19MCAC204: OPERATING SYSTEM

Unit I – Introduction of Operating System

Introduction

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

An operating system is a software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Operating System – Definition:

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

A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being application programs.

An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

Functions of Operating system – Operating system performs three functions:

Convenience: An OS makes a computer more convenient to use.

Efficiency: An OS allows the computer system resources to be used in an efficient manner.

Ability to Evolve: An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system functions at the same time without interfering with service.

Operating system as User Interface –

User

System and application programs

Operating system

Hardware

Every general-purpose computer consists of the hardware, operating system, system programs, and application programs. The hardware consists of memory, CPU, ALU, and I/O devices, peripheral device, and storage device. System program consists of compilers, loaders, editors, OS, etc. The application program consists of business programs, database programs.

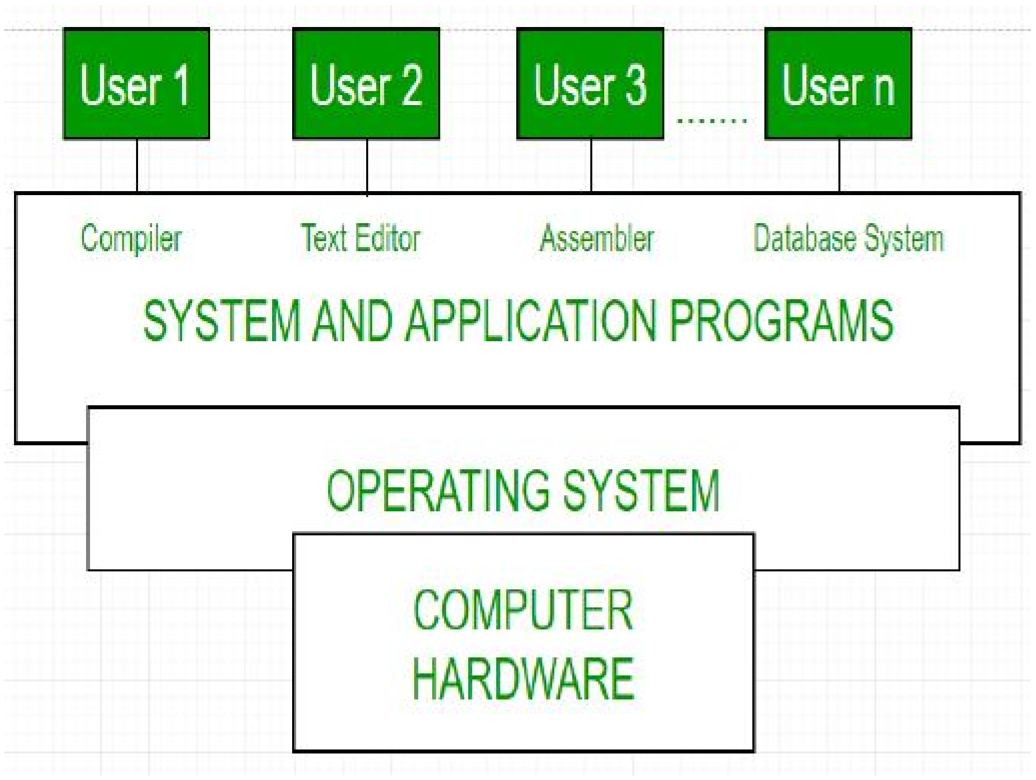


Fig1: Conceptual view of a computer system

Every computer must have an operating system to run other programs. The operating system coordinates the use of the hardware among the various system programs and application programs for various users. It simply provides an environment within which other programs can do useful work.

The operating system is a set of special programs that run on a computer system that allows it to work properly. It performs basic tasks such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling peripheral devices.

OS is designed to serve two basic purposes:

It controls the allocation and use of the computing System's resources among the various user and tasks.

It provides an interface between the computer hardware and the programmer that simplifies and makes feasible for coding, creation, debugging of application programs.

The Operating system must support the following tasks. The task are:

Provides the facilities to create, modification of programs and data files using an editor.

Access to the compiler for translating the user program from high level language to machine language.

Provide a loader program to move the compiled program code to the computer's memory for execution.

Provide routines that handle the details of I/O programming.

I/O System Management –

The module that keeps track of the status of devices is called the I/O traffic controller. Each I/O device has a device handler that resides in a separate process associated with that device.

The I/O subsystem consists of

A memory Management component that includes buffering caching and spooling.

A general device driver interface.

Drivers for specific hardware devices.

Assembler –
The input to an assembler is an assembly language program. The output is an object program plus information that enables the loader to prepare the object program for execution. At one time, the computer programmer had at his disposal a basic machine that interpreted, through hardware, certain fundamental instructions. He would program this computer by writing a series of ones and Zeros (Machine language), place them into the memory of the machine.

Compiler –
The High-level languages- examples are FORTRAN, COBOL, ALGOL and PL/I are processed by compilers and interpreters. A compiler is a program that accepts a source program in a "high-level language " and produces a corresponding object program. An interpreter is a program that appears to execute a source program as if it was machine language. The same name (FORTRAN, COBOL, etc.) is often used to designate both a compiler and its associated language.

Loader –
A Loader is a routine that loads an object program and prepares it for execution. There are various loading schemes: absolute, relocating and direct-linking. In general, the loader must load, relocate and link the object program. The loader is a program that places programs into memory and prepares them for execution. In a simple loading scheme, the assembler outputs the machine language translation of a program on a secondary device and a loader places it in the core. The

loader places into memory the machine language version of the user's program and transfers control to it. Since the loader program is much smaller than the assembler, those make more core available to the user's program.

History of Operating system –
Operating system has been evolving through the years. Following Table shows the history of OS.

GENERATION	YEAR	ELECTRONIC DEVICE USED	TYPES OF OS DEVICE
First	1945-55	Vaccum Tubes	Plug Boards
Second	1955-65	Transistors	Batch Systems
Third	1965-80	Integrated Circuits(IC)	Multiprogramming
Fourth	Since 1980	Large Scale Integration	PC

Types of Operating System –

Batch Operating System- Sequence of jobs in a program on a computer without manual interventions.

Time sharing operating System- allows many users to share the computer resources.(Max utilization of the resources).

Distributed operating System- Manages a group of different computers and make appear to be a single computer.

Network operating system- computers running in different operating system can participate in common network (It is used for security purpose).

Real time operating system – meant applications to fix the deadlines.

Examples of Operating System are –

Windows (GUI based, PC)

GNU/Linux (Personal, Workstations, ISP, File and print server, Three-tier client/Server)

macOS (Macintosh), used for Apple's personal computers and work stations (MacBook, iMac).

Android (Google's Operating System for smartphones/tablets/smartwatches)

iOS (Apple's OS for iPhone, iPad and iPod Touch)

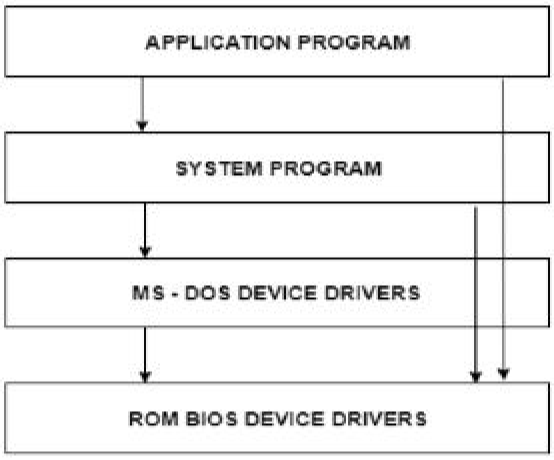
Operating System Structure

An operating system is a construct that allows the user application programs to interact with the system hardware. Since the operating system is such a complex structure, it should be created with utmost care so it can be used and modified easily. An easy way to do this is to create the operating system in parts. Each of these parts should be well defined with clear inputs, outputs and functions.

Simple Structure

There are many operating systems that have a rather simple structure. These started as small systems and rapidly expanded much further than their scope. A common example of this is MS-DOS. It was designed simply for a niche amount for people. There was no indication that it would become so popular.

An image to illustrate the structure of MS-DOS is as follows:



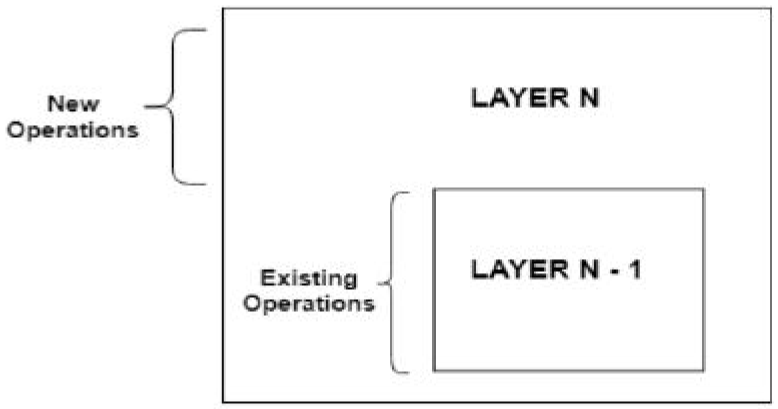
MS-DOS STRUCTURE

It is better that operating systems have a modular structure, unlike MS-DOS. That would lead to greater control over the computer system and its various applications. The modular structure would also allow the programmers to hide information as required and implement internal routines as they see fit without changing the outer specifications.

Layered Structure

One way to achieve modularity in the operating system is the layered approach. In this, the bottom layer is the hardware and the topmost layer is the user interface.

An image demonstrating the layered approach is as follows:



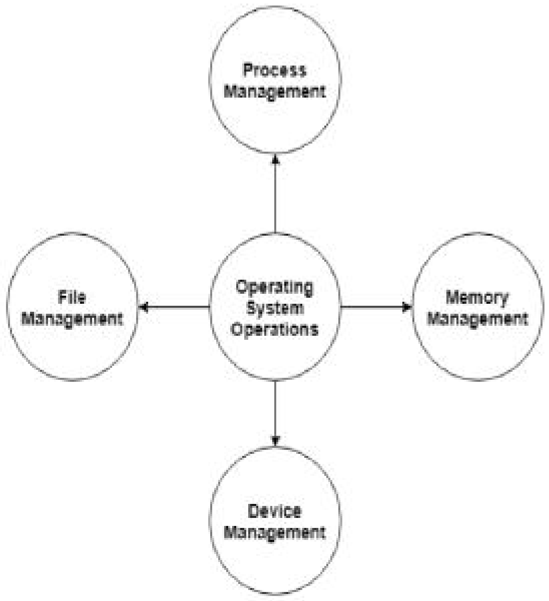
Layered Structure of Operating System

As seen from the image, each upper layer is built on the bottom layer. All the layers hide some structures, operations etc from their upper layers.

Operating System Operations

An operating system is a construct that allows the user application programs to interact with the system hardware. Operating system by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work.

The major operations of the operating system are process management, memory management, device management and file management. These are given in detail as follows:



Process Management

The operating system is responsible for managing the processes i.e assigning the processor to a process at a time. This is known as process scheduling. The different algorithms used for process scheduling are FCFS (first come first served), SJF (shortest job first), priority scheduling, round robin scheduling etc.

There are many scheduling queues that are used to handle processes in process management. When the processes enter the system, they are put into the job queue. The processes that are ready to execute in the main memory are kept in the ready queue. The processes that are waiting for the I/O device are kept in the device queue.

Memory Management

Memory management plays an important part in operating system. It deals with memory and the moving of processes from disk to primary memory for execution and back again.

The activities performed by the operating system for memory management are:

- The operating system assigns memory to the processes as required. This can be done using best fit, first fit and worst fit algorithms.
- All the memory is tracked by the operating system i.e. it nodes what memory parts are in use by the processes and which are empty.
- The operating system deallocated memory from processes as required. This may happen when a process has been terminated or if it no longer needs the memory.

Device Management

There are many I/O devices handled by the operating system such as mouse, keyboard, disk drive etc. There are different device drivers that can be connected to the operating system to handle a specific device. The device controller is an interface between the device and the device driver. The user applications can access all the I/O devices using the device drivers, which are device specific codes.

File Management

Files are used to provide a uniform view of data storage by the operating system. All the files are mapped onto physical devices that are usually non volatile so data is safe in the case of system failure.

The files can be accessed by the system in two ways i.e. sequential access and direct access:

- **Sequential Access**

  The information in a file is processed in order using sequential access. The files records are accessed on after another. Most of the file systems such as editors, compilers etc. use sequential access.

- **Direct Access**

  In direct access or relative access, the files can be accessed in random for read and write operations. The direct access model is based on the disk model of a file, since it allows random accesses.

<u>Protection and Security in Operating System</u>

Protection and security requires that computer resources such as CPU, softwares, memory etc. are protected. This extends to the operating system as well as the data in the system. This can be done by ensuring integrity, confidentiality and availability in the operating system. The system must be protect against unauthorized access, viruses, worms etc.

Threats to Protection and Security

A threat is a program that is malicious in nature and leads to harmful effects for the system. Some of the common threats that occur in a system are:

Virus

Viruses are generally small snippets of code embedded in a system. They are very dangerous and can corrupt files, destroy data, crash systems etc. They can also spread further by replicating themselves as required.

Trojan Horse

A trojan horse can secretly access the login details of a system. Then a malicious user can use these to enter the system as a harmless being and wreak havoc.

Trap Door

A trap door is a security breach that may be present in a system without the knowledge of the users. It can be exploited to harm the data or files in a system by malicious people.

Worm

A worm can destroy a system by using its resources to extreme levels. It can generate multiple copies which claim all the resources and don't allow any other processes to access them. A worm can shut down a whole network in this way.

Denial of Service

These type of attacks do not allow the legitimate users to access a system. It overwhelms the system with requests so it is overwhelmed and cannot work properly for other user.

Protection and Security Methods

The different methods that may provide protect and security for different computer systems are:

Authentication

This deals with identifying each user in the system and making sure they are who they claim to be. The operating system makes sure that all the users are authenticated before they access the system. The different ways to make sure that the users are authentic are:

- **Username/ Password**

  Each user has a distinct username and password combination and they need to enter it correctly before they can access the system.

- **User Key/ User Card**

  The users need to punch a card into the card slot or use they individual key on a keypad to access the system.

- **User Attribute Identification**

  Different user attribute identifications that can be used are fingerprint, eye retina etc. These are unique for each user and are compared with the existing samples in the database. The user can only access the system if there is a match.

One Time Password

These passwords provide a lot of security for authentication purposes. A one time password can be generated exclusively for a login every time a user wants to enter the system. It cannot be used more than once. The various ways a one time password can be implemented are:

- **Random Numbers**

  The system can ask for numbers that correspond to alphabets that are pre arranged. This combination can be changed each time a login is required.

- **Secret Key**

  A hardware device can create a secret key related to the user id for login. This key can change each time.

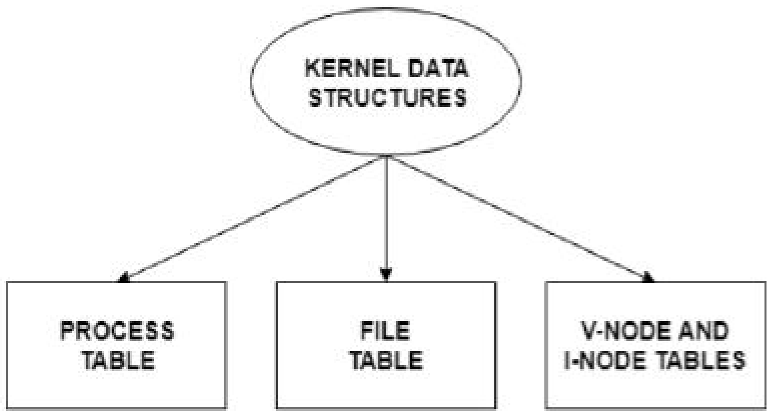
Kernel Data Structures

The kernel data structures are very important as they store data about the current state of the system. For example, if a new process is created in the system, a kernel data structure is created that contains the details about the process.

Most of the kernel data structures are only accessible by the kernel and its subsystems. They may contain data as well as pointers to other data structures.

Kernel Components

The kernel stores and organizes a lot of information. So it has data about which processes are running in the system, their memory requirements, files in use etc. To handle all this, three important structures are used. These are process table, file table and v node/ i node information.



Details about these are as follows:

Process Table

The process table stores information about all the processes running in the system. These include the storage information, execution status, file information etc.

When a process forks a child, its entry in the process table is duplicated including the file information and file pointers. So the parent and the child process share a file.

File Table

The file table contains entries about all the files in the system. If two or more processes use the same file, then they contain the same file information and the file descriptor number.

Each file table entry contains information about the file such as file status (file read or file write), file offset etc. The file offset specifies the position for next read or write into the file.

The file table also contains v-node and i-node pointers which point to the virtual node and index node respectively. These nodes contain information on how to read a file.

V-Node and I-Node Tables

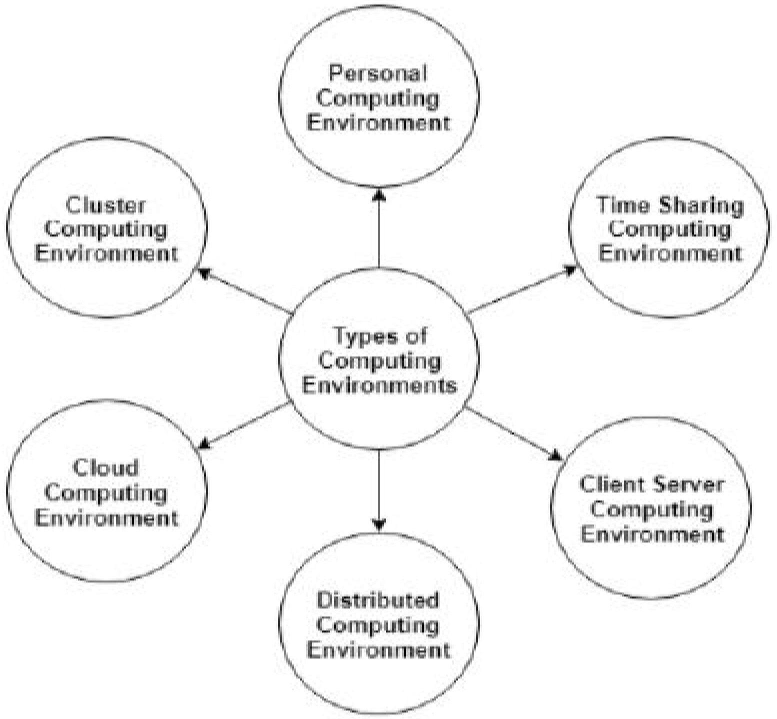
Both the v-node and i-node are references to the storage system of the file and the storage mechanisms. They connect the hardware to the software.

The v-node is an abstract concept that defines the method to access file data without worrying about the actual structure of the system. The i-node specifies file access information like file storage device, read/write procedures etc.

Computing Environments

A computer system uses many devices, arranged in different ways to solve many problems. This constitutes a computing environment where many computers are used to process and exchange information to handle multiple issues.

The different types of Computing Environments are:


Personal Computing Environment
Cluster Computing Environment
Time Sharing Computing Environment
Types of Computing Environments
Cloud Computing Environment
Client Server Computing Environment
Distributed Computing Environment

Let us begin with Personal Computing Environment:

Personal Computing Environment
In the personal computing environment, there is a single computer system. All the system processes are available on the computer and executed there. The different devices that constitute a personal computing environment are laptops, mobiles, printers, computer systems, scanners etc.

Time Sharing Computing Environment

The time sharing computing environment allows multiple users to share the system simultaneously. Each user is provided a time slice and the processor switches rapidly among the users according to it. Because of this, each user believes that they are the only ones using the system.

Client Server Computing Environment

In client server computing, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server. Both the client and server usually communicate via a computer network but sometimes they may reside in the same system.

Distributed Computing Environment

A distributed computing environment contains multiple nodes that are physically separate but linked together using the network. All the nodes in this system communicate with each other and handle processes in tandem. Each of these nodes contains a small part of the distributed operating system software.

Cloud Computing Environment

The computing is moved away from individual computer systems to a cloud of computers in cloud computing environment. The cloud users only see the service being provided and not the internal details of how the service is provided. This is done by pooling all the computer resources and then managing them using a software.

Cluster Computing Environment

The clustered computing environment is similar to parallel computing environment as they both have multiple CPUs. However a major difference is that clustered systems are created by two or more individual computer systems merged together which then work parallel to each other.

Open Source Operating Systems

Open Source operating systems are released under a license where the copyright holder allows others to study, change as well as distribute the software to other people. This can be done for any reason. The different open source operating system available in the market are:

Cosmos

This is an open source operating system written mostly in programming language C#. Its full form is C# Open Source Managed Operating System. Till 2016, Cosmos did not intend to be a fully fledged operating system but a system that allowed other developers to easily build their own operating systems. It also hid the inner workings of the hardware from the developers thus providing data abstraction.

FreeDOS

This was a free operating system developed for systems compatible with IBM PC computers. FreeDOS provides a complete environment to run legacy software and other embedded systems. It can booted from a floppy disk or USB flash drive as required. FreeDos is licensed under the GNU General Public license and contains free and open source software. So there is no license fees required for its distribution and changes to the system are permitted.

Genode

Genode is free as well as open source. It contains a microkernel layer and different user components. It is one of the few open source operating systems not derived from a licenced operating system such as Unix. Genode can be used as an operating system for computers, tablets etc. as required. It is also used as a base for virtualisation, interprocess communication, software development etc. as it has a small code system.

Ghost OS

This is a free, open source operating system developed for personal computers. It started as a research project and developed to contain various advanced features like graphical user interface, C library etc. The Ghost operating system features multiprocessing and multitasking and is based on the Ghost Kernel. Most of the programming in Ghost OS is done in C++.

