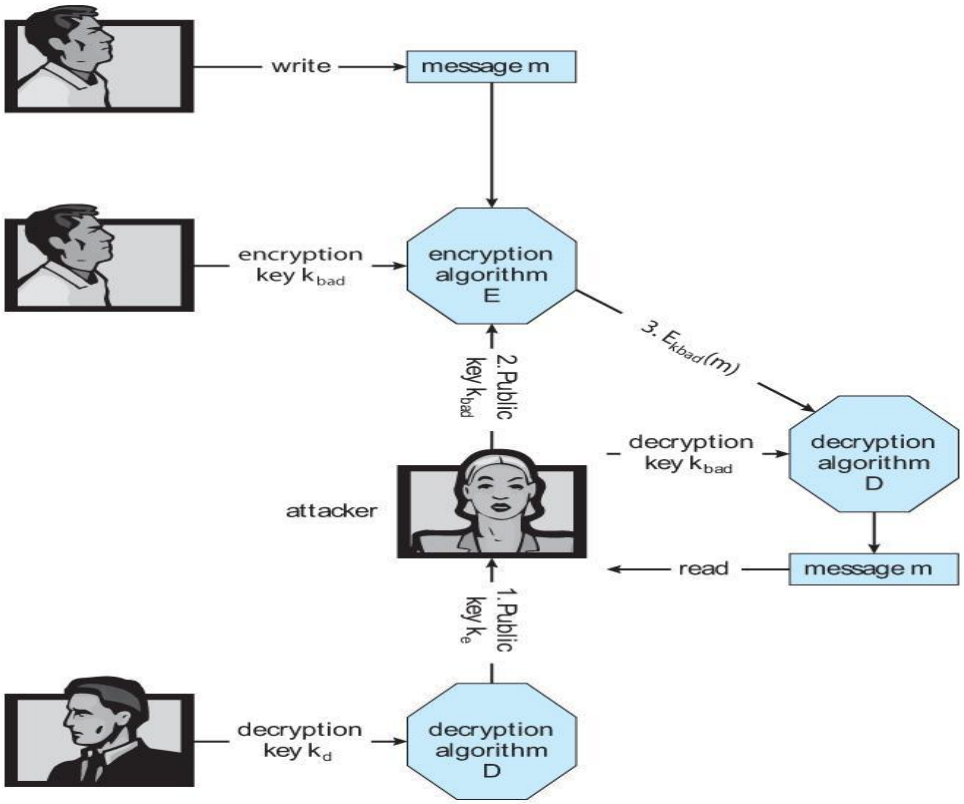


Figure 15.9 - A man-in-the-middle attack on asymmetric cryptography.

- One solution to the above problem involves ***digital certificates,*** which are public keys that have been digitally signed by a trusted third party. But wait a minute - How do we trust that third party, and how do we know ***they*** are really who they say they are? Certain ***certificate authorities*** have their public keys included within web browsers and other certificate consumers before they are distributed. These certificate authorities can then vouch for other trusted entities and so on in a web of trust, as explained more fully in section 15.4.3.

15.4.2 Implementation of Cryptography

- Network communications are implemented in multiple layers - Physical, Data Link, Network, Transport, and Application being the most common breakdown.
- Encryption and security can be implemented at any layer in the stack, with pros and cons to each choice:
  - Because packets at lower levels contain the contents of higher layers, encryption at lower layers automatically encrypts higher layer information at the same time.
  - However security and authorization may be important to higher levels independent of the underlying transport mechanism or route taken.
- At the network layer the most common standard is **IPSec,** a secure form of the IP layer, which is used to set up **Virtual Private Networks, VPNs.**
- At the transport layer the most common implementation is SSL, described below.

15.4.3 An Example: SSL

- SSL (Secure Sockets Layer) 3.0 was first developed by Netscape, and has now evolved into the industry-standard TLS protocol. It is used by web browsers to communicate securely with web servers, making it perhaps the most widely used security protocol on the Internet today.
- SSL is quite complex with many variations, only a simple case of which is shown here.
- The heart of SSL is ***session keys***, which are used once for symmetric encryption and then discarded, requiring the generation of new keys for each new session. The big challenge is how to safely create such keys while avoiding man-in-the-middle and replay attacks.

- Prior to commencing the transaction, the server obtains a ***certificate*** from a ***certification authority, CA,*** containing:
    - Server attributes such as unique and common names.
    - Identity of the public encryption algorithm, E( ), for the server.
    - The public key, k_e for the server.
    - The validity interval within which the certificate is valid.
    - A digital signature on the above issued by the CA:
        - a = S(K_CA)((attrs, E(k_e), interval)
- In addition, the client will have obtained a public ***verification algorithm,*** V(K_CA), for the certifying authority. Today's modern browsers include these built-in by the browser vendor for a number of trusted certificate authorities.
- The procedure for establishing secure communications is as follows:
    1. The client, c, connects to the server, s, and sends a random 28-byte number, n_c.
    2. The server replies with its own random value, n_s, along with its certificate of authority.
    3. The client uses its verification algorithm to confirm the identity of the sender, and if all checks out, then the client generates a 46 byte random **premaster secret, pms,** and sends an encrypted version of it as cpms = E(k_s)(pms)
    4. The server recovers pms as D(k_s)(cpms).
    5. Now both the client and the server can compute a shared 48-byte ***master secret, ms,*** = f(pms, n_s, n_c)
    6. Next, both client and server generate the following from ms:
        - Symmetric encryption keys k_sc_crypt and k_cs_crypt for encrypting messages from the server to the client and vice-versa respectively.
        - MAC generation keys k_sc_mac and k_cs_mac for generating authenticators on messages from server to client and client to server respectively.
    7. To send a message to the server, the client sends:
        - c = E(k_cs_crypt)(m, S(k_cs_mac))(m)))
    8. Upon receiving c, the server recovers:
        - (m,a) = D(k_cs_crypt)(c)
        - and accepts it if V(k_sc_mac)(m,a) is true.
- This approach enables both the server and client to verify the authenticity of every incoming message, and to ensure that outgoing messages are only readable by the process that originally participated in the key generation.
- SSL is the basis of many secure protocols,including ***Virtual Private Networks, VPNs,*** in which private data is distributed over the insecure public internet structure in an encrypted fashion that emulates a privately owned network.

15.5 User Authentication

- A lot of chapter 14, Protection, dealt with making sure that only certain users were allowed to perform certain tasks, i.e. that a users privileges were dependent on his or her identity. But how does one verify that identity to begin with?

15.5.1 Passwords

- Passwords are the most common form of user authentication. If the user is in possession of the correct password, then they are considered to have identified themselves.
- In theory separate passwords could be implemented for separate activities, such as reading this file, writing that file, etc. In practice most systems use one password to confirm user identity, and then authorization is based upon that identification. This is a result of the classic trade-off between security and convenience.

15.5.2 Password Vulnerabilities

- Passwords can be guessed.
    - Intelligent guessing requires knowing something about the intended target in specific, or about people and commonly used passwords in general.
    - Brute-force guessing involves trying every word in the dictionary, or every valid combination of characters. For this reason good passwords should not be in any dictionary (in any language), should be reasonably lengthy, and should use the full range of allowable characters by including upper and lower case characters, numbers, and special symbols.
- "Shoulder surfing" involves looking over people's shoulders while they are typing in their password.
    - Even if the lurker does not get the entire password, they may get enough clues to narrow it down, especially if they watch on repeated occasions.
    - Common courtesy dictates that you look away from the keyboard while someone is typing their password.
    - Passwords echoed as stars or dots still give clues, because an observer can determine how many characters are in the password. :-(
- "Packet sniffing" involves putting a monitor on a network connection and reading data contained in those packets.
    - SSH encrypts all packets, reducing the effectiveness of packet sniffing.
    - However you should still never e-mail a password, particularly not with the word "password" in the same message or worse yet the subject header.
    - Beware of any system that transmits passwords in clear text. ("Thank you for signing up for XYZ. Your new account and password information are shown below".) You probably want to have a spare throw-away password to give these entities, instead of using the same high-security password that you use for banking or other confidential uses.
- Long hard to remember passwords are often written down, particularly if they are used seldomly or must be changed frequently. Hence a security trade-off of passwords that are easily divined versus those that get written down. :-(
- Passwords can be given away to friends or co-workers, destroying the integrity of the entire user-identification system.
- Most systems have configurable parameters controlling password generation and what constitutes acceptable passwords.
    - They may be user chosen or machine generated.
    - They may have minimum and/or maximum length requirements.
    - They may need to be changed with a given frequency. (In extreme cases for every session.)
    - A variable length history can prevent repeating passwords.
    - More or less stringent checks can be made against password dictionaries.

15.5.3 Encrypted Passwords

- Modern systems do not store passwords in clear-text form, and hence there is no mechanism to look up an existing password.
- Rather they are encrypted and stored in that form. When a user enters their password, that too is encrypted, and if the encrypted version match, then user authentication passes.
- The encryption scheme was once considered safe enough that the encrypted versions were stored in the publicly readable file "/etc/passwd".
    - They always encrypted to a 13 character string, so an account could be disabled by putting a string of any other length into the password field.
    - Modern computers can try every possible password combination in a reasonably short time, so now the encrypted passwords are stored in files that are only readable by the super user. Any password-related programs run as setuid root to get access to these files. (/etc/shadow)

    - A random seed is included as part of the password generation process, and stored as part of the encrypted password. This ensures that if two accounts have the same plain-text password that they will not have the same encrypted password. However cutting and pasting encrypted passwords from one account to another will give them the same plain-text passwords.

15.5.4 One-Time Passwords

- One-time passwords resist shoulder surfing and other attacks where an observer is able to capture a password typed in by a user.
    - These are often based on a **challenge** and a **response.** Because the challenge is different each time, the old response will not be valid for future challenges.
        - For example, The user may be in possession of a secret function f(x). The system challenges with some given value for x, and the user responds with f(x), which the system can then verify. Since the challenger gives a different (random) x each time, the answer is constantly changing.
        - A variation uses a map (e.g. a road map) as the key. Today's question might be "On what corner is SEO located?", and tomorrow's question might be "How far is it from Navy Pier to Wrigley Field?" Obviously "Taylor and Morgan" would not be accepted as a valid answer for the second question!
    - Another option is to have some sort of electronic card with a series of constantly changing numbers, based on the current time. The user enters the current number on the card, which will only be valid for a few seconds. A ***two-factor authorization*** also requires a traditional password in addition to the number on the card, so others may not use it if it were ever lost or stolen.
    - A third variation is a ***code book,*** or ***one-time pad.*** In this scheme a long list of passwords is generated, and each one is crossed off and cancelled as it is used. Obviously it is important to keep the pad secure.

15.5.5 Biometrics

- Biometrics involve a physical characteristic of the user that is not easily forged or duplicated and not likely to be identical between multiple users.
    - Fingerprint scanners are getting faster, more accurate, and more economical.
    - Palm readers can check thermal properties, finger length, etc.
    - Retinal scanners examine the back of the users' eyes.
    - Voiceprint analyzers distinguish particular voices.
    - Difficulties may arise in the event of colds, injuries, or other physiological changes.

15.6 Implementing Security Defenses

15.6.1 Security Policy

- A security policy should be well thought-out, agreed upon, and contained in a living document that everyone adheres to and is updated as needed.
- Examples of contents include how often port scans are run, password requirements, virus detectors, etc.

15.6.2 Vulnerability Assessment

- Periodically examine the system to detect vulnerabilities.
    - Port scanning.
    - Check for bad passwords.

    - Look for suid programs.
    - Unauthorized programs in system directories.
    - Incorrect permission bits set.
    - Program checksums / digital signatures which have changed.
    - Unexpected or hidden network daemons.
    - New entries in startup scripts, shutdown scripts, cron tables, or other system scripts or configuration files.
    - New unauthorized accounts.
- The government considers a system to be only as secure as its most far-reaching component. Any system connected to the Internet is inherently less secure than one that is in a sealed room with no external communications.
- Some administrators advocate "security through obscurity", aiming to keep as much information about their systems hidden as possible, and not announcing any security concerns they come across. Others announce security concerns from the rooftops, under the theory that the hackers are going to find out anyway, and the only one kept in the dark by obscurity are honest administrators who need to get the word.