ITS

The incompatible time-sharing system was developed by the MIT Artificial Intelligence Library. It is principally a time sharing system. There is a remote login facility which allowed guest users to informally try out the operating system and its features using ARPAnet. ITS also gave out many new features that were unique at that time such as device independent graphics terminal, virtual devices, inter machine file system access etc.

OSv

This was an operating system released in 2013. It was mainly focused on cloud computing and was built to run on top of a virtual machine as a guest. This is the reason it doesn't include drivers for bare hardware. In the OSv operating system, everything runs in the kernel address space and there is no concept of a multi-user system.

Phantom OS

This is an operating system that is based on the concepts on persistent virtual memory and is code oriented. It was mostly developed by Russian developers. Phantom OS is not based on concepts of famous operating systems such as Unix. Its main goal is simplicity and effectiveness in process management.

Operating System Services

An Operating System provides services to both the users and to the programs.

- It provides programs an environment to execute.
- It provides users the services to execute the programs in a convenient manner.

Following are a few common services provided by an operating system −

- Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

Program execution

Operating systems handle many kinds of activities from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated as a process.

A process includes the complete execution context (code to execute, data to manipulate, registers, OS resources in use). Following are the major activities of an operating system with respect to program management −

- Loads a program into memory.
- Executes the program.
- Handles program's execution.
- Provides a mechanism for process synchronization.
- Provides a mechanism for process communication.
- Provides a mechanism for deadlock handling.

I/O Operation

An I/O subsystem comprises of I/O devices and their corresponding driver software. Drivers hide the peculiarities of specific hardware devices from the users.

An Operating System manages the communication between user and device drivers.

- I/O operation means read or write operation with any file or any specific I/O device.
- Operating system provides the access to the required I/O device when required.

File system manipulation

A file represents a collection of related information. Computers can store files on the disk (secondary storage), for long-term storage purpose. Examples of storage media include magnetic tape, magnetic disk and optical disk drives like CD, DVD. Each of these media has its own properties like speed, capacity, data transfer rate and data access methods.

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions. Following are the major activities of an operating system with respect to file management –

- Program needs to read a file or write a file.
- The operating system gives the permission to the program for operation on file.
- Permission varies from read-only, read-write, denied and so on.
- Operating System provides an interface to the user to create/delete files.
- Operating System provides an interface to the user to create/delete directories.
- Operating System provides an interface to create the backup of file system.

Communication

In case of distributed systems which are a collection of processors that do not share memory, peripheral devices, or a clock, the operating system manages communications between all the processes. Multiple processes communicate with one another through communication lines in the network.

The OS handles routing and connection strategies, and the problems of contention and security. Following are the major activities of an operating system with respect to communication –

- Two processes often require data to be transferred between them
- Both the processes can be on one computer or on different computers, but are connected through a computer network.

- Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

Error handling

Errors can occur anytime and anywhere. An error may occur in CPU, in I/O devices or in the memory hardware. Following are the major activities of an operating system with respect to error handling −

- The OS constantly checks for possible errors.
- The OS takes an appropriate action to ensure correct and consistent computing.

Resource Management

In case of multi-user or multi-tasking environment, resources such as main memory, CPU cycles and files storage are to be allocated to each user or job. Following are the major activities of an operating system with respect to resource management −

- The OS manages all kinds of resources using schedulers.
- CPU scheduling algorithms are used for better utilization of CPU.

Protection

Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

Protection refers to a mechanism or a way to control the access of programs, processes, or users to the resources defined by a computer system. Following are the major activities of an operating system with respect to protection −

- The OS ensures that all access to system resources is controlled.
- The OS ensures that external I/O devices are protected from invalid access attempts.
- The OS provides authentication features for each user by means of passwords.

Operating System and User Interface

As already mentioned, in addition to the hardware, a computer also needs a set of programs—an operating system—to control the devices. This page will discuss the following:

There are different kinds of operating systems:

such as Windows, Linux and Mac OS

There are also different versions of these operating systems,

e.g. Windows 7, 8 and 10

Operating systems can be used with different user interfaces (UI):

text user interfaces (TUI) and graphical user interfaces (GUI) as examples

Graphical user interfaces have many similarities in different operating systems:

such as the start menu, desktop etc.

When you can recognize the typical parts of each operating system's user interface, you will mostly be able to use both Windows and Linux as well as e.g. Mac OS.

THE ROLE OF OPERATING SYSTEM IN THE COMPUTER

An operating system (OS) is a set of programs which ensures the interoperability of the hardware and software in your computer. The operating system enables, among other things,

The identification and activation of devices connected to the computer,

The installation and use of programs, and

The handling of files.

What happens when you turn on your computer or smartphone?
– The computer checks the functionality of its components and any devices connected to it, and starts to look for the OS on a hard drive or other memory media.
– If the OS is found, the computer starts to load it into the RAM (Random Access Memory).
– When the OS has loaded, the computer waits for commands from you.

DIFFERENT OPERATING SYSTEMS

Over the years, several different operating systems have been developed for different purposes. The most typical operating systems in ordinary computers are Windows, Linux and Mac OS.

WINDOWS

The name of the Windows OS comes from the fact that programs are run in "windows": each program has its own window, and you can have several programs open at the same time. Windows is the most popular OS for home computers, and there are several versions of it. The newest version is Windows 10.

LINUX AND UNIX

Linux is an open-source OS, which means that its program code is freely available to software developers. This is why thousands of programmers around the world have developed Linux, and it is considered the most tested OS in the world. Linux has been very much influenced by the commercial Unix OS.

In addition to servers, Linux is widely used in home computers, since there are a great number of free programs for it (for text and image processing, spreadsheets, publishing, etc.). Over the years, many different versions of Linux have become available for distribution, most of which are free for the user (such as Ubuntu, Fedora and Mint, to name a few). See the additional reading material for more information on Linux.

MAC OS X

Apple's Mac computers have their own operating system, OS X. Most of the programs that are available for PCs are also available for Macs running under OS X, but these two types of computers cannot use the exact same programs: for example, you cannot install the Mac version of the Microsoft Office suite on a Windows computer. You can install other operating systems on Mac computers, but the OS X is only available for computers made by Apple. Apple's lighter portable devices (iPads, iPhones) use a light version of the same operating system, called iOS.

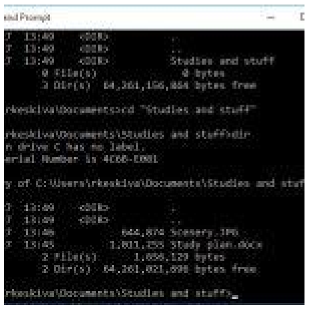
Mac computers are popular because OS X is considered fast, easy to learn and very stable and Apple's devices are considered well-designed—though rather expensive. See the additional reading material for more information on OS X.

ANDROID

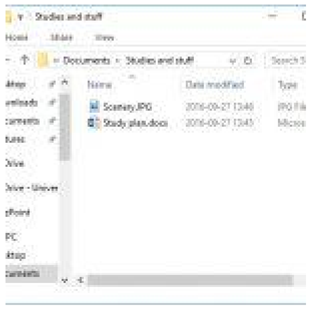
Android is an operating system designed for phones and other mobile devices. Android is not available for desktop computers, but in mobile devices it is extremely popular: more than a half of all mobile devices in the world run on Android.

USER INTERFACES

A user interface (UI) refers to the part of an operating system, program, or device that allows a user to enter and receive information. A **text-based user interface** (see the image to the left) displays text, and its commands are usually typed on a command line using a keyboard. With a **graphical user interface** (see the right-hand image), the functions are carried out by clicking or moving buttons, icons and menus by means of a pointing device.



```
nd Prompt
7  13:49    <DIR>          .
7  13:49    <DIR>          ..
7  13:49    <DIR>          Studies and stuff
       0 File(s)              0 bytes
       3 Dir(s)  64,261,156,864 bytes free

rkeskiva\Documents>cd "Studies and stuff"

rkeskiva\Documents\Studies and stuff>dir
n drive C has no label.
erial Number is 4C68-E081

y of C:\Users\rkeskiva\Documents\Studies and stuf

7  13:49    <DIR>          .
7  13:49    <DIR>          ..
7  13:46           644,874 Scenery.JPG
7  13:45         1,011,255 Study plan.docx
       2 File(s)      1,656,129 bytes
       2 Dir(s)  64,261,021,696 bytes free

rkeskiva\Documents\Studies and stuff>_
```



Larger image: text UI | graphical UI

The images contain the same information: a directory listing of a computer. You can often carry out the same tasks regardless of which kind of UI you are using.

TEXT USER INTERFACE (TUI)

Modern graphical user interfaces have evolved from text-based UIs. Some operating systems can still be used with a text-based user interface. In this case, the commands are entered as text (e.g., “cat story.txt”).

This demonstration will show you how to rename a file in a TUI: the example will show both the *ren* (rename) and the *dir* (directory listing) commands.The use of a TUI does not differ very much from a GUI (Graphical User Interface) controlled with e.g. a mouse (many TUIs mirror GUIs).

To display the text-based Command Prompt in Windows, open the **Start** menu and type **cmd**. Press **Enter** on the keyboard to launch the command prompt in a separate window. With the command prompt, you can type your commands from the keyboard instead of using the mouse.

GRAPHICAL USER INTERFACE

In most operating systems, the primary user interface is graphical, i.e. instead of typing the commands you manipulate various graphical objects (such as icons) with a pointing device. The underlying principle of different graphical user interfaces (GUIs) is largely the same, so by knowing how to use a Windows UI, you will most likely know how to use Linux or some other GUI.

Most GUIs have the following basic components:

- a start menu with program groups
- a taskbar showing running programs
- a desktop
- various icons and shortcuts.

System Calls

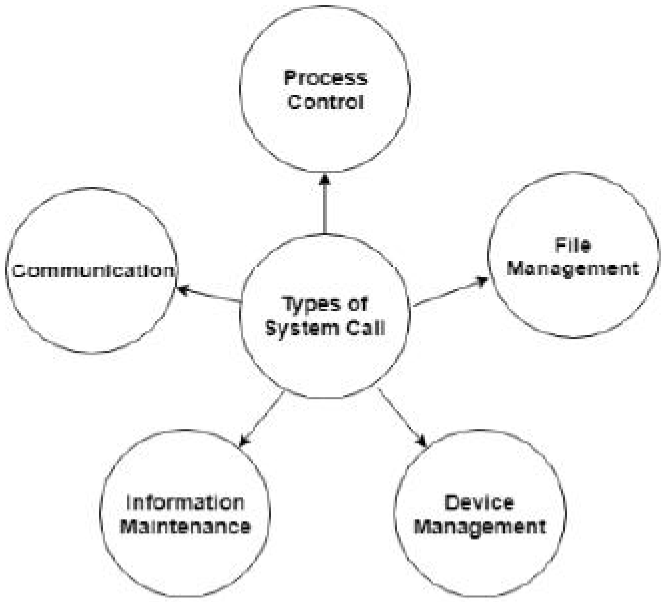
In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the **operating system** it is executed on. ... It provides an interface between a process and **operating system** to allow user-level processes to request services of the **operating system**.

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers.

System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows:



Process Control

These system calls deal with processes such as process creation, process termination etc.

File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

Some of the examples of all the above types of system calls in Windows and Unix are given as follows:

Types of System Calls	Windows	Linux

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Management	CreateFile()
	ReadFile()
	WriteFile()
	CloseHandle()
		open()
		read()
		write()
		close()

Device Management	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

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

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

There are many different system calls as shown above. Details of some of those system calls are as follows:

wait()

In some systems, a process may wait for another process to complete its execution. This happens when a parent process creates a child process and the execution of the parent process is suspended until the child process executes. The suspending of the parent process occurs with a wait() system call. When the child process completes execution, the control is returned back to the parent process.

exec()

This system call runs an executable file in the context of an already running process. It replaces the previous executable file. This is known as an overlay. The original process identifier remains since a new process is not created but data, heap, stack etc. of the process are replaced by the new process.

fork()

Processes use the fork() system call to create processes that are a copy of themselves. This is one of the major methods of process creation in operating systems. When a parent process creates a child process and the execution of the parent process is suspended until the child process executes. When the child process completes execution, the control is returned back to the parent process.

exit()

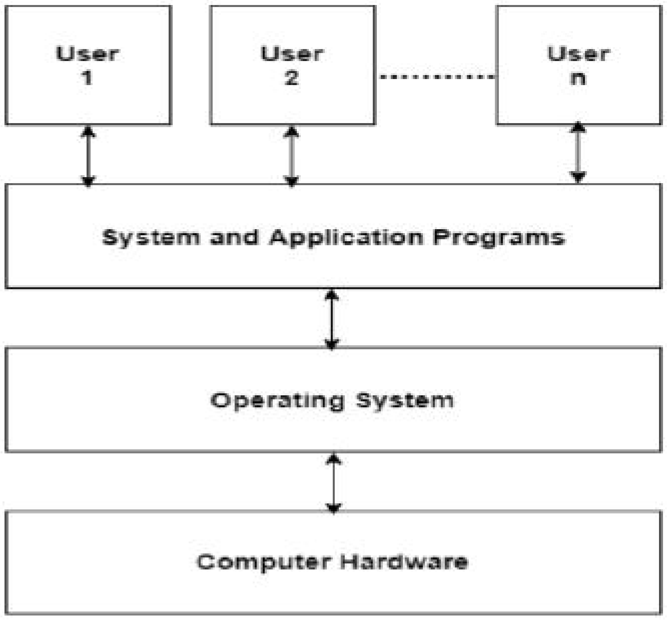
The exit() system call is used by a program to terminate its execution. In a multithreaded environment, this means that the thread execution is complete. The operating system reclaims resources that were used by the process after the exit() system call.

kill()

The kill() system call is used by the operating system to send a termination signal to a process that urges the process to exit. However, kill system call does not necessary mean killing the process and can have various meanings.

System Programs

There are mainly two categories of programs i.e. application programs and system programs. A diagram that demonstrates their place in the logical computer hierarchy is as follows:



Application Programs

These programs perform a particular function directly for the users. Some of the common application programs include Email, web browsers, gaming software, word processors, graphics software, media player etc.

All of these programs provide an application to the end users, so they are known as application programs. For example: a web browser is used to find information while a gaming software is used to play games.

The requests for service and application communication systems used in an application by a programmer is known as an application program interface (API).

System Programs

The system programs are used to program the operating system software. While application programs provide software that is used directly by the user, system programs provide software that are used by other systems such as SaaS applications, computational science applications etc.

The attributes of system programming are:

- Using system programming, a programmer can make assumptions about the hardware of the system that the program runs on.
- A low level programming language is used in system programming normally. This is so that the programs can operate in low resource environments easily.
- Most system programs are created to have a low runtime overhead. These programs may have small runtime library.

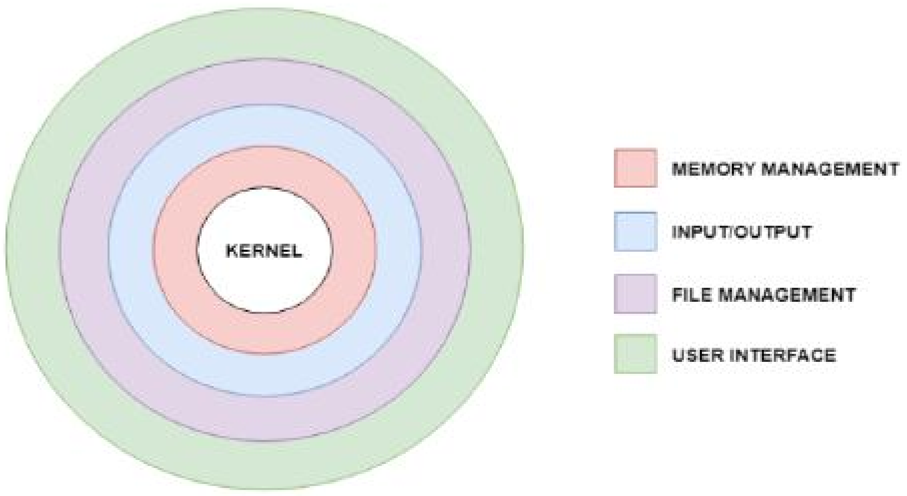
- Some parts of the system programs may be directly written in assembly language by the programmers.
- A debugger cannot be used on system programs mostly. This problem can be solved by running the programs in a simulated environment.

Some examples of system programs are operating system, networking system, web site server, data backup server etc.

Operating System design and Implementation

An operating system is a construct that allows the user application programs to interact with the system hardware. Operating system by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work.

There are many problems that can occur while designing and implementing an operating system. These are covered in operating system design and implementation.



Layered Operating System Design

Operating System Design Goals

It is quite complicated to define all the goals and specifications of the operating system while designing it.The design changes depending on the type of the operating system i.e if it is batch system, time shared system, single user system, multi user system, distributed system etc.

There are basically two types of goals while designing an operating system. These are:

User Goals

The operating system should be convenient, easy to use, reliable, safe and fast according to the users. However, these specifications are not very useful as there is no set method to achieve these goals.

System Goals

The operating system should be easy to design, implement and maintain. These are specifications required by those who create, maintain and operate the operating system. But there is not specific method to achieve these goals as well.

Operating System Mechanisms and Policies

There is no specific way to design an operating system as it is a highly creative task. However, there are general software principles that are applicable to all operating systems.

A subtle difference between mechanism and policy is that mechanism shows how to do something and policy shows what to do. Policies may change over time and this would lead to changes in mechanism. So, it is better to have a general mechanism that would require few changes even when a policy change occurs.

For example - If the mechanism and policy are independent, then few changes are required in mechanism if policy changes. If a policy favours I/O intensive processes over CPU intensive processes, then a policy change to preference of CPU intensive processes will not change the mechanism.

Operating System Implementation

The operating system needs to be implemented after it is designed. Earlier they were written in assembly language but now higher level languages are used. The first system not written in assembly language was the Master Control Program (MCP) for Burroughs Computers.

Advantages of Higher Level Language

There are multiple advantages to implementing an operating system using a higher level language such as: the code is written more fast, it is compact and also easier to debug and understand. Also, the operating system can be easily moved from one hardware to another if it is written in a high level language.

Disadvantages of Higher Level Language

Using high level language for implementing an operating system leads to a loss in speed and increase in storage requirements. However in modern systems only a small amount of code is needed for high performance, such as the CPU scheduler and memory manager. Also, the bottleneck routines in the system can be replaced by assembly language equivalents if required.

Operating System-debugging

Debugging is the process of finding the problems in a computer system and solving them. There are many different ways in which operating systems perform debugging. Some of these are:

Log Files

The log files record all the events that occur in an operating system. This is done by writing all the messages into a log file. There are different types of log files. Some of these are given as follows:

Event Logs

These stores the records of all the events that occur in the execution of a system. This is done so that the activities of all the events can be understood to diagnose problems.

Transaction Logs

The transaction logs store the changes to the data so that the system can recover from crashes and other errors. These logs are readable by a human.

Message Logs

These logs store both the public and private messages between the users. They are mostly plain text files, but in some cases they may be HTML files.

Core Dump Files

The core dump files contain the memory address space of a process that terminates unexpectedly. The creation of the core dump is triggered in response to program crashes by the kernel. The core dump files are used by the developers to find the program's state at the time of its termination so that they can find out why the termination occurred.

The automatic creation of the core dump files can be disabled by the users. This may be done to improve performance, clear disk space or increase security.

Crash Dump Files

In the event of a total system failure, the information about the state of the operating system is captured in crash dump files. There are three types of dump that can be captured when a system crashes. These are:

Complete Memory Dump

The whole contents of the physical memory at the time of the system crash are captured in the complete memory dump. This is the default setting on the Windows Server System.

Kernel Memory Dump

Only the kernel mode read and write pages that are present in the main memory at the time of the system crash are stored in the kernel memory dump.

Small Memory Dump

This memory dump contains the list of device drivers, stop code, process and thread information, kernel stack etc.

Trace Listings

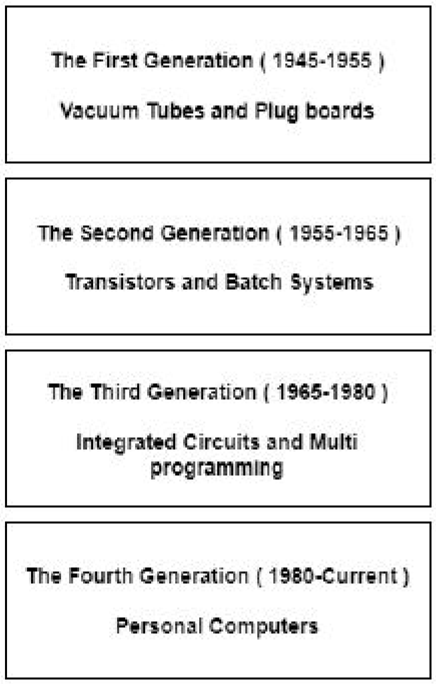
The trace listing record information about a program execution using logging. This information is used by programmers for debugging. System administrators and technical personnel can use the trace listings to find the common problems with software using software monitoring tools.

Profiling

This is a type of program analysis that measures various parameters in a program such as space and time complexity, frequency and duration of function calls, usage of specific instructions etc. Profiling is done by monitoring the source code of the required system program using a code profiler.

Operating System-Generation

Operating Systems have evolved over the years. So, their evolution through the years can be mapped using generations of operating systems. There are four generations of operating systems. These can be described as follows:


The First Generation (1945-1955)
Vacuum Tubes and Plug boards
The Second Generation (1955-1965)
Transistors and Batch Systems
The Third Generation (1965-1980)
Integrated Circuits and Multi programming
The Fourth Generation (1980-Current)
Personal Computers

OPERATING SYSTEM GENERATIONS

The First Generation (1945 - 1955): Vacuum Tubes and Plugboards

Digital computers were not constructed until the second world war. Calculating engines with mechanical relays were built at that time. However, the mechanical relays were very slow and were later replaced with vacuum tubes. These machines were enormous but were still very slow.