15.6.3 Intrusion Detection

- Intrusion detection attempts to detect attacks, both successful and unsuccessful attempts. Different techniques vary along several axes:
    - The time that detection occurs, either during the attack or after the fact.
    - The types of information examined to detect the attack(s). Some attacks can only be detected by analyzing multiple sources of information.
    - The response to the attack, which may range from alerting an administrator to automatically stopping the attack (e.g. killing an offending process), to tracing back the attack in order to identify the attacker.
        - Another approach is to divert the attacker to a ***honeypot***, on a ***honeynet.*** The idea behind a honeypot is a computer running normal services, but which no one uses to do any real work. Such a system should not see any network traffic under normal conditions, so any traffic going to or from such a system is by definition suspicious. Honeypots are normally kept on a honeynet protected by a ***reverse firewall,*** which will let potential attackers in to the honeypot, but will not allow any outgoing traffic. (So that if the honeypot is compromised, the attacker cannot use it as a base of operations for attacking other systems.) Honeypots are closely watched, and any suspicious activity carefully logged and investigated.
- Intrusion Detection Systems, IDSs, raise the alarm when they detect an intrusion. Intrusion Detection and Prevention Systems, IDPs, act as filtering routers, shutting down suspicious traffic when it is detected.
- There are two major approaches to detecting problems:
    - ***Signature-Based Detection*** scans network packets, system files, etc. looking for recognizable characteristics of known attacks, such as text strings for messages or the binary code for "exec /bin/sh". The problem with this is that it can only detect previously encountered problems for which the signature is known, requiring the frequent update of signature lists.
    - ***Anomaly Detection*** looks for "unusual" patterns of traffic or operation, such as unusually heavy load or an unusual number of logins late at night.
        - The benefit of this approach is that it can detect previously unknown attacks, so called ***zero-day attacks.***
        - One problem with this method is characterizing what is "normal" for a given system. One approach is to benchmark the system, but if the attacker is already present when the benchmarks are made, then the "unusual" activity is recorded as "the norm."

- Another problem is that not all changes in system performance are the result of security attacks. If the system is bogged down and really slow late on a Thursday night, does that mean that a hacker has gotten in and is using the system to send out SPAM, or does it simply mean that a CS 385 assignment is due on Friday? :-)
- To be effective, anomaly detectors must have a very low ***false alarm (false positive)*** rate, lest the warnings get ignored, as well as a low ***false negative*** rate in which attacks are missed.

15.6.4 Virus Protection

- Modern anti-virus programs are basically signature-based detection systems, which also have the ability (in some cases) of ***disinfecting*** the affected files and returning them back to their original condition.
- Both viruses and anti-virus programs are rapidly evolving. For example viruses now commonly mutate every time they propagate, and so anti-virus programs look for families of related signatures rather than specific ones.
- Some antivirus programs look for anomalies, such as an executable program being opened for writing (other than by a compiler.)
- Avoiding bootleg, free, and shared software can help reduce the chance of catching a virus, but even shrink-wrapped official software has on occasion been infected by disgruntled factory workers.
- Some virus detectors will run suspicious programs in a ***sandbox,*** an isolated and secure area of the system which mimics the real system.
- ***Rich Text Format, RTF,*** files cannot carry macros, and hence cannot carry Word macro viruses.
- Known safe programs (e.g. right after a fresh install or after a thorough examination) can be digitally signed, and periodically the files can be re-verified against the stored digital signatures. (Which should be kept secure, such as on off-line write-only medium.)

15.6.5 Auditing, Accounting, and Logging

- Auditing, accounting, and logging records can also be used to detect anomalous behavior.
- Some of the kinds of things that can be logged include authentication failures and successes, logins, running of suid or sgid programs, network accesses, system calls, etc. In extreme cases almost every keystroke and electron that moves can be logged for future analysis. (Note that on the flip side, all this detailed logging can also be used to analyze system performance. The down side is that the logging also *affects* system performance (negatively!), and so a Heisenberg effect applies.)
- "The Cuckoo's Egg" tells the story of how Cliff Stoll detected one of the early UNIX break ins when he noticed anomalies in the accounting records on a computer system being used by physics researchers.

Tripwire Filesystem (New Sidebar)

- The tripwire filesystem monitors files and directories for changes, on the assumption that most intrusions eventually result in some sort of undesired or unexpected file changes.
- The tw.config file indicates what directories are to be monitored, as well as what properties of each file are to be recorded. (E.g. one may choose to monitor permission and content changes, but not worry about read access times.)
- When first run, the selected properties for all monitored files are recorded in a database. Hash codes are used to monitor file contents for changes.
- Subsequent runs report any changes to the recorded data, including hash code changes, and any newly created or missing files in the monitored directories.

- For full security it is necessary to also protect the tripwire system itself, most importantly the database of recorded file properties. This could be saved on some external or write-only location, but that makes it harder to change the database when legitimate changes are made.
- It is difficult to monitor files that are *supposed to* change, such as log files. The best tripwire can do in this case is to watch for anomalies, such as a log file that shrinks in size.
- Free and commercial versions are available at http://tripwire.org and http://tripwire.com.

15.7 Firewalling to Protect Systems and Networks

- Firewalls are devices (or sometimes software) that sit on the border between two security domains and monitor/log activity between them, sometimes restricting the traffic that can pass between them based on certain criteria.
- For example a firewall router may allow HTTP: requests to pass through to a web server inside a company domain while not allowing telnet, ssh, or other traffic to pass through.
- A common architecture is to establish a de-militarized zone, DMZ, which sort of sits "between" the company domain and the outside world, as shown below. Company computers can reach either the DMZ or the outside world, but outside computers can only reach the DMZ. Perhaps most importantly, the DMZ cannot reach any of the other company computers, so even if the DMZ is breached, the attacker cannot get to the rest of the company network. (In some cases the DMZ may have limited access to company computers, such as a web server on the DMZ that needs to query a database on one of the other company computers.)

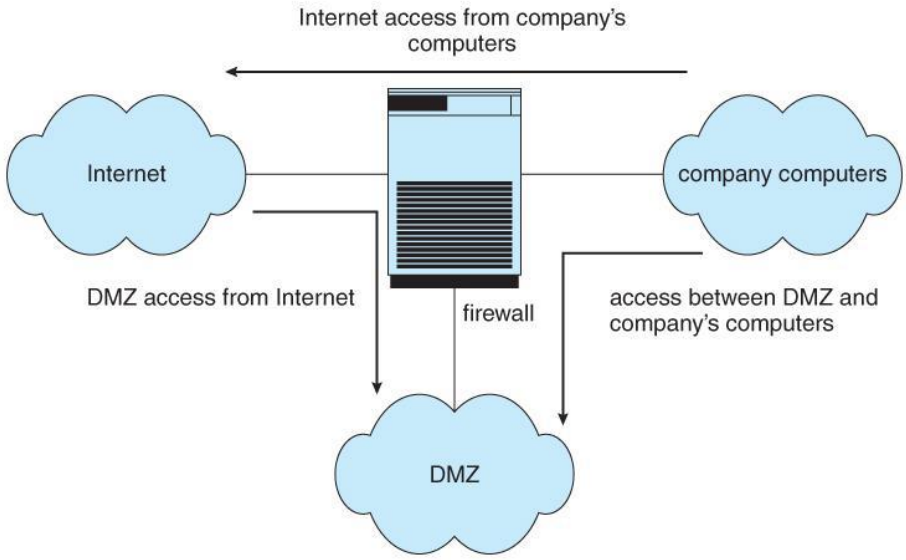


Figure 15.10 - Domain separation via firewall.

- Firewalls themselves need to be resistant to attacks, and unfortunately have several vulnerabilities:
  - ***Tunneling,*** which involves encapsulating forbidden traffic inside of packets that are allowed.
  - Denial of service attacks addressed at the firewall itself.
  - Spoofing, in which an unauthorized host sends packets to the firewall with the return address of an authorized host.
- In addition to the common firewalls protecting a company internal network from the outside world, there are also some specialized forms of firewalls that have been recently developed:
  - A ***personal firewall*** is a software layer that protects an individual computer. It may be a part of the operating system or a separate software package.
  - An ***application proxy firewall*** understands the protocols of a particular service and acts as a stand-in (and relay) for the particular service. For example, and SMTP proxy firewall would accept SMTP requests from the outside world, examine them for security concerns, and forward only the "safe" ones on to the real SMTP server behind the firewall.
  - ***XML firewalls*** examine XML packets only, and reject ill-formed packets. Similar firewalls exist for other specific protocols.

- ***System call firewalls*** guard the boundary between user mode and system mode, and reject any system calls that violate security policies.

15.8 Computer-Security Classifications (Optional)

- No computer system can be 100% secure, and attempts to make it so can quickly make it unusable.
- However one can establish a level of trust to which one feels "safe" using a given computer system for particular security needs.
- The U.S. Department of Defense's "Trusted Computer System Evaluation Criteria" defines four broad levels of trust, and sub-levels in some cases:
  - Level D is the least trustworthy, and encompasses all systems that do not meet any of the more stringent criteria. DOS and Windows 3.1 fall into level D, which has no user identification or authorization, and anyone who sits down has full access and control over the machine.
  - Level C1 includes user identification and authorization, and some means of controlling what users are allowed to access what files. It is designed for use by a group of mostly cooperating users, and describes most common UNIX systems.
  - Level C2 adds individual-level control and monitoring. For example file access control can be allowed or denied on a per-individual basis, and the system administrator can monitor and log the activities of specific individuals. Another restriction is that when one user uses a system resource and then returns it back to the system, another user who uses the same resource later cannot read any of the information that the first user stored there. (I.e. buffers, etc. are wiped out between users, and are not left full of old contents.) Some special secure versions of UNIX have been certified for C2 security levels, such as SCO.
  - Level B adds sensitivity labels on each object in the system, such as "secret", "top secret", and "confidential". Individual users have different clearance levels, which controls which objects they are able to access. All human-readable documents are labeled at both the top and bottom with the sensitivity level of the file.
  - Level B2 extends sensitivity labels to all system resources, including devices. B2 also supports covert channels and the auditing of events that could exploit covert channels.
  - B3 allows creation of access-control lists that denote users NOT given access to specific objects.
  - Class A is the highest level of security. Architecturally it is the same as B3, but it is developed using formal methods which can be used to ***prove*** that the system meets all requirements and cannot have any possible bugs or other vulnerabilities. Systems in class A and higher may be developed by trusted personnel in secure facilities.
  - These classifications determine what a system *can* implement, but it is up to security policy to determine *how* they are implemented in practice. These systems and policies can be reviewed and certified by trusted organizations, such as the National Computer Security Center. Other standards may dictate physical protections and other issues.

I/O Systems

I/O Hardware

- I/O devices can be roughly categorized as storage, communications, user-interface, and other
- Devices communicate with the computer via signals sent over wires or through the air.
- Devices connect with the computer via ***ports***, e.g. a serial or parallel port.
- A common set of wires connecting multiple devices is termed a ***bus*.**
    - Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
    - Figure 13.1 below illustrates three of the four bus types commonly found in a modern PC:
        1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
        2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
        3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
        4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