These early computers were designed, built and maintained by a single group of people. Programming languages were unknown and there were no operating systems so all the programming was done in machine language. All the problems were simple numerical calculations.

By the 1950's punch cards were introduced and this improved the computer system. Instead of using plugboards, programs were written on cards and read into the system.

The Second Generation (1955 - 1965): Transistors and Batch Systems

Transistors led to the development of the computer systems that could be manufactured and sold to paying customers. These machines were known as mainframes and were locked in air-conditioned computer rooms with staff to operate them.

The Batch System was introduced to reduce the wasted time in the computer. A tray full of jobs was collected in the input room and read into the magnetic tape. After that, the tape was rewound and mounted on a tape drive. Then the batch operating system was loaded in which read the first job from the tape and ran it. The output was written on the second tape. After the whole batch was done, the input and output tapes were removed and the output tape was printed.

The Third Generation (1965 - 1980): Integrated Circuits and Multiprogramming

Until the 1960's, there were two types of computer systems i.e the scientific and the commercial computers. These were combined by IBM in the System/360. This used integrated circuits and provided a major price and performance advantage over the second generation systems.

The third generation operating systems also introduced multiprogramming. This meant that the processor was not idle while a job was completing its I/O operation. Another job was scheduled on the processor so that its time would not be wasted.

The Fourth Generation (1980 - Present): Personal Computers

Personal Computers were easy to create with the development of large-scale integrated circuits. These were chips containing thousands of transistors on a square centimeter of silicon. Because of these, microcomputers were much cheaper than minicomputers and that made it possible for a single individual to own one of them.

The advent of personal computers also led to the growth of networks. This created network operating systems and distributed operating systems. The users were aware of a network while using a network operating system and could log in to remote machines and copy files from one machine to another.

System Boot

The booting of an operating system works by loading a very small program into the computer and then giving that program control so that it in turn loads the entire operating system. Booting or loading an operating system is different than installing it, which is generally an initial one-time activity.

Booting and Dual Booting of Operating System

The procedure of starting a computer by loading the kernel is known as **Booting** the system. Hence it needs a special program, stored in ROM to do this job known as the Bootstrap loader. Example: BIOS (boot input output system). A modern PC BIOS (Basic Input/Output System) supports booting from various devices.Typically, the BIOS will allow the user to configure a boot order. If the boot order is set to:

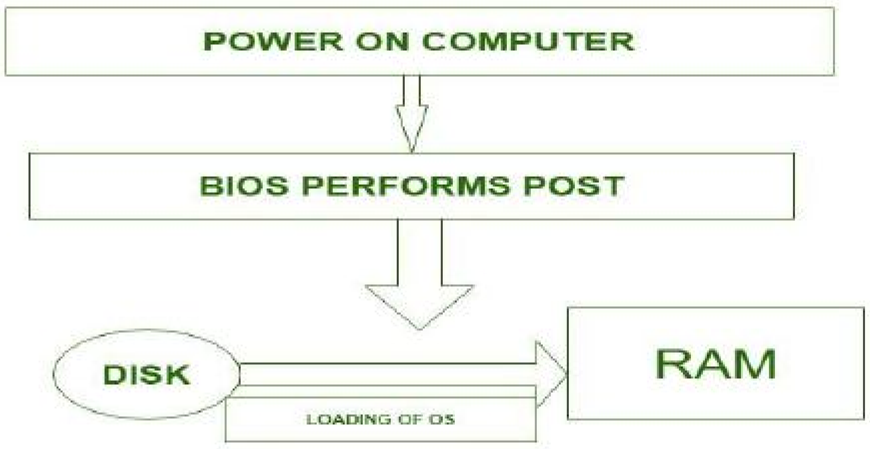
CD Drive

Hard Disk Drive

Network

Then the BIOS will try to boot from the CD drive first, and if that fails then it will try to boot from the hard disk drive, and if that fails then it will try to boot from the network, and if that fails then it won't boot at all.

Booting is a startup sequence that starts the operating system of a computer when it is turned on. A boot sequence is the initial set of operations that the computer performs when it is switched on. Every computer has a boot sequence. Bootstrap loader locates the kernel, loads it into main memory and starts its execution.In some systems, a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel.



Dual Booting:

When two operating system are installed on the computer system then it is called dual booting. In fact multiple operating systems can be installed on such a system. But how system knows which operating system is to boot? A boot loader that understand multiple file systems and multiple operating system can occupy the boot space.Once loaded, it can boot one of the operating systems available on the disk.The disk can have multiple partitions, each containing a different type of operating system. When a computer system turn on, a boot manager program displays a menu, allowing user to choose the operating system to use.

Unit II – Process Management

- A process can be thought of as a program in execution. A process will need certain resources - such as CPU time, memory, files, and I/O devices - to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.
- A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.
- Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.
- The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Process Concepts:

A question that arises in discussing operating systems involves what to call all the CPU activities. A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**.

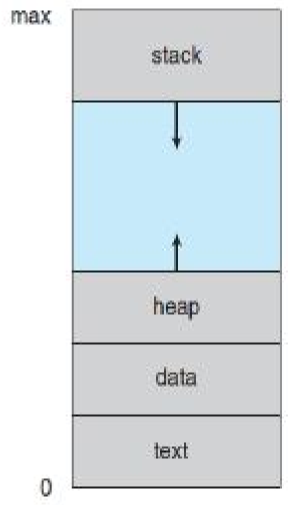
Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package.

And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management.

In many respects, all these activities are similar, so we call all of them **processes**.

The terms ***job*** and ***process*** are used almost interchangeably in this text.

The Process:



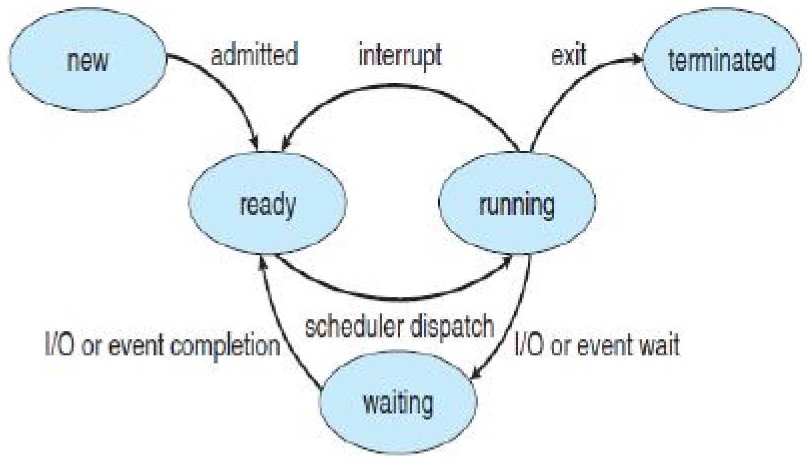
- A process is more than the program code, which is sometimes known as the **text section**.
- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- A process generally also includes the process **stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and
- A **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in Figure. We emphasize that a program by itself is not a process.
  - A program is a ***passive*** entity, such as a file containing a list of instructions stored on disk (often called an **executable file**).
  - In contrast, a process is an ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

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).

Process State

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



New- The process is being created.

Running- Instructions are being executed.

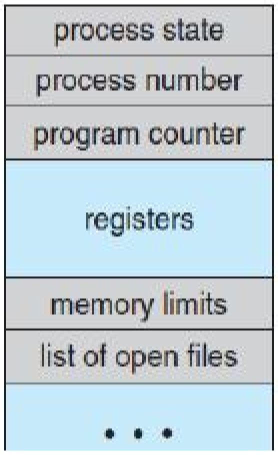
Waiting.-The process is waiting for some event to occur (such as an I/O completion)

Ready.-The process is waiting to be assigned to a processor.

Terminated- The process has finished execution.

It is important to realize that only one process can be *running* on any processor at any instant. Many processes may be *ready* and *waiting*.

Process Control Block-[PCB]:



Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.

A **PCB** is shown in Figure.

It contains many pieces of information associated with a specific process, including these:

- **Process state**. The state may be new, ready, running, and waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information**. This information includes the amount

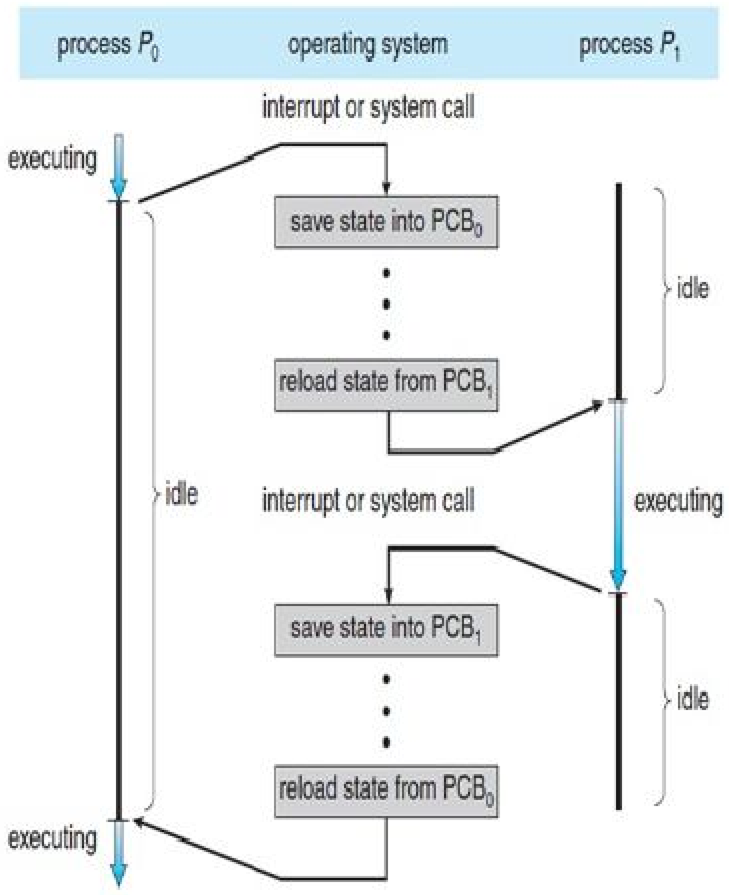


Diagram showing CPU switch from process to process.

of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

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

Process Scheduling

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

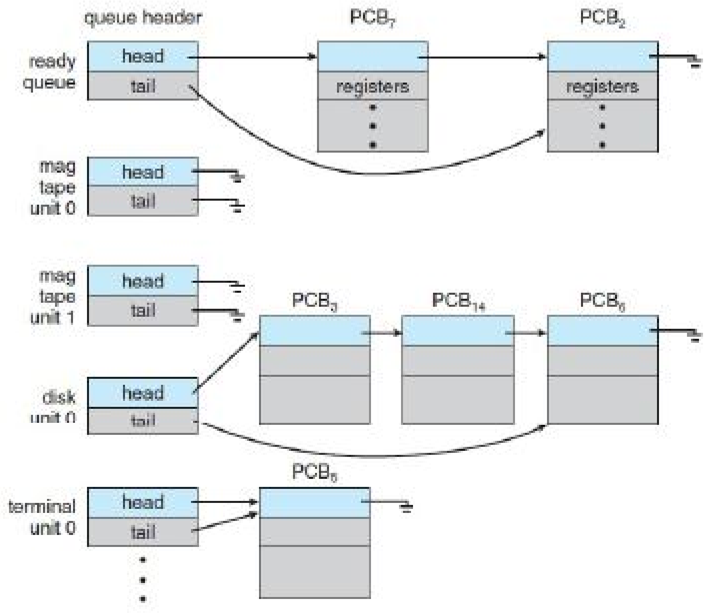
To meet these objectives, the **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU.

For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues:

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

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**.

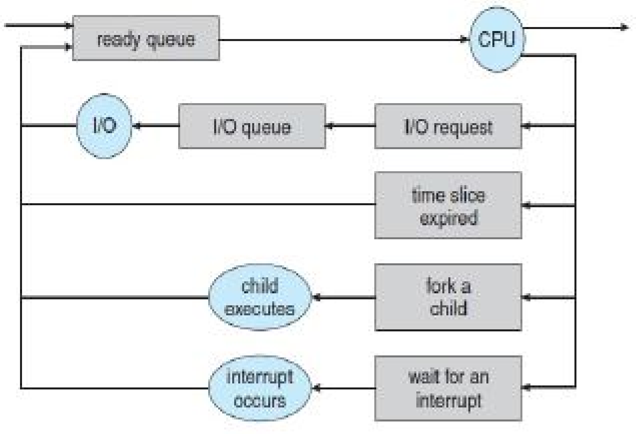


The ready queue and various I/O device queues.

This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk.

Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.



Queueing-diagram representation of process scheduling.

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

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

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

• The process could create a new child process and wait for the child's termination.

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

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

Schedulers:

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

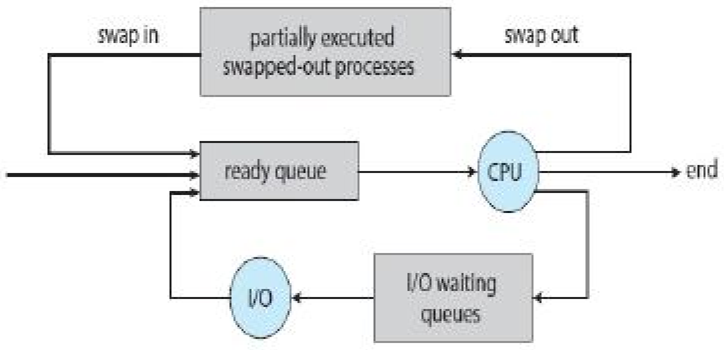
Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.

The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The 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).

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



Context Switch:

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch**.

When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

Context-switch time is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

Operations on Processes

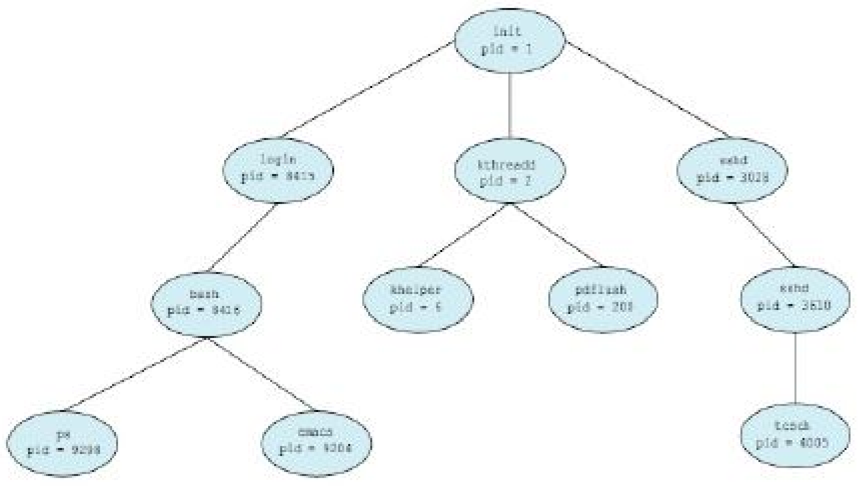
The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

In this section, we explore the mechanisms involved in creating processes and illustrate process creation on UNIX and Windows systems.

Process Creation

During the course of execution, a process may create several new processes. As mentioned earlier, the creating process is called a **parent process**, and the new processes are called the **children of that process.**

Each of these new processes may in turn create other processes, forming a **tree** of processes.



A tree of processes on a typical Linux system.

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 With in the kernel.

In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.

A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.

Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

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:

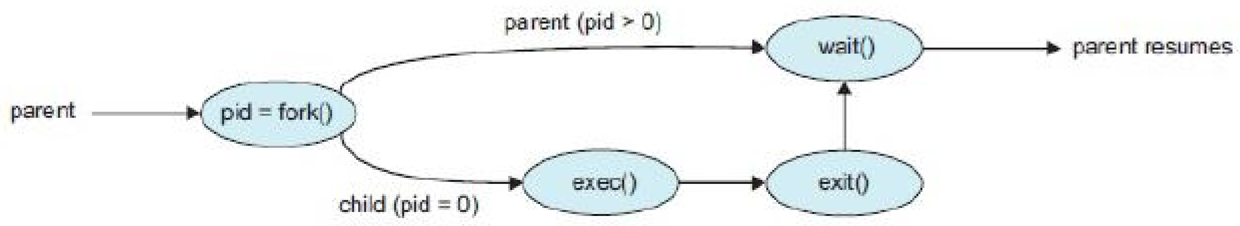
1. The child process is a duplicate of the parent process (same program and data as the parent).
2. The child process has a new program loaded into it.

```
#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;
}
```

The C program shown in Figure illustrates the UNIX system calls previously described. We now have two different processes running copies of the same program. The only difference is that the value of pid (the process identifier) for the child process is zero, while that for the parent is an integer value greater than zero (in fact, it is the actual pid of the child process).

The child process inherits privileges and scheduling attributes from the parent, as well certain resources, such as open files. The child process then overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).

The parent waits for the child process to complete with the wait() 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 the following Figure.



Process creation using the fork() system call.

Of course, there is nothing to prevent the child from ***not*** invoking exec()and continuing to execute as a copy of the parent process. In this scenario, the parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).

All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, Terminate Process() 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()).

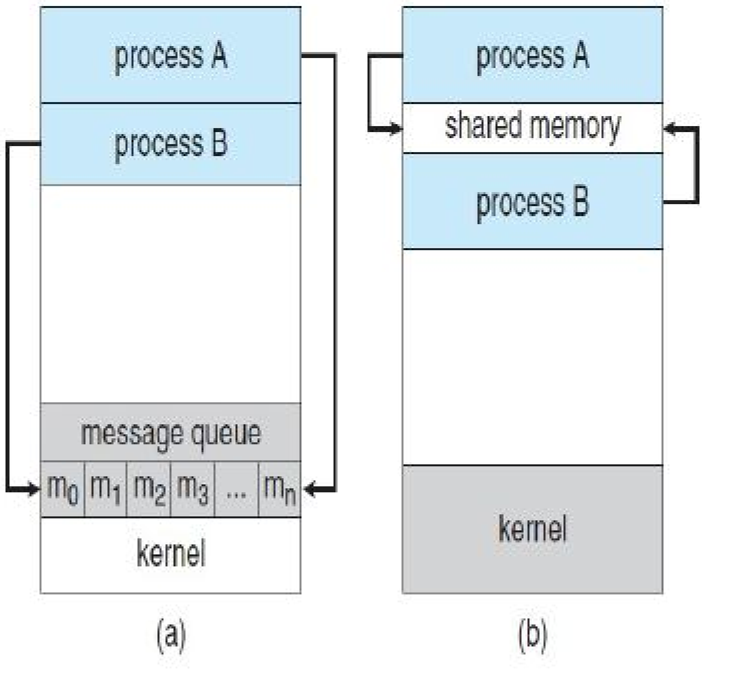
<u>Interprocess Communication</u>

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

A process is ***<u>independent</u>*** if it cannot affect or be affected by the other processes executing in the system.

Any process that does not share data with any other process is independent.

A process is ***<u>cooperating</u>*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.



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

<u>There are several reasons</u> for providing an environment that allows process cooperation:

• **Information sharing**. Since several users may be interested in the same piece of information. We must provide an environment to allow concurrent access to such information.

• **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

• **Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

• **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication:

- **Shared memory**
- **Message passing**.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

The two communications models are contrasted in above Figure.

Both of the models just mentioned are common in operating systems, and many systems implement both.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement in a distributed system than shared memory.

Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

They can then exchange information by reading and writing data in the shared areas.

The form of the data and the location are determined by these processes and are not under the operating system's control.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

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

For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader.

The producer – consumer problem also provides a useful metaphor for the client – server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

One solution to the producer – consumer problem uses shared memory.

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

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

The producer process using shared memory.

Two types of buffers can be used.

The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems

The shared-memory environment requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer.

Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.

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 only fixed-sized messages can be sent, the system-level implementation is straight-forward. This restriction, however, makes the task of programming more difficult.

Conversely, variable-sized messages require a more complex system- level implementation, but the programming task becomes simpler.

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. We are concerned here not with the link's physical implementation (such as shared memory, hardware bus, or network) but rather with its logical implementation.

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.

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**. (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)

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

Buffering

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

Zero capacity. The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
Bounded capacity. The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
Unbounded capacity. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Multi-Threaded Programming

Thread

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. The following figure shows the working of a single-threaded and a multithreaded process.

Difference between Process and Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.

5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

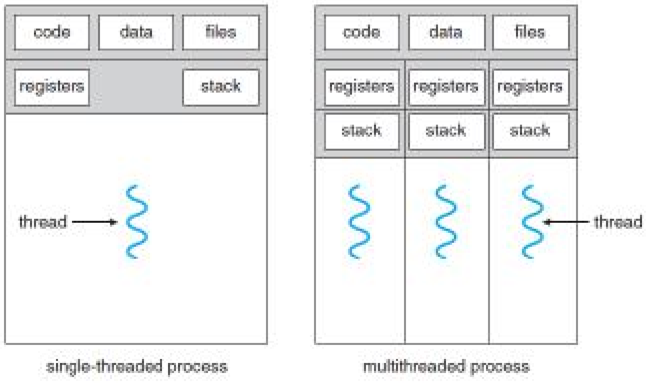
- Threads minimize the context switching time.
- Use of threads provides concurrency within a process.
- Efficient communication.
- It is more economical to create and context switch threads.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.



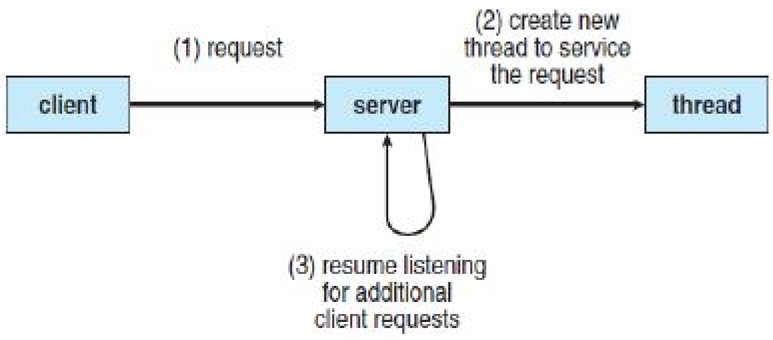
A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. The following Figure illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control.