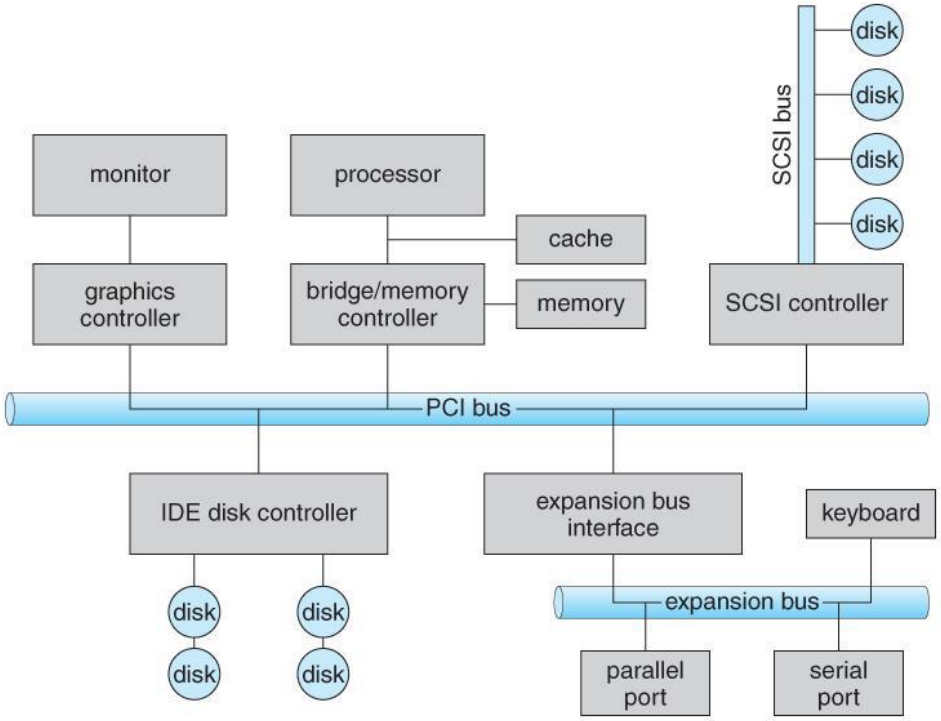


Figure 13.1 - A typical PC bus structure.

- One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:
    1. The ***data-in register*** is read by the host to get input from the device.
    2. The ***data-out register*** is written by the host to send output.
    3. The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
    4. The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.
- Figure 13.2 shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 13.2 - Device I/O port locations on PCs (partial).

- Another technique for communicating with devices is ***memory-mapped I/O.***
  - In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
  - Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
  - Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
  - A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
  - (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA. See section 13.2.3 below.)

13.2.1 Polling

- One simple means of device ***handshaking*** involves polling:
  1. The host repeatedly checks the ***busy bit*** on the device until it becomes clear.
  2. The host writes a byte of data into the data-out register, and sets the ***write bit*** in the command register (in either order.)
  3. The host sets the ***command ready bit*** in the command register to notify the device of the pending command.
  4. When the device controller sees the command-ready bit set, it first sets the busy bit.
  5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
  6. The device controller then clears the ***error bit*** in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.
- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

13.2.2 Interrupts

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an ***interrupt-request line*** that is sensed after every instruction.

    - A device's controller ***raises*** an interrupt by asserting a signal on the interrupt request line.
    - The CPU then performs a state save, and transfers control to the ***interrupt handler*** routine at a fixed address in memory. (The CPU ***catches*** the interrupt and ***dispatches*** the interrupt handler.)
    - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a ***return from interrupt*** instruction to return control to the CPU. (The interrupt handler ***clears*** the interrupt by servicing the device.)
        - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)
- Figure 13.3 illustrates the interrupt-driven I/O procedure:

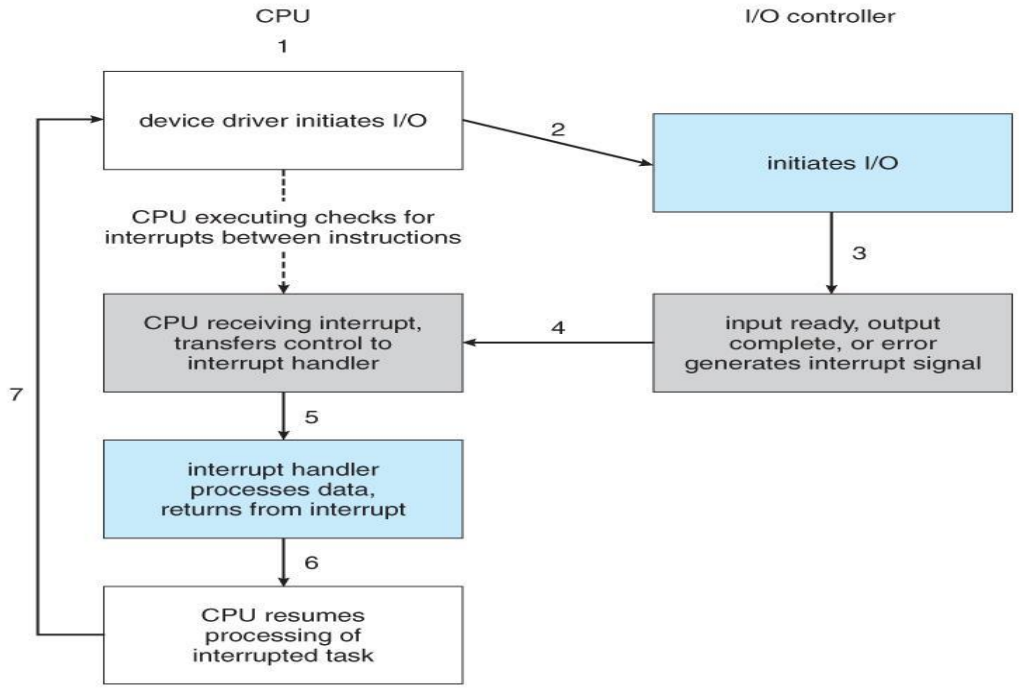


Figure 13.3 - Interrupt-driven I/O cycle.

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
    1. The need to defer interrupt handling during critical processing,
    2. The need to determine ***which*** interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
    3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.
- These issues are handled in modern computer architectures with ***interrupt-controller*** hardware.
    - Most CPUs now have two interrupt-request lines: One that is ***non-maskable*** for critical error conditions and one that is ***maskable,*** that the CPU can temporarily ignore during critical processing.
    - The interrupt mechanism accepts an ***address,*** which is usually one of a small set of numbers for an offset into a table called the ***interrupt vector.*** This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
    - The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be ***interrupt chained***. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.

- Figure 13.4 shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.
- Modern interrupt hardware also supports ***interrupt priority levels***, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 13.4 - Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
  - The scheduler sets a hardware timer before transferring control over to a user process.
  - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
  - The scheduler does a state-restore of a ***different*** process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.
- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves **two** interrupts:
  - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
  - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

- The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

13.2.3 Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.***
- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.
- A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,***and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation.)
- Figure 13.5 below illustrates the DMA process.

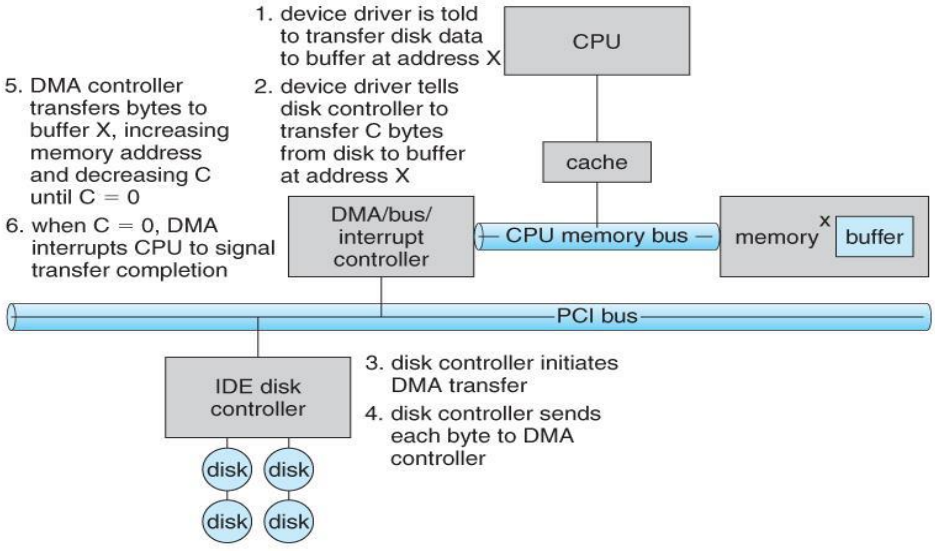


Figure 13.5 - Steps in a DMA transfer.

13.2.4 I/O Hardware Summary

13.3 Application I/O Interface

- User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into ***device drivers***, while application layers are presented with a common interface for all (or at least large general categories of) devices.

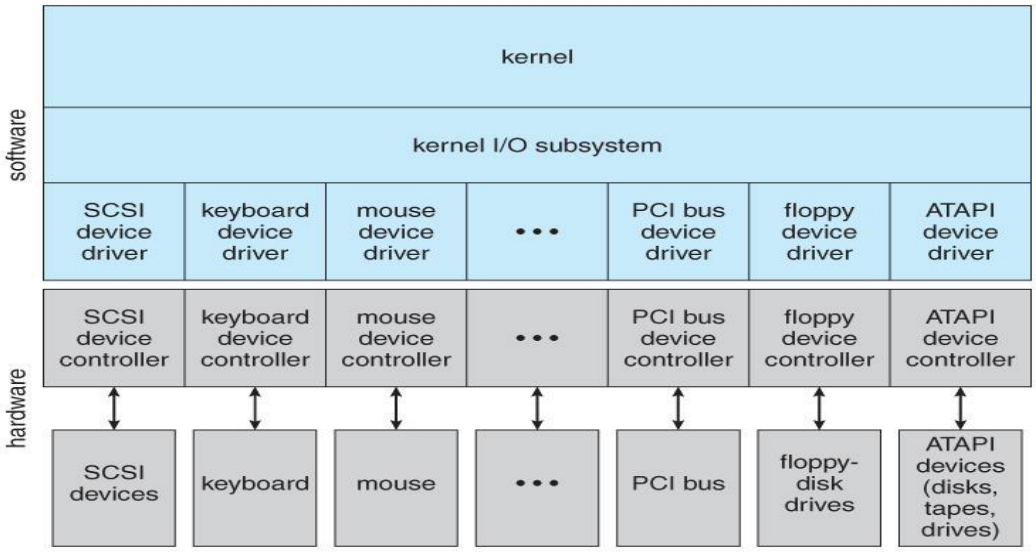


Figure 13.6 - A kernel I/O structure.

- Devices differ on many different dimensions, as outlined in Figure 13.7:

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 13.7 - Characteristics of I/O devices.

- Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.
- Most OSes also have an ***escape,*** or ***back door,*** which allows applications to send commands directly to device drivers if needed. In UNIX this is the ***ioctl( )*** system call (I/O Control). Ioctl( ) takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.

13.3.1 Block and Character Devices

- ***Block devices*** are accessed a block at a time, and are indicated by a "b" as the first character in a long listing on UNIX systems. Operations supported include read( ), write( ), and seek( ).
    - Accessing blocks on a hard drive directly (without going through the filesystem structure) is called ***raw I/O***, and can speed up certain operations by bypassing the

buffering and locking normally conducted by the OS. (It then becomes the application's responsibility to manage those issues.)
    - A new alternative is ***direct I/O,*** which uses the normal filesystem access, but which disables buffering and locking operations.
- Memory-mapped file I/O can be layered on top of block-device drivers.
    - Rather than reading in the entire file, it is mapped to a range of memory addresses, and then paged into memory as needed using the virtual memory system.
    - Access to the file is then accomplished through normal memory accesses, rather than through read( ) and write( ) system calls. This approach is commonly used for executable program code.
- ***Character devices*** are accessed one byte at a time, and are indicated by a "c" in UNIX long listings. Supported operations include get( ) and put( ), with more advanced functionality such as reading an entire line supported by higher-level library routines.

13.3.2 Network Devices

- Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
- One common and popular interface is the ***socket*** interface, which acts like a cable or pipeline connecting two networked entities. Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
- The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

13.3.3 Clocks and Timers

- Three types of time services are commonly needed in modern systems:
    - Get the current time of day.
    - Get the elapsed time (system or wall clock) since a previous event.
    - Set a timer to trigger event X at time T.
- Unfortunately time operations are not standard across all systems.
- A ***programmable interrupt timer, PIT*** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.
    - The scheduler uses a PIT to trigger interrupts for ending time slices.
    - The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
    - Networks use PIT to abort or repeat operations that are taking too long to complete. I.e. resending packets if an acknowledgement is not received before the timer goes off.
    - More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.
- On most systems the system clock is implemented by counting interrupts generated by the PIT. Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time. An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

13.3.4 Blocking and Non-blocking I/O

- With ***blocking I/O*** a process is moved to the wait queue when an I/O request is made, and moved back to the ready queue when the request completes, allowing other processes to run in the meantime.