A web browser might have one thread display images or text while another thread retrieves data from the network.

For example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background. Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

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. In fact, this process-creation method was in common use before threads became popular. 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.

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

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

- **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing.
- **Economy**. Allocating memory and resources for process creation is costly. Because threads

share the resources of the process to which they belong, it is more economical to create and context-switch threads.

- **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

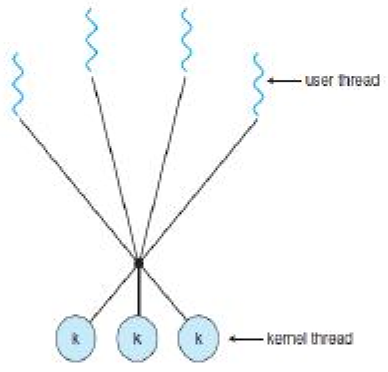
Multithreading Models

Some operating system provide a combined user level thread and Kernel level thread facility.

Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

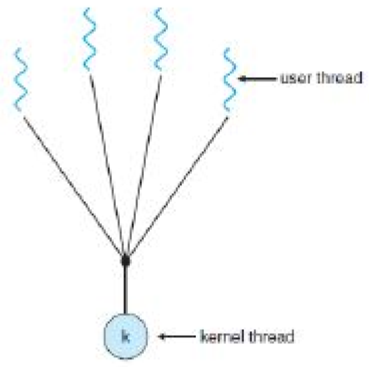
- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Model



The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.

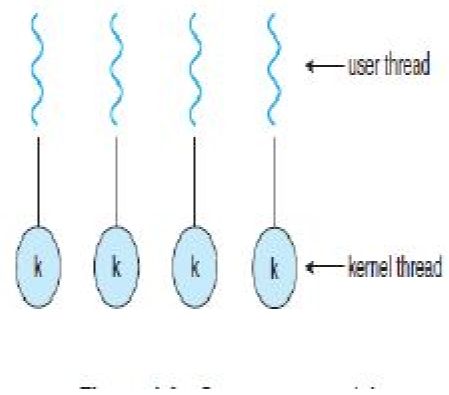


Many to One Model

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.

One to One Model



There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.

Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

<u>Threading Issues</u>

<u>The fork( ) and exec( ) System Calls</u>

Recall that when fork() is called, a separate, duplicate process is created

• How should fork() behave in a multithreaded program? - Should all threads be duplicated?
  - Should only the thread that made the call to fork() be duplicated?

• In some systems, different versions of fork() exist depending on the desired behavior

- Some UNIX systems have fork1() and forkall() • fork1() only duplicates the calling thread
• forkall() duplicates all of the threads in a process
- In a POSIX-compliant system, fork() behaves the same as fork1()
• The exec() system call continues to behave as expected - Replaces the entire process that called it, including all threads
• If planning to call exec() after fork(), then there is no need to duplicate all of the threads in the calling process - All threads in the child process will be terminated when exec() is called
- Use fork1(), rather than forkall() if using in conjunction with exec()

Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred
- CTRL-C is an example of an asynchronous signal that might be sent to a process
• An asynchronous signal is one that is generated from outside the process that receives it
- Divide by 0 is an example of a synchronous signal that might be sent to a process
• A synchronous signal is delivered to the same process that caused the signal to occur
• All signals follow the same basic pattern: - A signal is generated by particular event
- The signal is delivered to a process
- The signal is handled by a signal handler (all signals are handled exactly once)
Signal handling is straightforward in a single-threaded process - The one (and only) thread in the process receives and handles the signal
In a multithreaded program, where should signals be delivered? - Options:
(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 only to certain threads in the process
(4) Assign a specific thread to receive all signals for the process
• Option 1 - Deliver the signal to the thread to which the signal applies - Most likely option when handling synchronous signals (e.g. only the thread that attempts to divide by zero needs to know of the error)
• Option 2 - Deliver the signal to every thread in the process - Likely to be used in the event that the process is being terminated (e.g. a CTRLC is sent to terminate the process, all threads need to receive this signal and terminate)

Thread Cancellation

• Thread cancellation is the act of terminating a thread before it has completed - Example - clicking the stop button on your web browser will stop the thread that is rendering the web page
• The thread to be cancelled is called the target thread
• Threads can be cancelled in a couple of ways - Asynchronous cancellation terminates the target thread immediately
• Thread may be in the middle of writing data ... not so good

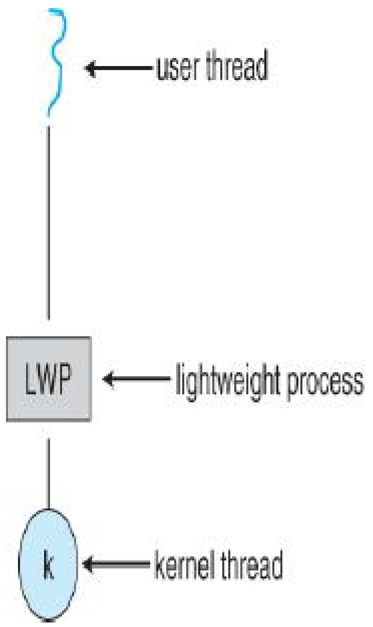
- Deferred cancellation allows the target thread to periodically check if it should be cancelled
• Allows thread to terminate itself in an orderly fashion

- Threads that are no longer needed may be cancelled by another thread in one of two ways:
    1. **Asynchronous Cancellation** cancels the thread immediately.
    2. **Deferred Cancellation** sets a flag indicating the thread should cancel itself when it is convenient. It is then up to the cancelled thread to check this flag periodically and exit nicely when it sees the flag set.
- (Shared) resource allocation and inter-thread data transfers can be problematic with asynchronous cancellation.

Thread-Local Storage

- Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
- However sometimes threads need thread-specific data also.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as **thread-local storage** or **TLS.** Note that this is more like static data than local variables, because it does not cease to exist when the function ends.

Scheduler Activations



- Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many or two-tier models.
- This virtual processor is known as a "Lightweight Process", LWP.
    - There is a one-to-one correspondence between LWPs and kernel threads.
    - The number of kernel threads available, (and hence the number of LWPs) may change dynamically.
    - The application (user level thread library) maps user threads onto available LWPs.
    - Kernel threads are scheduled onto the real processor(s) by the OS.
    - The kernel communicates to the user-level thread library when certain events occur (such as a thread about to block) via an **upcall**, which is handled in the thread library by an **upcall handler**. The upcall also provides a new LWP for the upcall handler to run on, which it can then use to reschedule the user thread that is about to become blocked. The OS will also issue

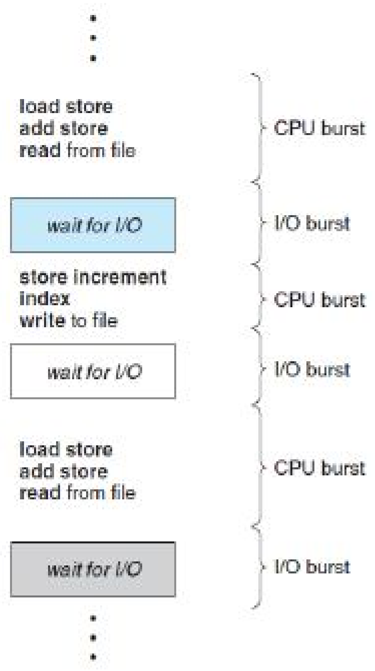
upcalls when a thread becomes unblocked, so the thread library can make appropriate adjustments.

- If the kernel thread blocks, then the LWP blocks, which blocks the user thread.
- Ideally there should be at least as many LWPs available as there could be concurrently blocked kernel threads. Otherwise if all LWPs are blocked, then user threads will have to wait for one to become available.

Process Scheduling

CPU Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. We introduce basic CPU-scheduling concepts and present several CPU- scheduling algorithms. We also consider the problem of selecting an algorithm for a particular system.

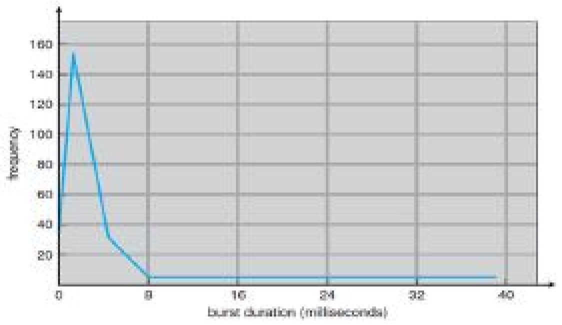


Basic Concepts

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time.

When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.



CPU – I/O Burst Cycle

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

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Fig. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler**, or CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

Preemptive Scheduling

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

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

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

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state. This scheduling method was used by Microsoft Windows 3.x. Windows 95 introduced preemptive scheduling, and all subsequent versions of Windows operating systems have used preemptive scheduling. The Mac OS X operating system for the Macintosh also uses preemptive scheduling; previous versions of the Macintosh operating system relied on cooperative scheduling. Cooperative scheduling is the only method that can be used on certain hardware platforms, because it does not require the special hardware (for example, a timer) needed for preemptive scheduling.

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

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Scheduling Criteria

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

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

- **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.
- **Turnaround time**. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

Cpu-Scheduling Algorithms

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

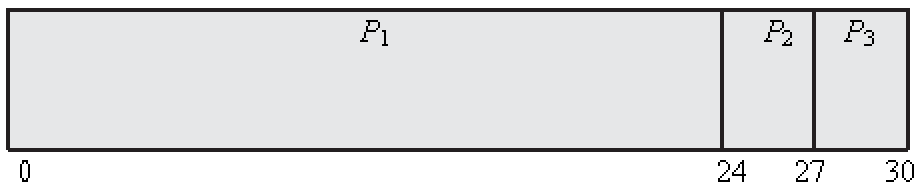
First-Come, First-Served Scheduling

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.

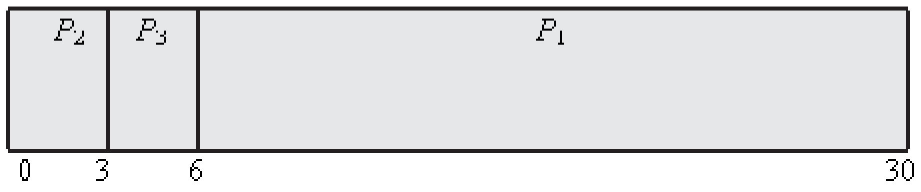
On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P_1 , 24 milliseconds for process P_2 , and 27 milliseconds for process P_3. Thus, the average waiting time is (0+24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P_2, P_3 , P_1, however, the results will be as shown in the following Gantt chart:



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

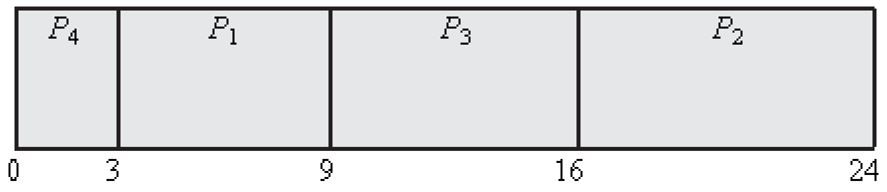
Shortest-Job-First Scheduling

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

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

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

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



The waiting time is 3 milliseconds for process P_1, 16 milliseconds for process P_2, 9 milliseconds for process P_3, and 0 milliseconds for process P_4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

Priority Scheduling

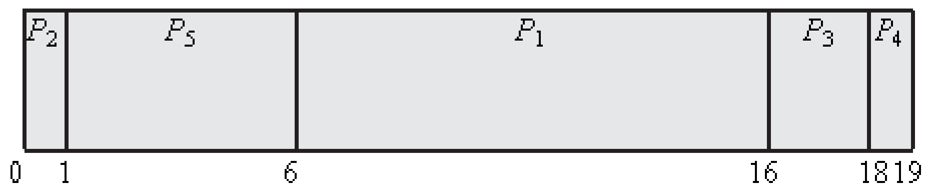
The SJF algorithm is a special case of the general **priority-scheduling** algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where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

Note that we discuss scheduling in terms of ***high*** priority and ***low*** priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

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

Process	Burst Time	Priority
P1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P5	5	2

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



The average waiting time is 8.2 milliseconds.

Priorities can be defined either internally or externally.

Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities.

External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice,** is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

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

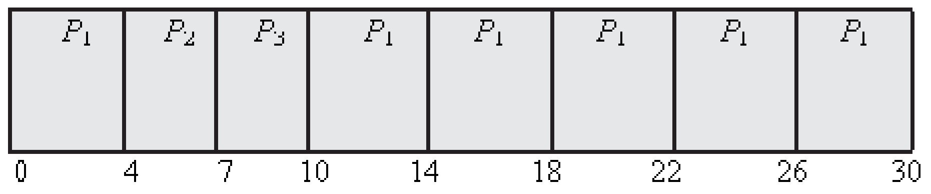
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

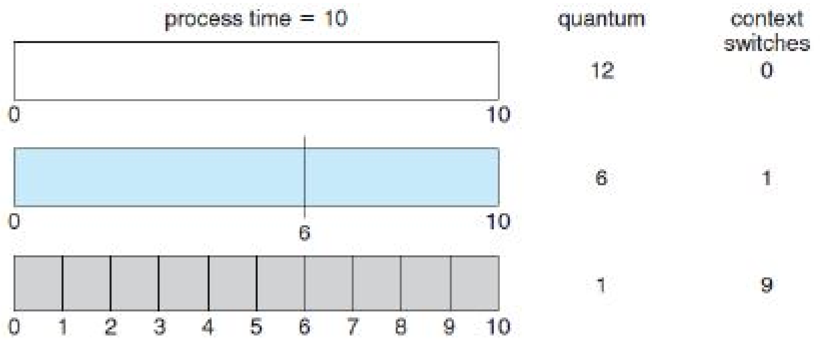
If we use a time quantum of 4 milliseconds, then process P_1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P_1 waits for 6 milliseconds (10 - 4), P_2 waits for 4 milliseconds, and P_3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

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

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units. Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.



The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy

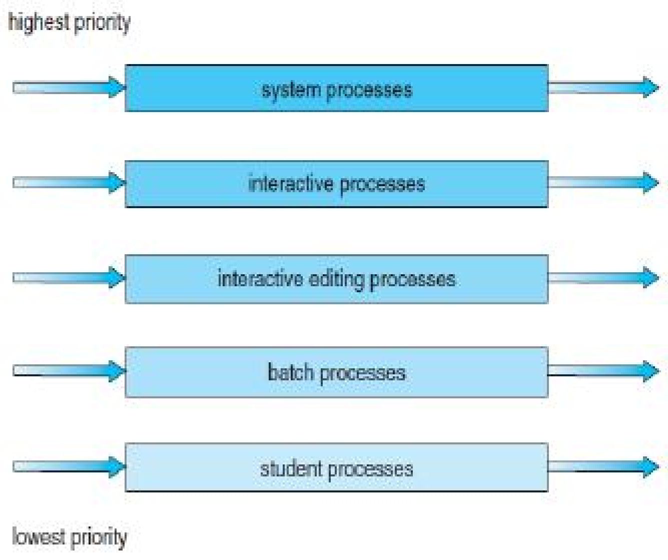
is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly.

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

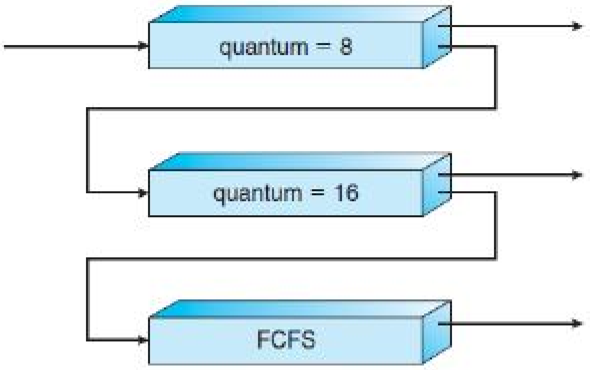
For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.



Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

System processes Interactive processes Interactive editing processes Batch processes Student processes

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



Multilevel Feedback Queue Scheduling

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

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

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

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

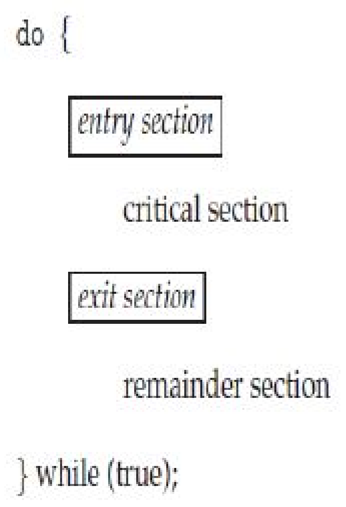
Process Synchronization

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

<u>The Critical-Section Problem</u>

We begin our consideration of process synchronization by discussing the so-called critical-section problem. Consider a system consisting of *n* processes {P_0, P_1... P_{n-1}}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and soon.

The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.



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

The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.

The section of code implementing this request is the **entry section**.

The critical section may be followed by an **exit section**.

The remaining code is the **remainder section**.

The general structure of a typical process P_i is shown in Fig. The entry section and exit section are enclosed in boxes to highlight these important segments of code.

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

- **Mutual exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
- **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

We assume that each process is executing at a nonzero speed. However, we can make no assumption concerning the relative speed of the *n* processes.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (***kernel code***) is subject to several possible race conditions. Consider as an example a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list) . If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling. It is up to kernel developers to ensure that the operating system is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems:

Preemptive kernels. A preemptive kernel allows a process to be preempted while it is running in kernel mode.

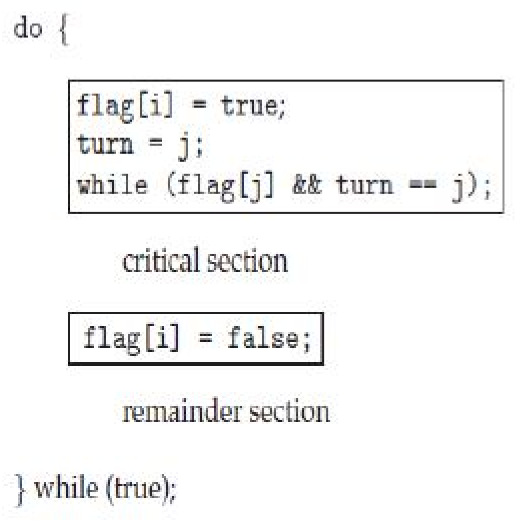
A **nonpreemptive kernel** does not allow a process running in kernel mode to be preempted; a kernel- mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Obviously, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

<u>Peterson's Solution</u>

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

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For



```
do {

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

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

convenience, when presenting P_i , we use P_j to denote the other process; that is, j equals 1 − i.

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

int turn;
boolean flag[2];

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

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

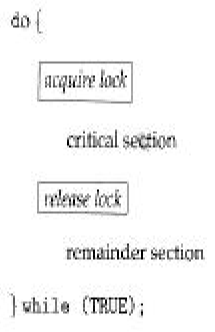
We now prove that this solution is correct. We need to show that:

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

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

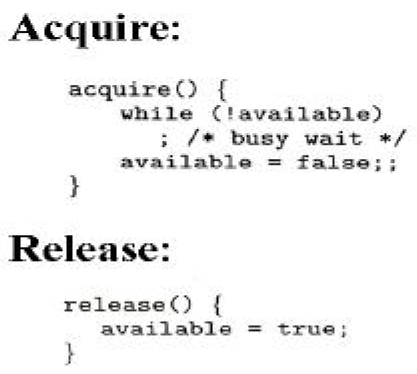
Mutex Locks:

- The hardware solutions presented above are often difficult for ordinary programmers to access, particularly on multi-processor machines, and particularly because they are often platform-dependent.



- Therefore most systems offer a software API equivalent called ***mutex locks*** or simply ***mutexes.*** (For mutual exclusion)
- The terminology when using mutexes is to ***acquire*** a lock prior to entering a critical section, and to ***release* it** when exiting, as shown in Figure

- Just as with hardware locks, the acquire step will block the process if the lock is in use by another process, and both the acquire and release operations are atomic.
- Acquire and release can be implemented as shown here, based on a boolean variable "available":



Acquire:

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

Release:

```
release() {
   available = true;
}
```

- One problem with the implementation shown here, (and in the hardware solutions presented earlier), is the busy loop used to block processes in the acquire phase. These types of locks are referred to as ***spinlocks***, because the CPU just sits and spins while blocking the process.
- Spinlocks are wasteful of cpu cycles, and are a really bad idea on single-cpu single-threaded machines, because the spinlock blocks the entire computer, and doesn't allow any other process to release the lock. (Until the scheduler kicks the spinning process off of the cpu.)
- On the other hand, spinlocks do not incur the overhead of a context switch, so they are effectively used on multi-threaded machines when it is expected that the lock will be released after a short time.

<u>Semaphores</u>

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P (from the Dutch ***proberen,*** “to test”); signal() was originally called V (from ***verhogen,*** “to increment”). The definition of wait() is as follows:

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

The definition of signal() is as follows:

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

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

Semaphore Usage

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

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

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore synch, initialized to 0. In process P_1, we insert the statements

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

In process P_2, we insert the statements

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

Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

Semaphore Implementation

The definitions of the wait() and signal() semaphore operations just described present the same problem. To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

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

```
typedef struct {

    int value;

    struct process *list;

} semaphore;
```

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

Now, the wait() semaphore operation can be defined as

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

and the signal() semaphore operation can be defined as

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

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

Deadlocks and Starvation

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

To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