- With ***non-blocking I/O*** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not. This allows the process to check for available data without getting hung completely if it is not there.
- One approach for programmers to implement non-blocking I/O is to have a multi-threaded application, in which one thread makes blocking I/O calls (say to read a keyboard or mouse), while other threads continue to update the screen or perform other tasks.
- A subtle variation of the non-blocking I/O is the ***asynchronous I/O,*** in which the I/O request returns immediately allowing the process to continue on with other tasks, and then the process is notified (via changing a process variable, or a software interrupt, or a callback function) when the I/O operation has completed and the data is available for use. (The regular non-blocking I/O returns immediately with whatever results are available, but does not complete the operation and notify the process later.)

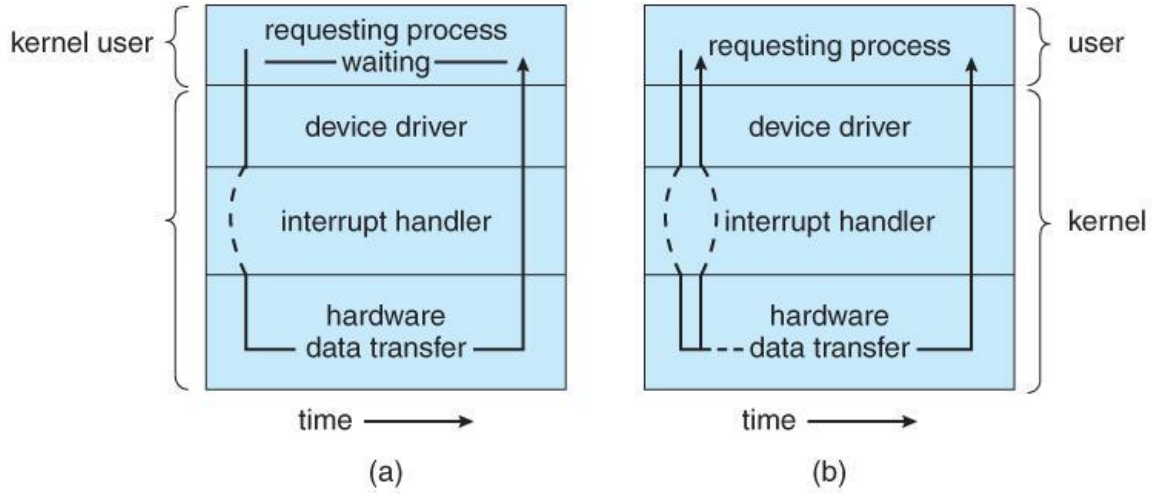


Figure 13.8 - Two I/O methods: (a) synchronous and (b) asynchronous.

13.4 Kernel I/O Subsystem

13.4.1 I/O Scheduling

- Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part in request scheduling.
- The classic example is the scheduling of disk accesses, as discussed in detail in chapter 12.
- Buffering and caching can also help, and can allow for more flexible scheduling options.
- On systems with many devices, separate request queues are often kept for each device:

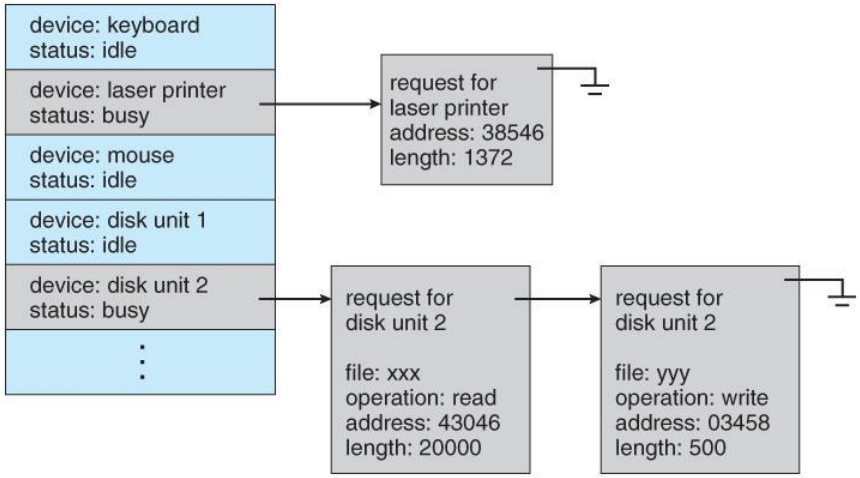


Figure 13.9 - Device-status table.

13.4.2 Buffering

- Buffering of I/O is performed for (at least) 3 major reasons:
    1. Speed differences between two devices. (See Figure 13.10 below.) A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once. So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full. This is known as***double buffering.*** (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen. This prevents the user from ever seeing any half-finished screen images.)
    2. Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.
    3. To support ***copy semantics.*** For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer. Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.

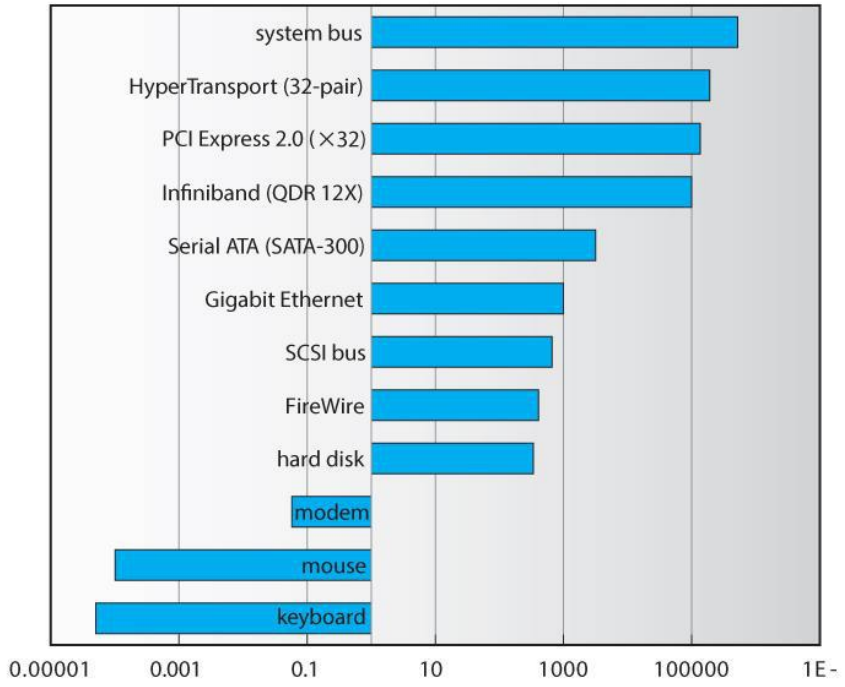


Figure 13.10 - Sun Enterprise 6000 device-transfer rates (logarithmic).