```
      P0              P1
   wait(S);        wait(Q);
   wait(Q);        wait(S);
      .               .
      .               .
      .               .
   signal(S);      signal(Q);
   signal(Q);      signal(S);
```

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S) , it must wait until P_0 executes signal(S). Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. Other types of events may result in deadlocks, as we show in Chapter 7. In that chapter, we describe various mechanisms for dealing with the deadlock problem.

Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process — or a chain of lower-priority processes. Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

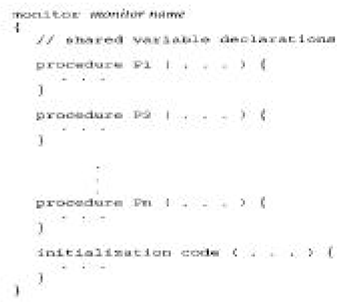
As an example, assume we have three processes — *L* , *M*, and *H* — whose priorities follow the order $L < M < H$. Assume that process *H* requires resource *R*, which is currently being accessed by process *L*. Ordinarily, process *H* would wait for *L* to finish using resource *R*. However, now suppose that process *M* becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority — process *M*— has affected how long process *H* must wait for *L* to relinquish resource *R*.

This problem is known as **priority inversion**. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. That is insufficient for most general-purpose operating systems, however. Typically these systems solve the problem by implementing a **priority-inheritance protocol**.

According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process *H*, thereby preventing process *M* from preempting its execution. When process had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority. Because resource *R* would now be available, process *H* — not *M*— would run next.

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. (And since concurrency problems are by definition rare events, the problem code may easily go unnoticed and/or be heinous to debug.)

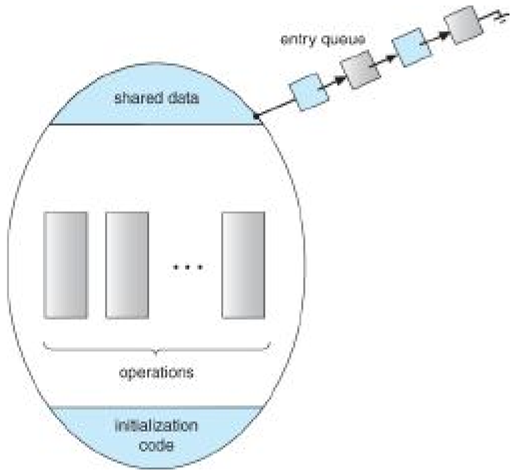


```
monitor monitor name
{
    // shared variable declarations
    procedure P1 ( . . . ) {
      . . .
    }
    procedure P2 ( . . . ) {
      . . .
    }
          .
          .
          .
    procedure Pn ( . . . ) {
      . . .
    }
    initialization code ( . . . ) {
      . . .
    }
}
```

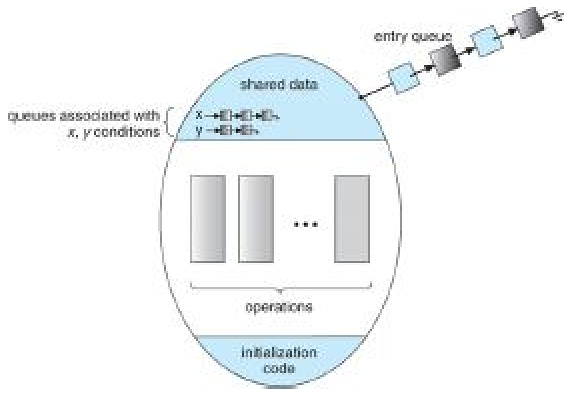
- For this reason a higher-level language construct has been eveloped, called ***monitors***.

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.



- 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, then legal operations would be X.wait( ) and X.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.)



- 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.

<u>Classic Problems of Synchronization</u>

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

The Bounded-Buffer Problem

The ***bounded-buffer problem*** is commonly used to illustrate the power of synchronization primitives. Here, we present a general structure of this scheme without committing ourselves to any particular implementation. We provide a related programming project in the exercises at the end of the chapter.

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

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

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

The code for the producer process is shown in Fig, and the code for the consumer process is shown in Fig. Note the symmetry between the producer and the consumer. We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

```
do {
    wait(full);
    wait(mutex);
        . . .
   /* remove an item from buffer to next consumed */
            . . .
    signal(mutex);
    signal(empty);
```

```
        . . .
      /* consume the item in next consumed */
          . . .
  } while (true);
```

The structure of the consumer process.

The Readers – Writers Problem

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

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers – writers problem has several variations, all involving priorities. The simplest one, referred to as the ***first*** readers – writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The ***second*** readers– writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers – writers problem.

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

```
semaphore rw mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

The semaphores mutex and rw mutex are initialized to 1; read count is initialized to 0. The semaphore rw mutex is common to both reader and writer

```
do {
   wait(rw mutex);
```

```
     . . .
/* writing is performed */
     . . .
signal(rw mutex);
} while (true);
```

Fig-The structure of a writer process.

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

The code for a writer process is shown in Fig, the code for a reader process is shown in Fig. Note that, if a writer is in the critical section and *n* readers are waiting, then one reader is queued on rw mutex, and *n* − 1 readers are queued on mutex. Also observe that, when a writer executes signal(rw mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers – writers problem and its solutions have been generalized to provide **reader–writer** locks on some systems. Acquiring a reader – writer lock requires specifying the mode of the lock: either *read* or *write* access. When a process wishes only to read shared data, it requests the reader – writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader – writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

<u>**Dead Locks**</u>

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas legislature early in the 20th century. It said, in part: **"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."**

Although some applications can identify programs that may deadlock, operating systems typically do not provide deadlock-prevention facilities, and it remains the responsibility of programmers to ensure that they design deadlock-free programs.

Deadlock problems can only become more common, given current trends, including larger numbers of processes, multithreaded programs, many more resources within a system, and an emphasis on long-lived file and database servers rather than batch systems.

System Model

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

If a process requests an instance of a resource type, the allocation of ***any*** instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly.

For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

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

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

- **Request**. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- **Use**. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- **Release**. The process releases the resource.

The request and release of resources may be system calls,. **Examples are** the request() and release() device, open() and close() file, and allocate() and free() memory system calls.

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

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

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

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

Deadlock Characterization

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

Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultane-ously in a system:

- **Mutual exclusion**. At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
- **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
- **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait**. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**.

This graph consists of a set of vertices V and a set of edges E.

The set of vertices V is partitioned into two different types of nodes: $P = \{P_1 , P_2 , ..., P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1 , R_2 , ..., R_m\}$, the set consisting of all resource types in the system.

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

A directed edge $P_i \rightarrow R_j$ is called a **request edge**; a directed edge $R_j \rightarrow P_i$ is called an **assignment edge**.

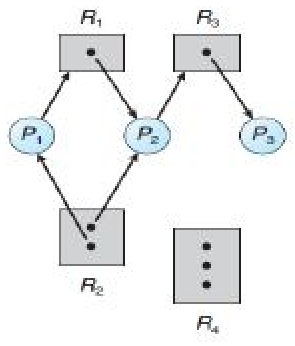
Pictorially, we represent each process P_i as a circle and each resource type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle. Note that a request edge points to only the rectangle R_j , whereas an assignment edge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

When process P_i requests an instance of resource type R_j , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is ***instantaneously*** transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

The resource-allocation graph shown in Fig depicts the following situation.

The sets P, R, and E:

$P = \{P_1, P_2, P_3\}$
$R = \{R_1 , R_2 , R_3, R_4 \}$
◦ $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3 , R_1 \rightarrow P_2 , R_2 \rightarrow P_2 , R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$



Resource instances:

One instance of resource type R_1

Two instances of resource type R_2

One instance of resource type R_3

Three instances of resource type R_4

Process states:

Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.

Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.

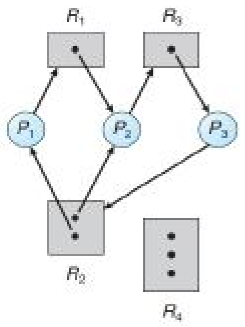
Process P_3 is holding an instance of R_3.

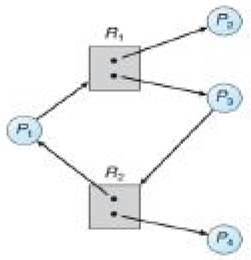
Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred. If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Fig Suppose that process P_3 requests an instance of resource





type R_2. Since no resource instance is currently available, we add a request edge $P_3 \rightarrow R_2$ to the graph. At this point, two minimal cycles exist in the system:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \; P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Processes P_1 , P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3 , which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2 . In addition, process P_1 is waiting for process P_2 to release resource R_1 .

Now consider the resource-allocation graph in Figure 7.3. In this example, we also have a cycle:

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

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3 , breaking the cycle.

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

Methods for Handling Deadlocks

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

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will ***never*** enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

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

To ensure that deadlocks never occur, the system can use either a deadlock-prevention or a deadlock-avoidance scheme. **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Deadlock Prevention

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

Mutual Exclusion

The mutual exclusion condition must hold. That is, at least one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.

Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

A process never needs to wait for a sharable resource. In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable.

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

Hold and Wait

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

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

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

The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file. The process must then request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. In the example given, for instance, we can release the DVD drive and disk file, and then request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.

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

No Preemption

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

Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let $R = \{ R_1, R_2, ..., R_m\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

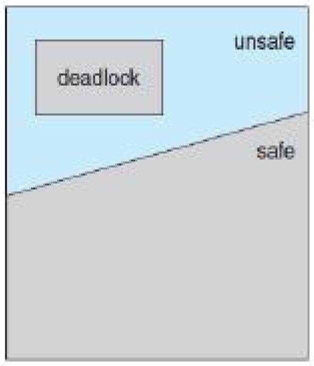
F (tape drive) = 1
F (disk drive) = 5
F (printer) = 12

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

Deadlock Avoidance

An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.

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



Safe State

A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**.

A sequence of processes $<P_1, P_2, ..., P_n>$ is a safe sequence for the current allocation state if, for each P_i , the resource requests that P_i can still make can be satisfied by the currently available resources plus the resources held by all P_j , with $j < i$.

In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

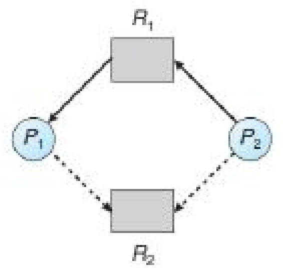
A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however. An unsafe state ***may*** lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states. In an unsafe state, the operating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

To illustrate, we consider a system with twelve magnetic tape drives and three processes: P_0, P_1, and P_2 . Process P_0 requires ten tape drives, process P_1 may need as many as four tape drives, and process P_2 may need up to nine tape drives. Suppose that, at time t_0 , process P_0 is holding five tape drives, process P_1 is holding two tape drives, and process P_2 is holding two tape drives. (Thus, there are three free tape drives.)

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
$P2$	9	2

At time t_0, the system is in a safe state. The sequence <P_1 , P_0 , P_2> satisfies the safety condition. Process P_1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process P_0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process P_2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

Resource-Allocation-Graph Algorithm



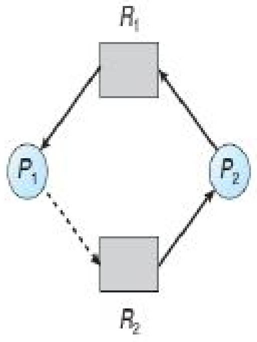
If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph for deadlock avoidance.

In addition to the request and assignment edges already described, we introduce a new type of edge, called a **claim edge**.

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line. When process P_i requests resource R_j , the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by P_i , the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

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

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



If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P_i will have to wait for its requests to be satisfied.

To illustrate this algorithm, we consider the resource- allocation graph of Fig. Suppose that P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph. A cycle, as mentioned, indicates that the system is in an unsafe state. If P_1 requests R_2, and P_2 requests R_1, then a deadlock will occur.

Banker's Algorithm

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

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker 's algorithm. These data structures encode the state of the resource-allocation system. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

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

Max. An $n \times m$ matrix defines the maximum demand of each process. If ***Max***$[i][j]$ equals k, then process P_i may request at most k instances of resource type R_j .

Allocation. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***$[i][j]$ equals k, then process P_i is currently allocated k instances of resource type R_j .

Need. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***$[i][j]$ equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that ***Need***$[i][j]$ equals ***Max***$[i][j]$ − ***Allocation***$[i][j]$.

These data structures vary over time in both size and value.

To simplify the presentation of the banker 's algorithm, we next establish some notation. Let X and Y be vectors of length n. We say that $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all $i = 1, 2, ..., n$. For example, if $X = (1,7,3,2)$ and $Y = (0,3,2,1)$, then $Y \leq X$. In addition, $Y < X$ if $Y \leq X$ and $Y \neq X$.

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

Safety Algorithm

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

```
Let Work and Finish be vectors of length $m$ and $n$, respectively. Initialize Work =
Available and Finish[$i$] = false for $i = 0, 1, ..., n - 1$.
Find an index $i$ such that both
    Finish[$i$] == false
    Need$_i \leq$ Work
  If no such $i$ exists, go to step 4.
  Work = Work + Allocation$_i$ Finish[$i$] =
  true
```

Go to step 2.
If ***Finish***[i] == ***true*** for all i, then the system is in a safe state.

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

Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted.

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

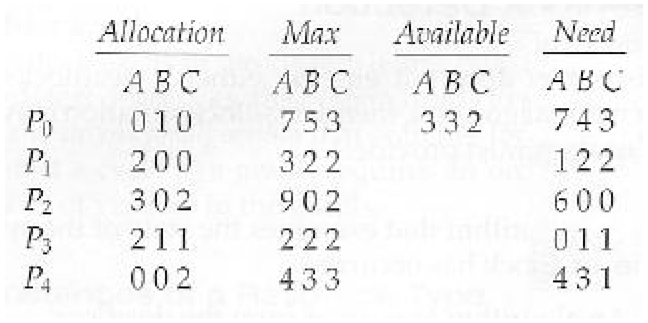
If ***Request***$_i$ ≤ ***Need***$_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
If ***Request***$_i$ ≤ ***Available,*** go to step 3. Otherwise, P_i must wait, since the resources are not available.
Have the system pretend to have allocated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

Available = Available   Request$_i$;
Allocation$_i$ ***= Allocation***$_i$ ***+ Request***$_i$;
Need$_i$ ***= Need***$_i$ ***Request***$_i$;

If the resulting resource-allocation state is safe, the transaction is com-pleted, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for ***Request***$_i$ **,** and the old resource-allocation state is restored.

An Illustrative Example

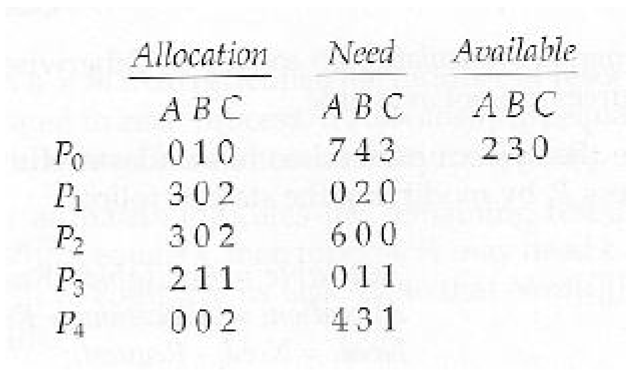
- Consider the following situation:



	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

The system is in a safe state since the sequence < P1, P3, P4, P2, P0> satisfies safety criteria

- 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

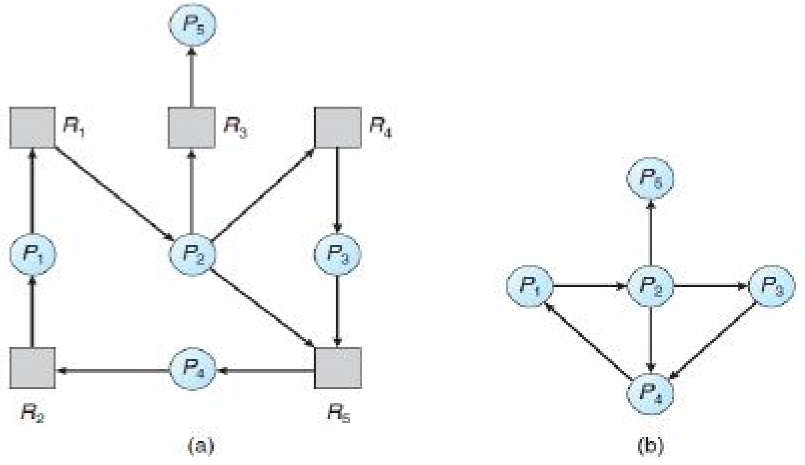
If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

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

Single Instance of Each Resource Type

If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.



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

As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to ***maintain*** the wait-for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

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

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

To simplify notation, we again treat the rows in the matrices ***Allocation*** and ***Request*** as vectors; we refer to them as ***Allocation***$_i$ and ***Request***$_i$ **.** The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed. Compare this algorithm with the banker's algorithm.

Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available.*** For $i = 0, 1, ..., n - 1$, if ***Allocation***$_i \neq 0$, then ***Finish***$[i]$ = ***false.*** Otherwise, ***Finish***$[i]$ = ***true.***

Find an index i such that both

Finish$[i]$ == ***false***

Request$_i \leq$ ***Work***

If no such i exists, go to step 4.

Work − ***Work*** + ***Allocation***$_i$ ***Finish***$[i]$ = ***true***

Go to step 2.

If ***Finish***$[i]$ == ***false*** for some i, $0 \le i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***$[i]$ == ***false***, then process P_i is deadlocked.

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

You may wonder why we reclaim the resources of process P_i (in step 3) as soon as we determine that ***Request**$_i$* $\le$ ***Work*** (in step 2b). We know that P_i is currently ***not*** involved in a deadlock (since ***Request**$_i$* $\le$ ***Work***). Thus, we take an optimistic attitude and assume that P_i will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P_0 through P_4 and three resource types *A*, *B*, and *C*. Resource type *A* has seven instances, resource type *B* has two instances, and resource type *C* has sixinstances. Suppose that, at time T_0, we have the following resource-allocation state:

	Allocation	***Request***	***Available***
	A B C	*A B C*	*A B C*
*P*0	0 1 0	0 0 0	0 0 0
*P*1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence $<P_0$, P_2, P_3 , P_1, $P_4>$ results in ***Finish***$[i]$ == *true* for all *i*.

Suppose now that process P_2 makes one additional request for an instance of type *C*. The ***Request*** matrix is modified as follows:

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

We claim that the system is now deadlocked. Although we can reclaim the resources held by process P_0 , the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3 , and P_4 .

Detection-Algorithm Usage

When should we invoke the detection algorithm? The answer depends on two factors:

How ***often*** is a deadlock likely to occur?
How ***many*** processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Deadlocks occur only when some process makes a request that cannot be granted immediately. This request may be the final request that completes a chain of waiting processes. In the extreme, then, we can invoke the deadlock-detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of processes but also the specific process that “caused” the deadlock. (In reality, each of the deadlocked processes is a link in the cycle in the resource graph, so all of them, jointly, caused the deadlock.) If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and “caused” by the one identifiable process.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, several alter-natives are available. One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually. Another possibility is to let the system **recover** from the deadlock automatically.

There are two options for breaking a deadlock.

One is simply to abort one or more processes to break the circular wait.

The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

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

- **Abort all deadlocked processes**.
  This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated**.
  This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state. Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions. The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost.

Unfortunately, the term ***minimum cost*** is not a precise one. Many factors may affect which process is chosen, including:

What the priority of the process is
How long the process has computed and how much longer the process will compute before completing its designated task
How many and what types of resources the process has used (for example, whether the resources are simple to preempt)
How many more resources the process needs in order to complete
How many processes will need to be terminated
Whether the process is interactive or batch

Resource Preemption

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

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

- **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.
- **Rollback**. If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.
- **Starvation**. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

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

Unit III - Memory Management

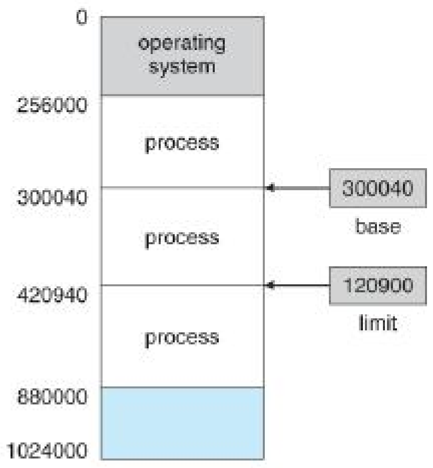
Background

- Obviously memory accesses and memory management are a very important part of modern computer operation. Every instruction has to be fetched from memory before it can be executed, and most instructions involve retrieving data from memory or storing data in memory or both.
- The advent of multi-tasking Operating Systems compounds the complexity of memory management, because as processes are swapped in and out of the CPU, so must their code and data be swapped in and out of memory, all at high speeds and without interfering with any other processes.
- Shared memory, virtual memory, the classification of memory as read-only versus read-write, and concepts like copy-on-write forking all further complicate the issue.

Basic Hardware

- It should be noted that from the memory chips point of view, all memory accesses are equivalent. The memory hardware doesn't know what a particular part of memory is being used for, nor does it care. This is almost true of the OS as well, although not entirely.
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it. (Device drivers communicate with their hardware via interrupts and "memory" accesses, sending short instructions for example to transfer data from the hard drive to a specified location in main memory. The disk controller monitors the bus for such instructions, transfers the data, and then notifies the CPU that the data is there with another interrupt, but the CPU never gets direct access to the disk.)
- Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.
- Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. This would require intolerable waiting by the CPU if it were not for an intermediary fast memory ***cache*** built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.
- User processes must be restricted so that they only access memory locations that "belong" to that particular process. This is usually implemented using a base register and a limit register for each process, as shown in Figures below.

- ***Every*** memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated. The OS obviously has access to all existing memory locations, as this is necessary to swap users' code and data in and out of memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.



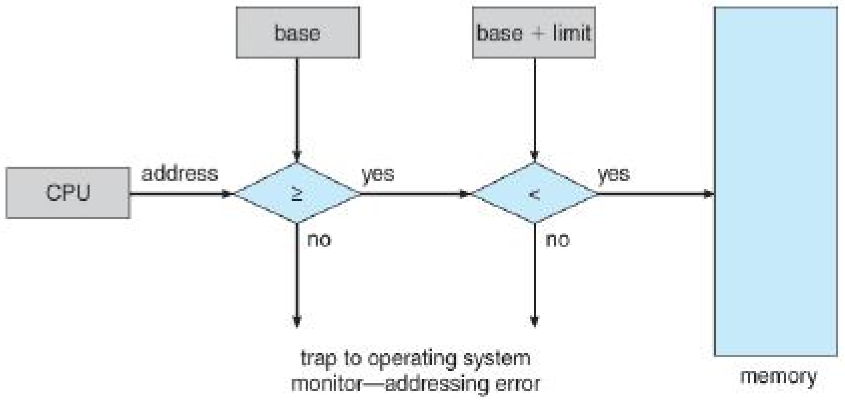
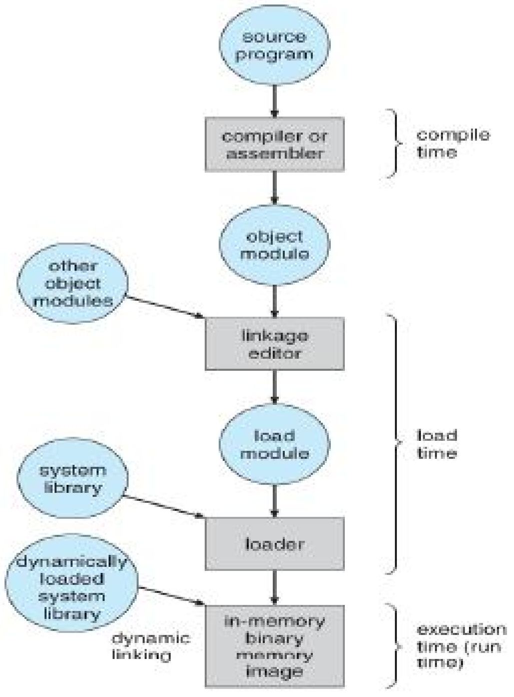


Fig. - Hardware address protection with base and limit registers

Address Binding:



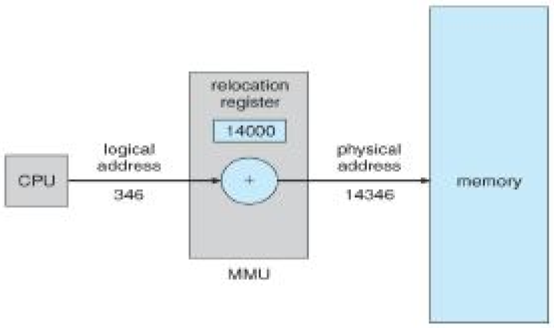
- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:
  - **Compile Time** - If it is known at compile time where 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 Operating Systems.
  - Fig shows the various stages of the binding processes and the units involved in each stage

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.

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 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. (Each process would have their own copy of the ***data*** section of the routines, but that may be small relative to the code segments.) Obviously the OS must manage shared routines in memory.
  - An added benefit of ***dynamically linked libraries*** (***DLLs*** also known as ***shared libraries*** or ***shared objects*** on UNIX systems) involves easy upgrades and updates. When a program uses a routine from a standard library and the routine changes, then the program must be re-built (re-linked) in order to incorporate the changes. However if DLLs are used, then as long as the stub doesn't change, the program can be updated merely by loading new versions of the DLLs onto the system. Version information is maintained in both the program and the DLLs, so that a program can specify a particular version of the DLL if necessary.
  - 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. Further calls to the same routine will access the routine directly and not incur the overhead of the stub access. (Following the UML ***Proxy Pattern***.)

<u>Swapping:</u>

- 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. For efficient processor scheduling the CPU time slice should be significantly longer than this lost transfer time.
- 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.

- Most modern Operating Systems 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. Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.

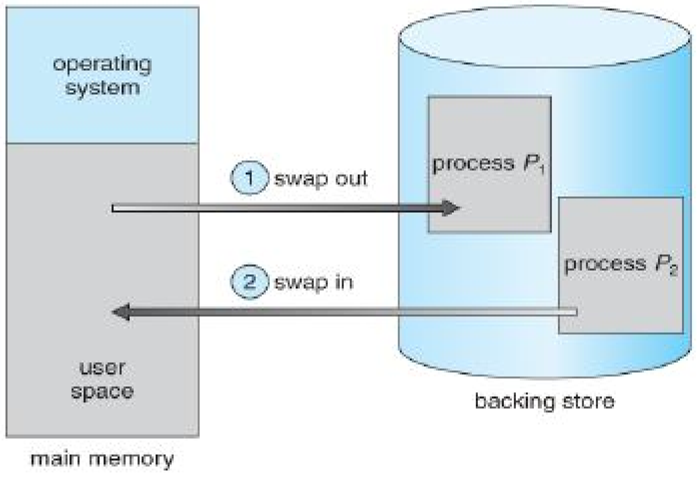


Fig - Swapping of two processes using a disk as a backing store

Swapping on Mobile Systems (New Section in 9th Edition)

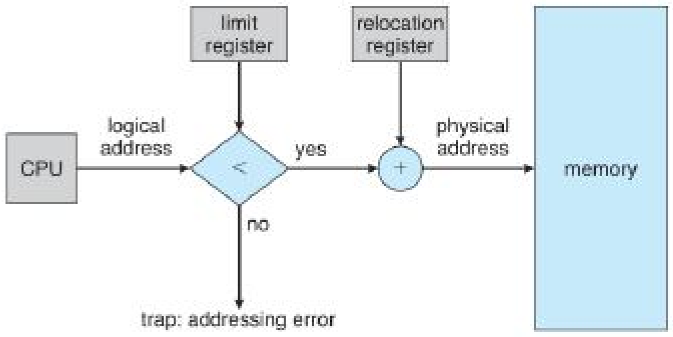
- Swapping is typically not supported on mobile platforms, for several reasons:
  - Mobile devices typically use flash memory in place of more spacious hard drives for persistent storage, so there is not as much space available.
  - Flash memory can only be written to a limited number of times before 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.

<u>Contiguous Memory Allocation</u>

- 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 processes.)

<u>Memory Protection</u> (was Memory Mapping and Protection)



The system shown in Fig 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.

Fig - Hardware support for relocation and limit registers

<u>Memory Allocation</u>

- 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. This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- 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.

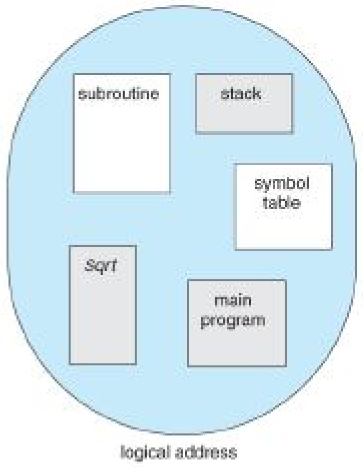
Fragmentation:

- 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 **Programmer's view of a program.**

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

Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The programmer therefore specifies each address by two quantities: a segment name and an offset.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple:*

<segment-number, offset>.

Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

A C compiler might create separate segments for the following:

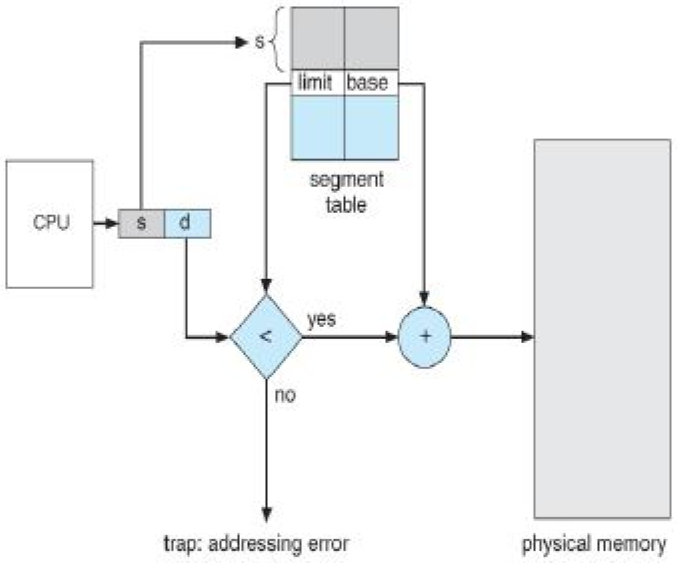
The code
Global variables
The heap, from which memory is allocated
The stacks used by each thread
The standard C library

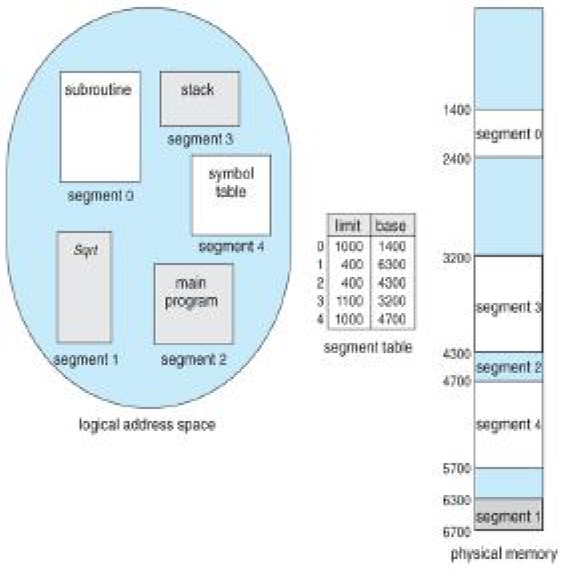
Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers.

Segmentation Hardware

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

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





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

Fig -

Segmentation hardware

Fig - Example of segmentation

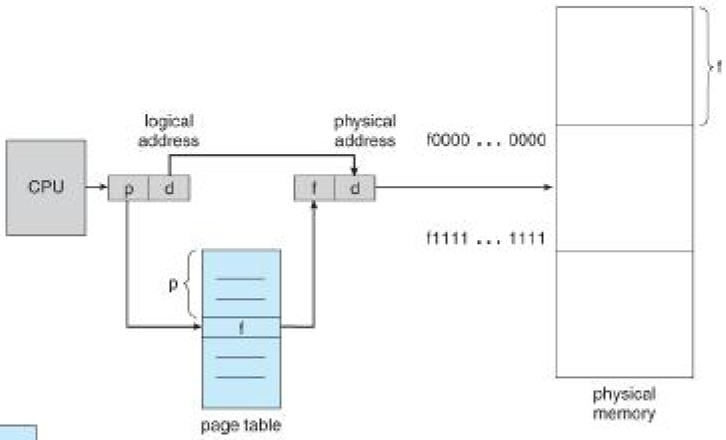
As an example, consider the situation shown in Fig. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of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

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 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:



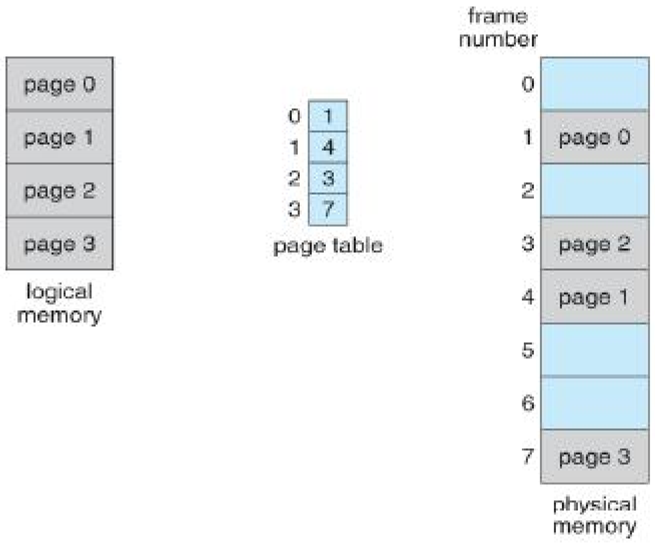
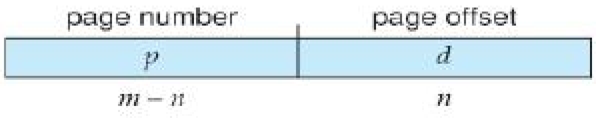


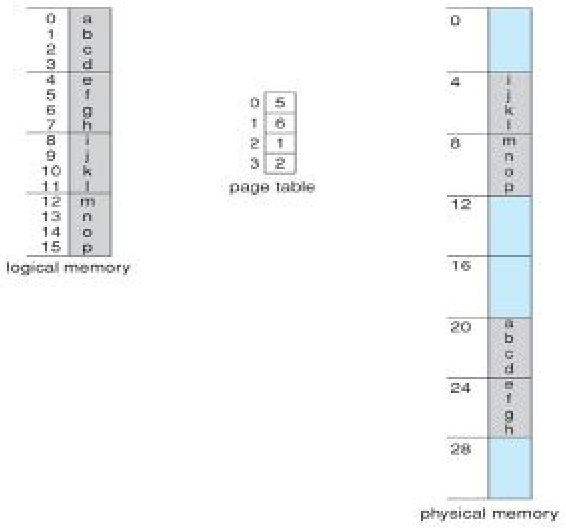
Fig - Paging model of logical and physical memory

- 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.
- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.



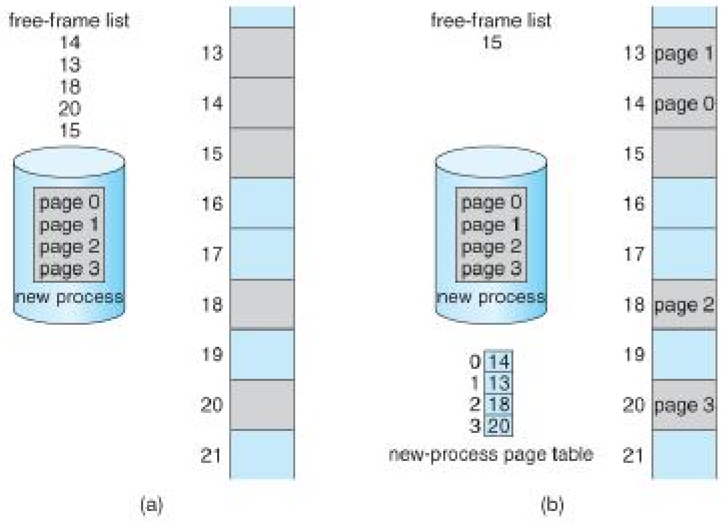
- (DOS used to use an addressing scheme with 16 bit frame numbers and 16-bit offsets, on hardware that only supported 24-bit hardware addresses. The result was a resolution of starting frame addresses finer than the size of a single frame, and multiple frame-offset combinations that mapped to the same physical hardware address.)
- 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. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)



- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- 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. (Possibly more, if processes keep their code and data in separate pages.)

- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.
- 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.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table.

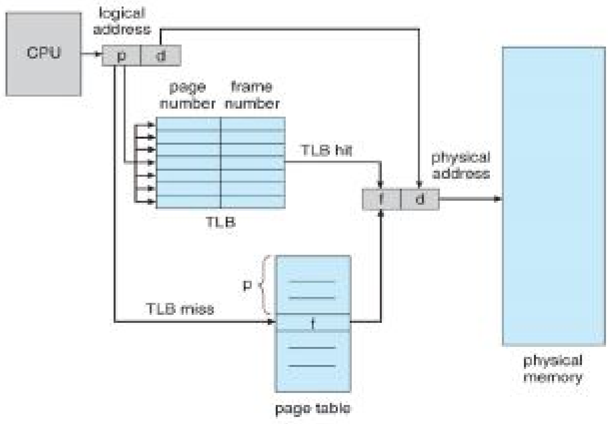
Fig- Free frames (a) before allocation and (b) after allocation

- There is no way for them to generate an address that maps into any other process's memory space.
- 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. (The currently active page table must be updated to reflect the process that is currently running.)

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.***



- 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.

Fig - Paging hardware with TLB

- The TLB is very expensive, however, and therefore very small. (Not large enough to hold the entire page table.) 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***.

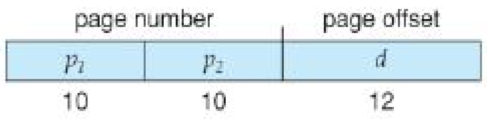
Protection

- 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 8.12 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.

<u>**Structure of the Page Table**</u>

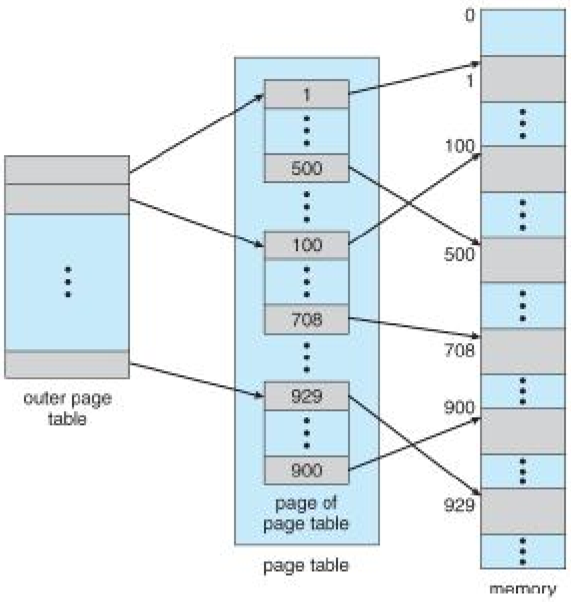
1. Hierarchical Paging

- 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 amount 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.)

Fig- A two-level page-table scheme



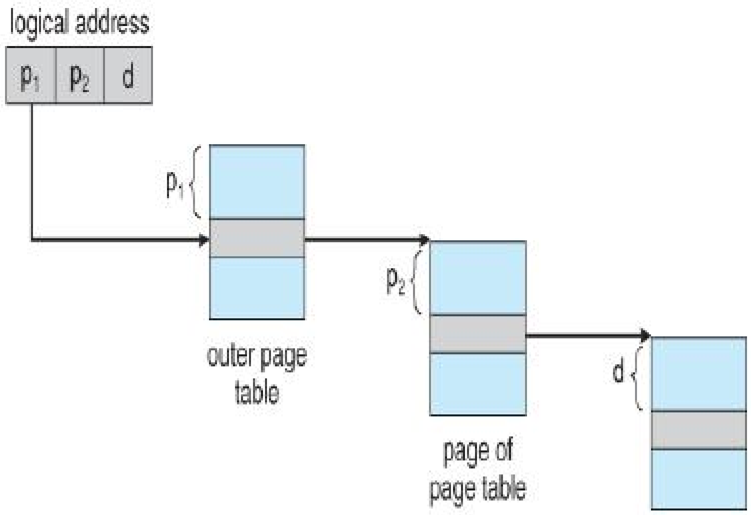


Fig - Address translation for a two-level 32-bit paging architecture

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

section	page	offset
s	*p*	*d*
2	21	9

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.

outer page	inner page	offset
p_1	p_2	*d*
42	10	12

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

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	*d*
32	10	10	12

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

2. Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***. Fig. below illustrates a ***hashed page table*** using chain-and-bucket hashing:

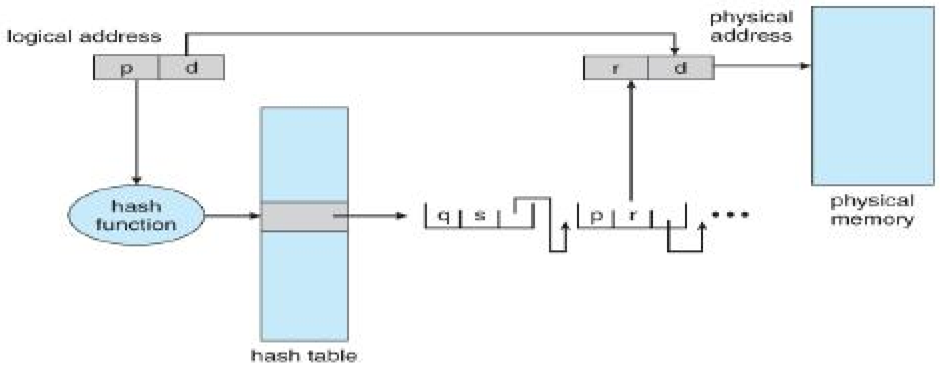
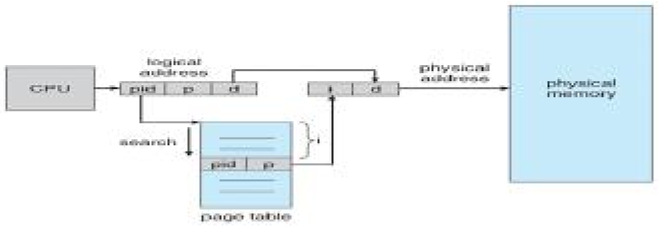


Fig - Hashed page table

3. Inverted Page Tables

- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (i.e. there is one entry per ***frame*** instead of one entry per ***page***.)



Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page (or to discover that it is not there.) Hashing the table can help speed up the search process.

Fig. - Inverted page table

- Inverted page tables prohibit the normal method of implementing shared memory, which is to map multiple logical pages to a common physical frame. (Because each frame is now mapped to one and only one process.)