13.4.3 Caching

- Caching involves keeping a ***copy*** of data in a faster-access location than where the data is normally stored.
- Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.
- Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes.)

13.4.4 Spooling and Device Reservation

- A ***spool (Simultaneous Peripheral Operations On-Line)*** buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

- If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.
- Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.
- Spool queues can be general (any laser printer) or specific (printer number 42.)
- OSes can also provide support for processes to request / get exclusive access to a particular device, and/or to wait until a device becomes available.

13.4.5 Error Handling

- I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash).
- I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable ***errno*** to one of a hundred or so well-defined values to indicate the specific error that has occurred. (See errno.h for a complete listing, or man errno.)
- Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

13.4.6 I/O Protection

- The I/O system must protect against either accidental or deliberate erroneous I/O.
- User applications are not allowed to perform I/O in user mode - All I/O requests are handled through system calls that must be performed in kernel mode.
- Memory mapped areas and I/O ports must be protected by the memory management system, **but** access to these areas cannot be totally denied to user programs. (Video games and some other applications need to be able to write directly to video memory for optimal performance for example.) Instead the memory protection system restricts access so that only one process at a time can access particular parts of memory, such as the portion of the screen memory corresponding to a particular window.

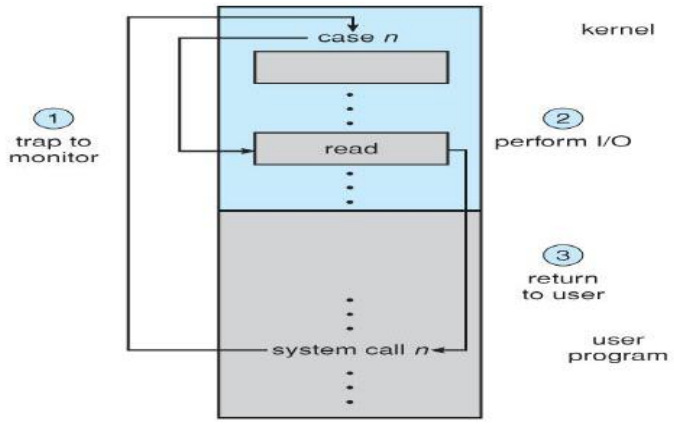


Figure 13.11 - Use of a system call to perform I/O.

13.4.7 Kernel Data Structures

- The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table.
- These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure 13.12 below.)
- Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.

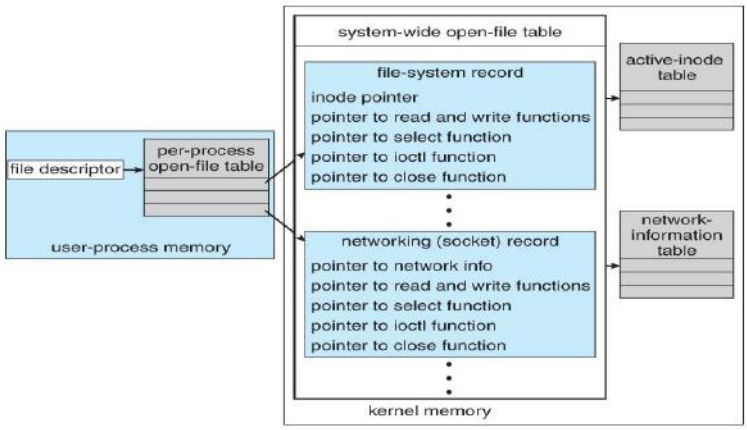


Figure 13.12 - UNIX I/O kernel structure.

13.5 Transforming I/O Requests to Hardware Operations

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
- UNIX uses a ***mount table*** to map filename prefixes (e.g. /usr) to specific mounted devices. Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file.)
- UNIX uses special ***device files,*** usually located in /dev, to represent and access physical devices directly.
    - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
    - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
    - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- Figure 13.13 illustrates the steps taken to process a (blocking) read request:

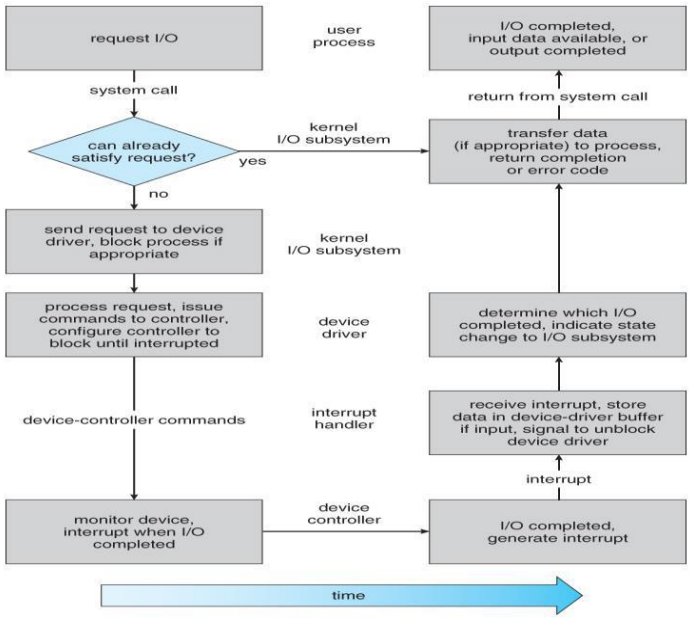


Figure 13.13 - The life cycle of an I/O request.