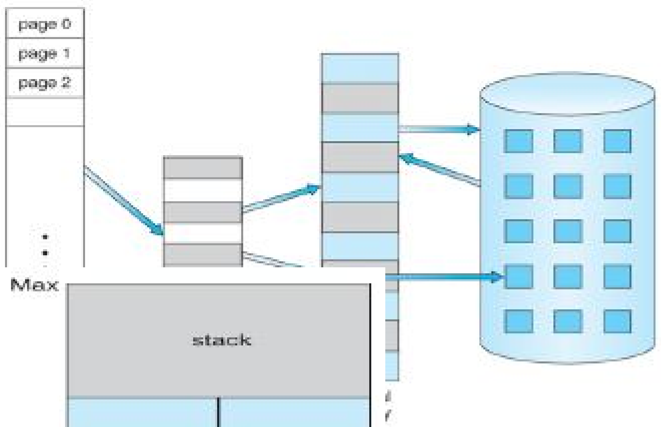
Virtual Memory

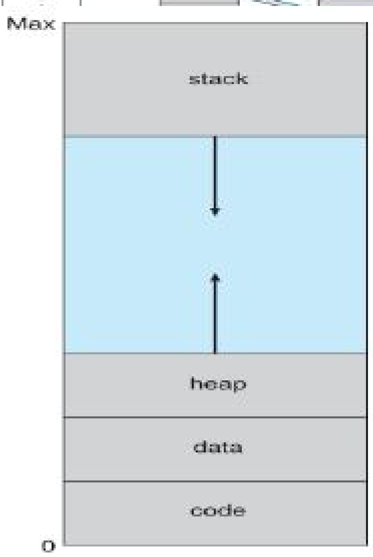
Background

- 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.

Fig - Diagram showing virtual memory that is larger than physical memory Fig. shows





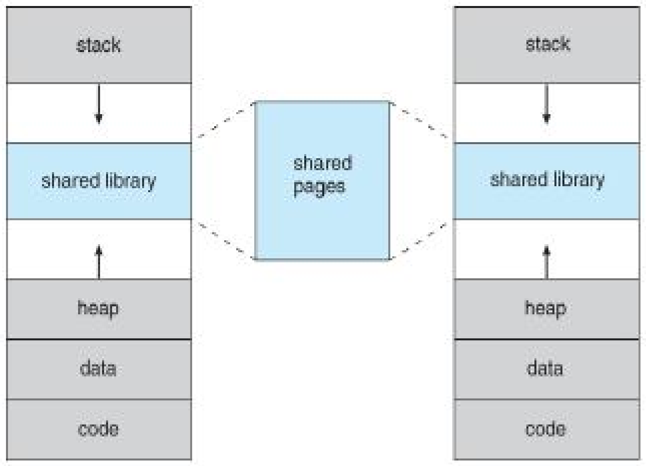
virtual address space, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.

- Note that the address space shown in Fig. 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.

Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits: The System libraries can be shared by mapping them into the virtual address space of more than one process.

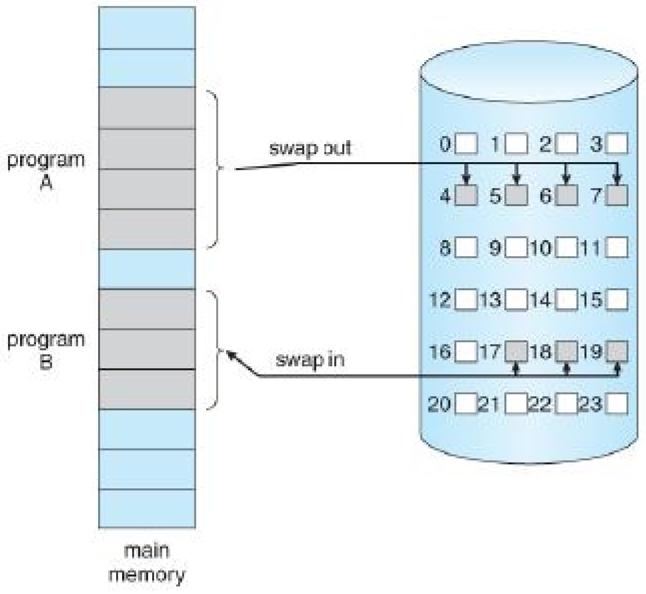
- o
- o
- o
- o
- o
- o
- o
- o

Fig- Virtual address space



- 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.

Fig - Shared library using virtual memory

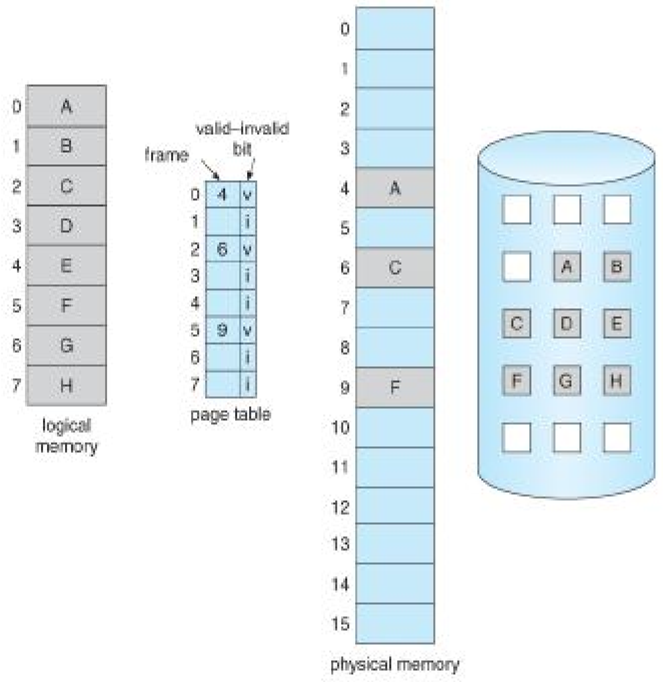


Demand Paging:

- 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.

Fig - 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. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)

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

- 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.

- 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.)

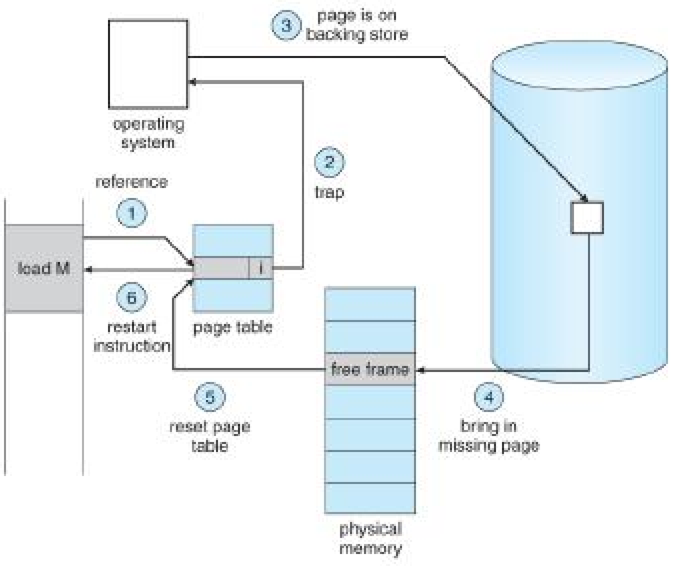


Fig - Steps in handling a page fault

- 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. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, (which may span a page boundary), and if some of the data gets modified before the page fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

Performance of Demand Paging

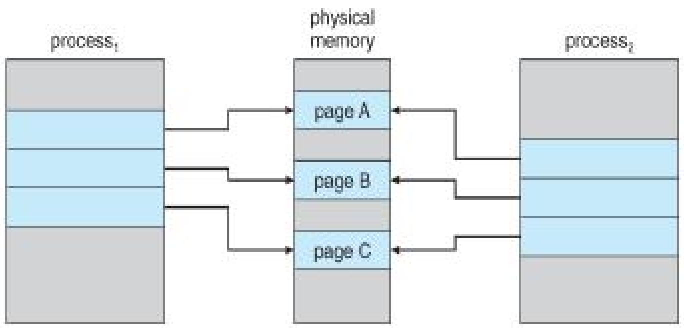
- Obviously there is some slowdown and performance hit whenever a page fault occurs and the system has to go get it from memory, but just how big a hit is it exactly?
- There are many steps that occur when servicing a page fault (see book for full details), and some of the steps are optional or variable. 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.

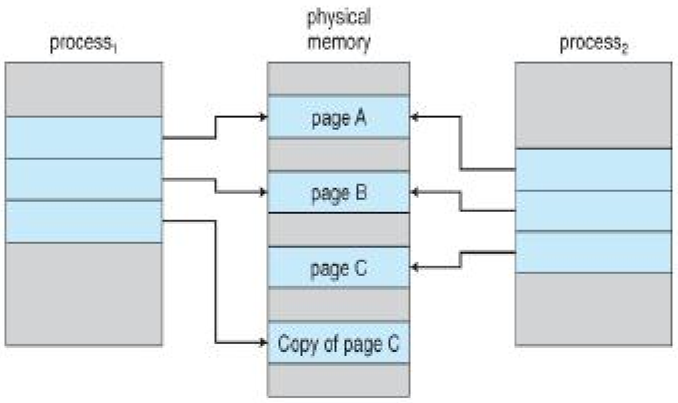
- A subtlety is that swap space is faster to access than the regular file system, because it does not have to go through the whole directory structure. For this reason some systems will transfer an entire process from the file system to swap space before starting up the process, so that future paging all occurs from the (relatively) faster swap space.
- Some systems use demand paging directly from the file system for binary code (which never changes and hence does not have to be stored on a page operation), and to reserve the swap space for data segments that must be stored. This approach is used by both Solaris and BSD Unix.

Copy-on-Write:



- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They can be simply shared between the two processes in the meantime, with a bit set that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec( ) system call immediately after the fork.

Obviously only pages that can be modified even need to be labeled as copy-on-write. Code segments can simply be shared.

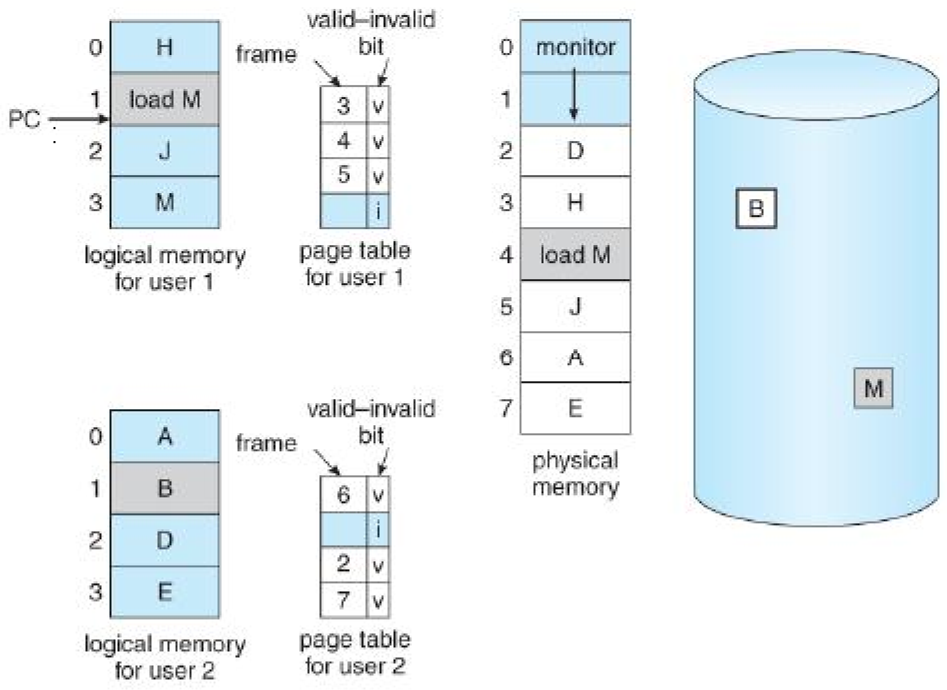


- Pages used to satisfy copy-on-write duplications are typically allocated using ***zero-fill-on-demand***, meaning that their previous contents are zeroed out before the copy proceeds.
- Some systems provide an alternative to the fork( ) system call called a ***virtual memory fork, vfork( )***. In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the exec( ) system call. (In essence this addresses the question of which process executes first after a call to fork, the parent or the child. With vfork, the parent is suspended, allowing the child to execute first until it calls exec( ), sharing pages with the parent in the meantime.

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. (Some allocate a fixed amount for I/O, and others let the I/O system contend for memory along with everything else.)
    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.

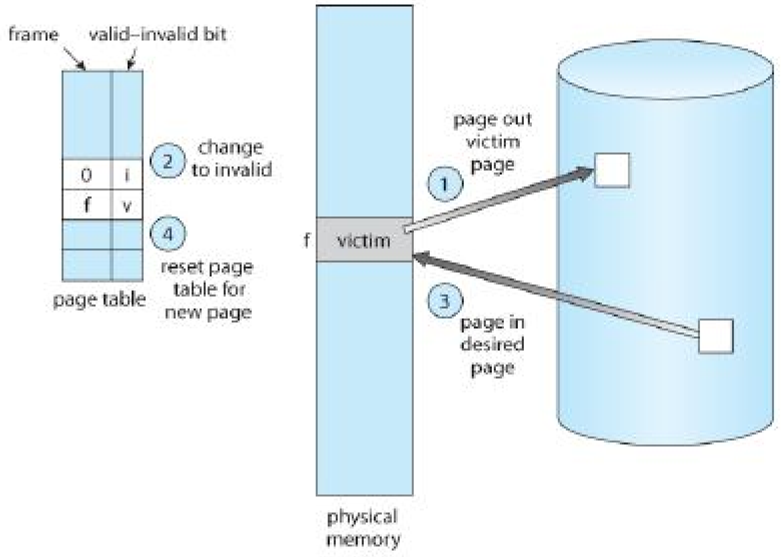


Basic Page Replacement

- 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.



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. It should come as no surprise that many page replacement strategies specifically look for pages that do not have their dirty bit set, and preferentially select clean pages as victim pages. It should also be obvious that unmodifiable code pages never get their dirty bits set.

- 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:

i. Only the page number that was accessed is relevant. The offset within that page does not affect paging operations.
ii. 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 Fig.

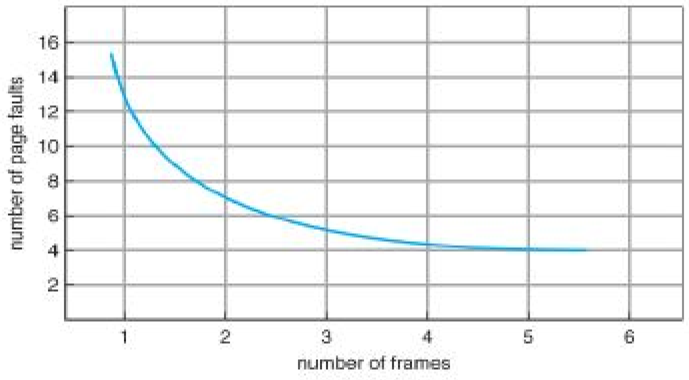


Fig - Graph of page faults versus number of frames.

2. 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:

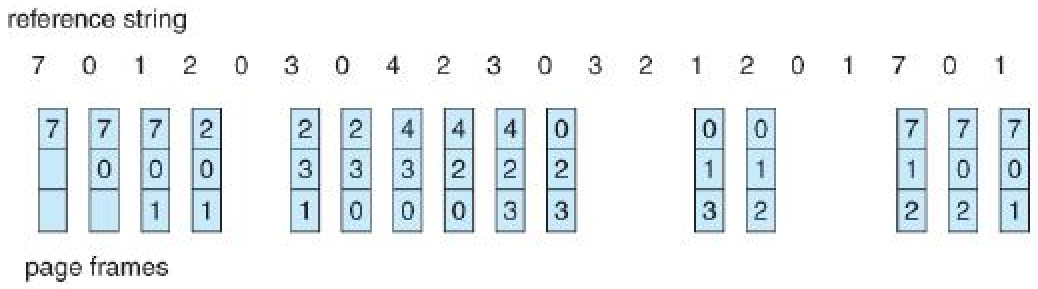


Fig - FIFO page-replacement algorithm.

- 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:

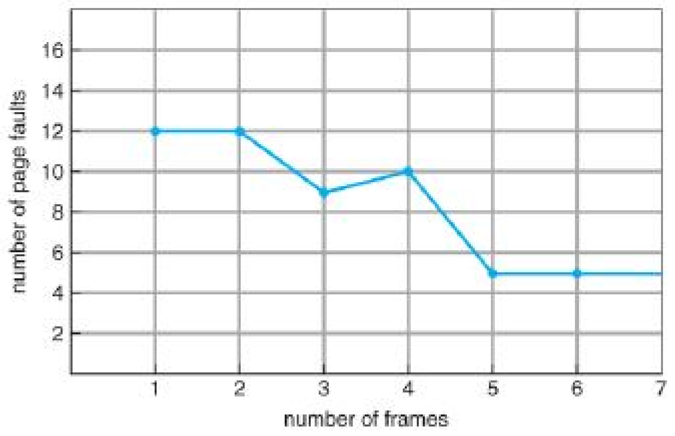


Fig - Page-fault curve for FIFO replacement on a reference string

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, Figure 9.14 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. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.)
- 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.
- In practice most page-replacement algorithms try to approximate OPT by predicting (estimating) in one fashion or another what page will not be used for the longest period of time. The basis of FIFO is the prediction that the page that

was brought in the longest time ago is the one that will not be needed again for the longest future time, but as we shall see, there are many other prediction methods, all striving to match the performance of OPT.

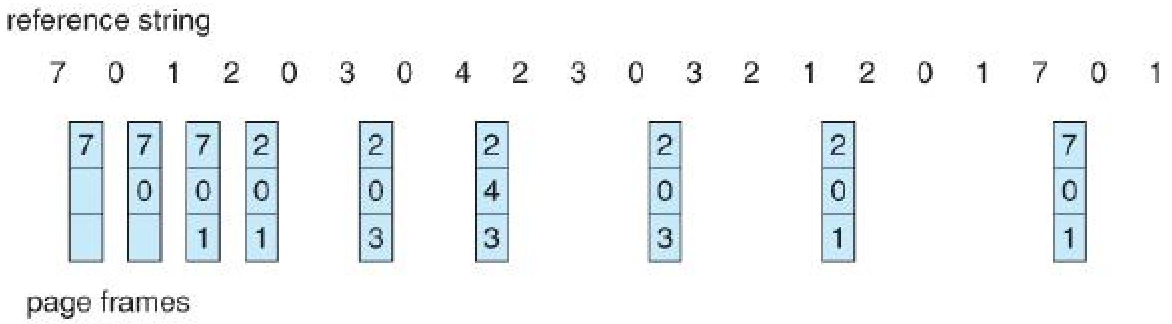


Fig - Optimal page-replacement algorithm

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 9.15 illustrates LRU for our sample string, yielding 12 page faults, (as compared to 15 for FIFO and 9 for OPT.)

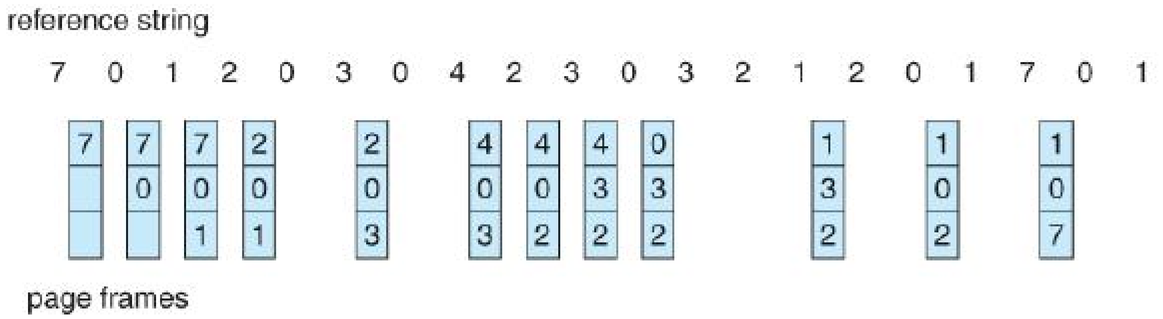


Fig - LRU page-replacement algorithm.

- 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.

- Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for ***every*** memory access.
- 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. A stack algorithm is one in which the pages kept in memory for a frame set of size N will always be a subset of the pages kept for a frame size of N + 1. In the case of LRU, (and particularly the stack implementation thereof), the top N pages of the stack will be the same for all frame set sizes of N or anything larger.

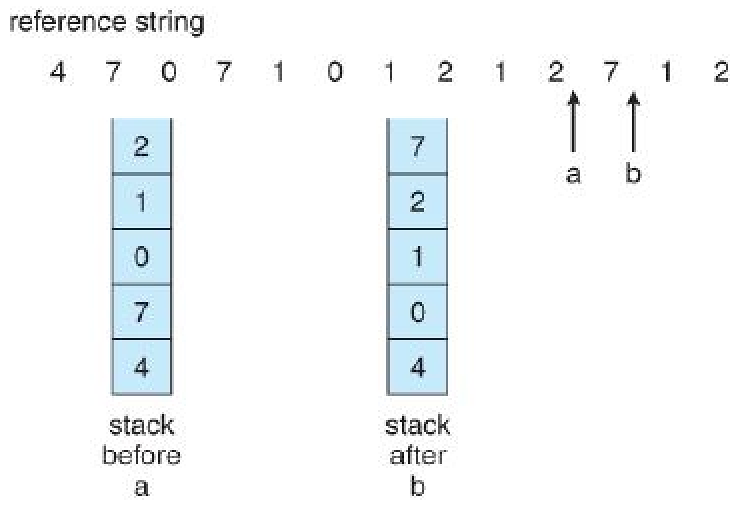


Fig - Use of a stack to record the most recent page references.

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 the absence of ANY hardware support, FIFO might be the best available choice.)

- 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.

1 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.

2 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.

- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the ***clock*** algorithm, from the hands of the clock moving around the circular queue.

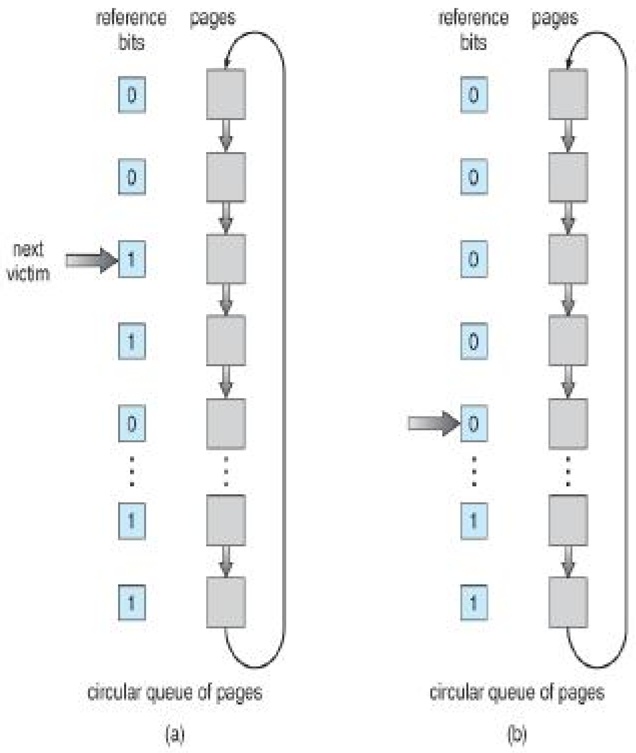


Fig - Second-chance (clock) page-replacement algorithm.

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. For this reason architectures place a limit (say 16) on the number of levels of indirection allowed in an instruction, which is enforced with a counter initialized to the limit and decremented with every level of indirection in an instruction - If the counter reaches zero, then an "excessive indirection" trap occurs. This example would still require a minimum frame allocation of 17 per process.

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 P_i 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

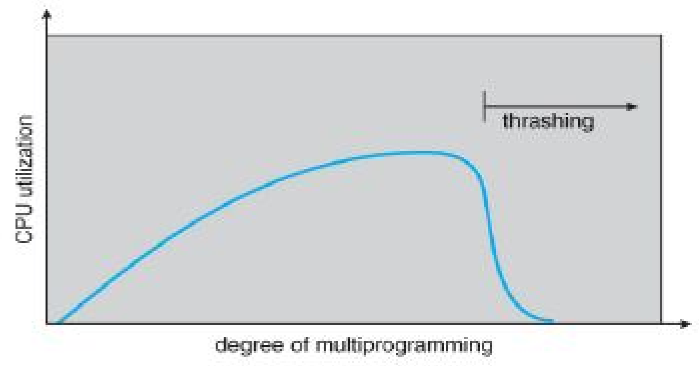
- The above arguments all assume that all memory is equivalent, or at least has equivalent access times.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- 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. For example, all processors and RAM on a single board would probably be in the same lgroup. Memory assignments are made within the same lgroup if possible, or to the next nearest lgroup otherwise. (Where "nearest" is defined as having the lowest access time.)

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

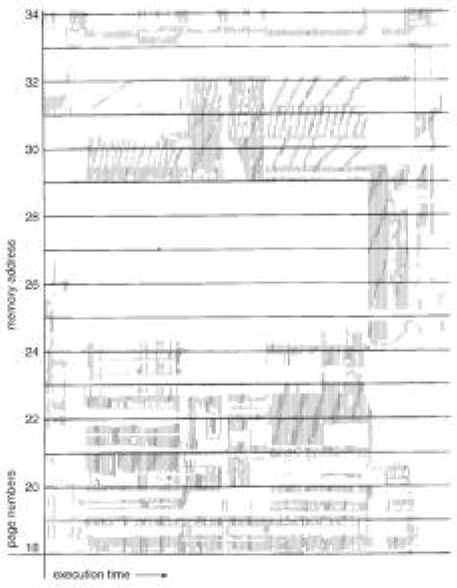
- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU 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.)

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.)

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 9.20:

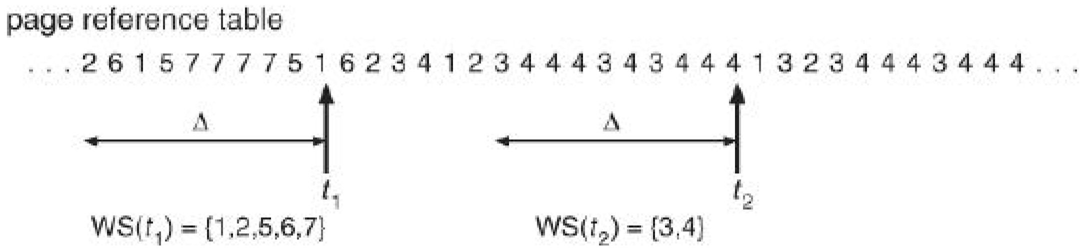


Fig - Working-set model.

- 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:
    - For example, suppose that we set the timer to go off after every 5000 references (by any process), and we can store two additional historical reference bits in addition to the current reference bit.
    - Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared.
    - If any of the three bits is set, then that page was referenced within the last 15,000 references, and is considered to be in that processes reference set.
    - Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead.

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.
- (I suppose a page-replacement strategy could be devised that would select victim frames based on the process with the lowest current page-fault frequency.)

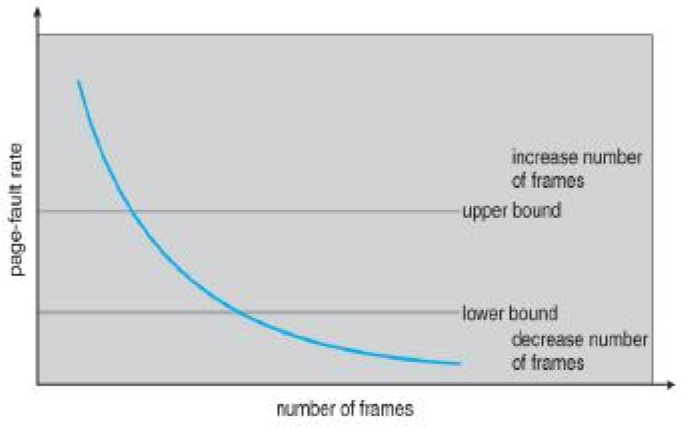
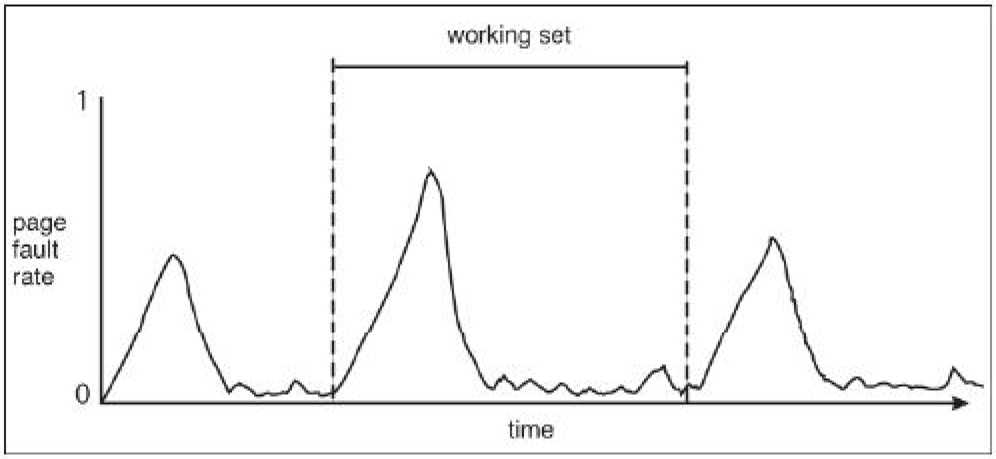


Fig - Page-fault frequency.

- Note that there is a direct relationship between the page-fault rate and the working-set, as a process moves from one locality to another:

Unnumbered side bar in Ninth Edition

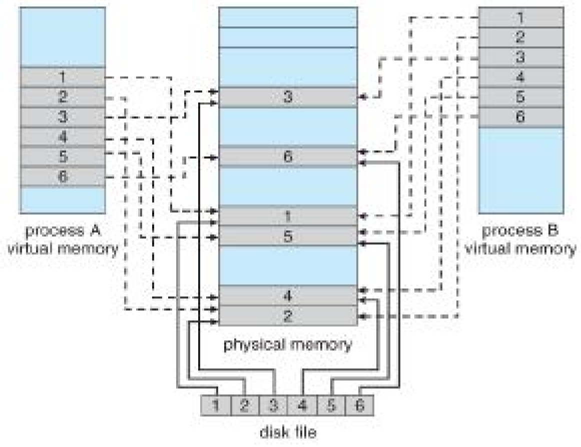


Memory-Mapped Files

- Rather than accessing data files directly via the file system with every file access, data files can be paged into memory the same as process files, resulting in much faster accesses (except of course when page-faults occur.) This is known as ***memory-mapping*** a file.

Basic Mechanism

- Basically a file is mapped to an address range within a process's virtual address space, and then paged in as needed using the ordinary demand paging system.
- Note that file writes are made to the memory page frames, and are not immediately written out to disk. (This is the purpose of the "flush( )" system call, which may also be needed for stdout in some cases. See the timekiller program for an example of this.)



- This is also why it is important to "close( )" a file when one is done writing to it - So that the data can be safely flushed out to disk and so that the memory frames can be freed up for other purposes.
- Some systems provide special system calls to memory map files

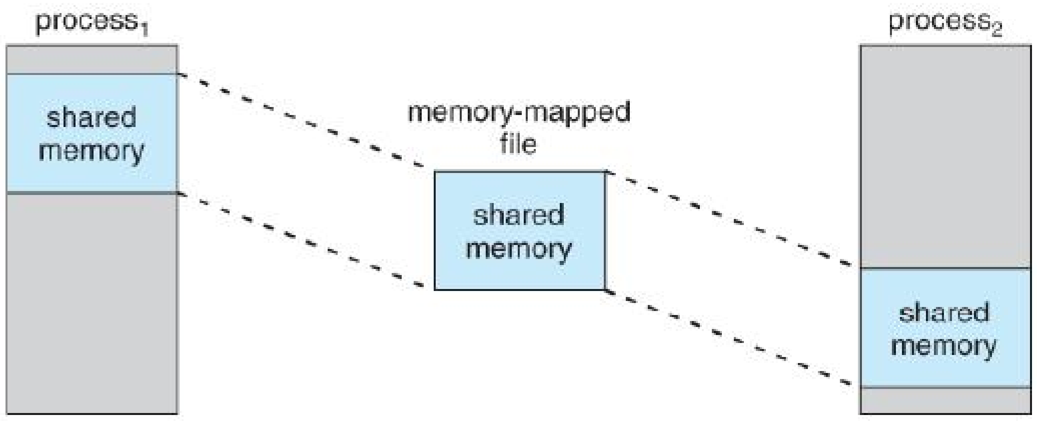
and use direct disk access otherwise. Other systems map the file to process address space if the special system calls are used and map the file to kernel address space otherwise, but do memory mapping in either case.

- File sharing is made possible by mapping the same file to the address space of more than one process, as shown in Figure below. Copy-on-write is supported, and mutual exclusion techniques (chapter 6) may be needed to avoid synchronization problems.

- Shared memory can be implemented via shared memory-mapped files (Windows), or it can be implemented through a separate process (Linux, UNIX.)

Shared Memory in the Win32 API

- Windows implements shared memory using shared memory-mapped files,



involving three basic steps:

1. Create a file, producing a HANDLE to the new file.
2. Name the file as a shared object, producing a HANDLE to the shared object.
3. Map the shared object to virtual memory address space, returning its base address as a void pointer (LPVOID).

Memory-Mapped I/O

- All access to devices is done by writing into (or reading from) the device's registers. Normally this is done via special I/O instructions.
- For certain devices it makes sense to simply map the device's registers to addresses in the process's virtual address space, making device I/O as fast and simple as any other memory access. Video controller cards are a classic example of this.
- erial and parallel devices can also use memory mapped I/O, mapping the device registers to specific memory addresses known as ***I/O Ports***, e.g. 0xF8. Transferring a

series of bytes must be done one at a time, moving only as fast as the I/O device is prepared to process the data, through one of two mechanisms:

- ***Programmed I/O (PIO)***, also known as ***polling.*** The CPU periodically checks the control bit on the device, to see if it is ready to handle another byte of data.
- ***Interrupt Driven.*** The device generates an interrupt when it either has another byte of data to deliver or is ready to receive another byte.

Allocating Kernel Memory

- Previous discussions have centered on process memory, which can be conveniently broken up into page-sized chunks, and the only fragmentation that occurs is the average half-page lost to internal fragmentation for each process (segment.)
- There is also additional memory allocated to the kernel, however, which cannot be so easily paged. Some of it is used for I/O buffering and direct access by devices, example, and must therefore be contiguous and not affected by paging. Other memory is used for internal kernel data structures of various sizes, and since kernel memory is often locked (restricted from being ever swapped out), management of this resource must be done carefully to avoid internal fragmentation or other waste. (I.e. you would like the kernel to consume as little memory as possible, leaving as much as possible for user processes.) Accordingly there are several classic algorithms in place for allocating kernel memory structures.

Buddy System

- The Buddy ***System*** allocates memory using a ***power of two allocator***.
- Under this scheme, memory is always allocated as a power of 2 (4K, 8K, 16K, etc), rounding up to the next nearest power of two if necessary.
- If a block of the correct size is not currently available, then one is formed by splitting the next larger block in two, forming two matched buddies. (And if that larger size is not available, then the next largest available size is split, and so on.)
- One nice feature of the buddy system is that if the address of a block is exclusively ORed with the size of the block, the resulting address is the address of the buddy of the same size, which allows for fast and easy ***coalescing*** of free blocks back into larger blocks.
  - Free lists are maintained for every size block.
  - If the necessary block size is not available upon request, a free block from the next largest size is split into two buddies of the desired size. (Recursively splitting larger size blocks if necessary.)
  - When a block is freed, its buddy's address is calculated, and the free list for that size block is checked to see if the buddy is also free. If it is, then

the two buddies are coalesced into one larger free block, and the process is repeated with successively larger free lists.

- See the (annotated) Figure below for an example.

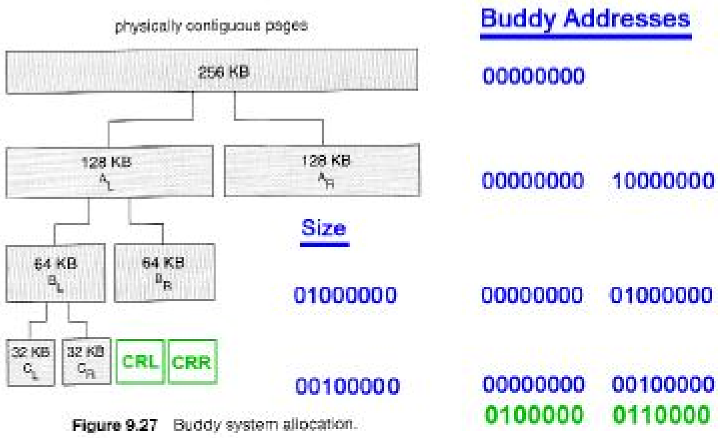
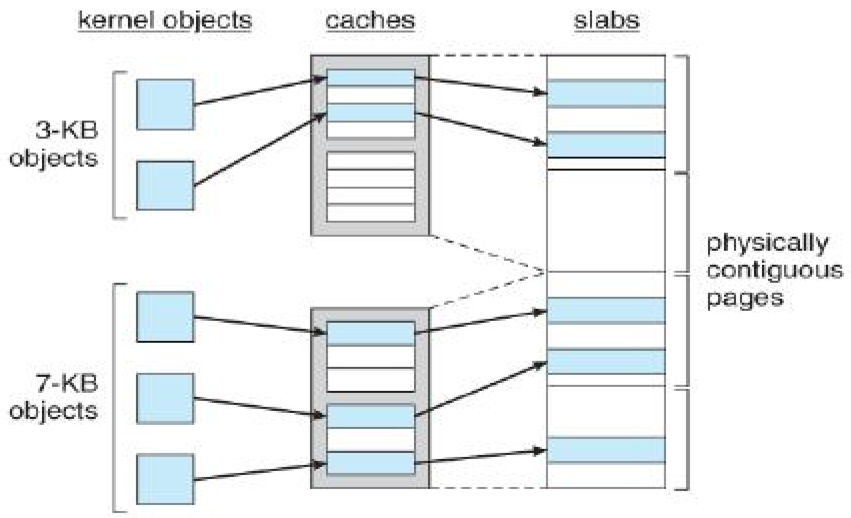


Figure 9.27 Buddy system allocation.

Slab Allocation

- ***Slab Allocation*** allocates memory to the kernel in chunks called ***slabs***, consisting of one or more contiguous pages. The kernel then creates separate caches for each type of data structure it might need from one or more slabs. Initially the caches are marked empty, and are marked full as they are used.
- New requests for space in the cache is first granted from empty or partially empty slabs, and if all slabs are full, then additional slabs are allocated.
- This essentially amounts to allocating space for arrays of structures, in large chunks suitable to the size of the structure being stored. For example if a particular structure were 512 bytes long, space for them would be allocated in groups of 8 using 4K pages. If the structure were 3K, then space for 4 of them could be allocated at one time in a slab of 12K using three 4K pages.
- Benefits of slab allocation include lack of internal fragmentation and fast allocation of space for individual structures.

- Solaris uses slab allocation for the kernel and also for certain user-mode memory



  allocations. Linux used the buddy system prior to 2.2 and switched to slab allocation since then. SLOB, Simple List of Blocks, maintains 3 linked lists of free blocks - small, medium, and large - designed for (imbedded) systems with limited amounts of memory.
    - SLUB modifies some implementation issues for better performance on systems with large numbers of processors.

Operating-System Examples (Optional)

Windows

- Windows uses demand paging with ***clustering,*** meaning they page in multiple pages whenever a page fault occurs.
- The working set minimum and maximum are normally set at 50 and 345 pages respectively. (Maximums can be exceeded in rare circumstances.)
- Free pages are maintained on a free list, with a minimum threshold indicating when there are enough free frames available.
- If a page fault occurs and the process is below their maximum, then additional pages are allocated. Otherwise some pages from this process must be replaced, using a local page replacement algorithm.
- If the amount of free frames falls below the allowable threshold, then ***working set trimming*** occurs, taking frames away from any processes which are above their minimum, until all are at their minimums. Then additional frames can be allocated to processes that need them.
- The algorithm for selecting victim frames depends on the type of processor:
    - On single processor 80x86 systems, a variation of the clock (second chance) algorithm is used.

- On Alpha and multiprocessor systems, clearing the reference bits may require invalidating entries in the TLB on other processors, which is an expensive operation. In this case Windows uses a variation of FIFO.

Solaris

- Solaris maintains a list of free pages, and allocates one to a faulting thread whenever a fault occurs. It is therefore imperative that a minimum amount of free memory be kept on hand at all times.
- Solaris has a parameter, *lotsfree,* usually set at 1/64 of total physical memory. Solaris checks 4 times per second to see if the free memory falls below this threshhold, and if it does, then the ***pageout*** process is started.
- Pageout uses a variation of the clock (second chance) algorithm, with two hands rotating around through the frame table. The first hand clears the reference bits, and the second hand comes by afterwards and checks them. Any frame whose reference bit has not been reset before the second hand gets there gets paged out.
- The Pageout method is adjustable by the distance between the two hands, (the ***handspan***), and the speed at which the hands move. For example, if the hands each check 100 frames per second, and the handspan is 1000 frames, then there would be a 10 second interval between the time when the leading hand clears the reference bits and the time when the trailing hand checks them.
- The speed of the hands is usually adjusted according to the amount of free memory, as shown below. ***Slowscan*** is usually set at 100 pages per second, and ***fastscan*** is usually set at the smaller of 1/2 of the total physical pages per second and 8192 pages per second.

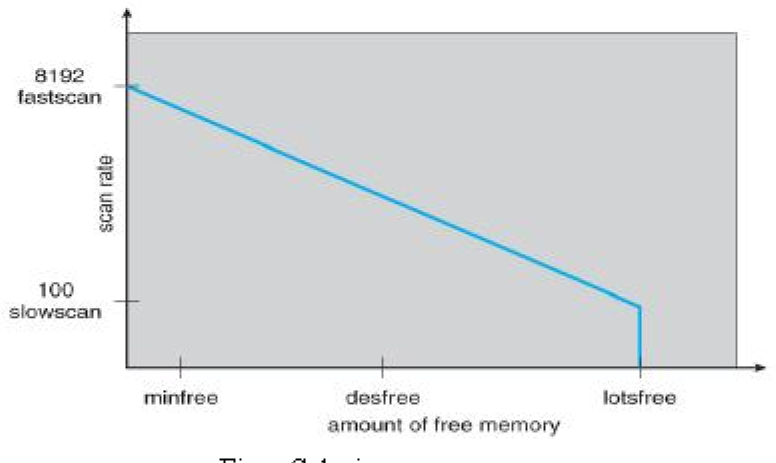


Fig - Solaris page scanner.

- Solaris also maintains a cache of pages that have been reclaimed but which have not yet been overwritten, as opposed to the free list which only holds pages whose current contents are invalid. If one of the pages from the cache is needed before it gets moved to the free list, then it can be quickly recovered.

- Normally pageout runs 4 times per second to check if memory has fallen below *lotsfree.* However if it falls below *desfree,* then pageout will run at 100 times per second in an attempt to keep at least desfree pages free. If it is unable to do this for a 30-second average, then Solaris begins swapping processes, starting preferably with processes that have been idle for a long time.
- If free memory falls below *minfree,* then pageout runs with every page fault.
- Recent releases of Solaris have enhanced the virtual memory management system, including recognizing pages from shared libraries, and protecting them from being paged out.

Unit IV - Storage Management

Overview of Mass-Storage Structure

1 Magnetic Disks

Traditional magnetic disks have the following basic structure:

One or more ***platters*** in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.

Each platter has two working ***surfaces.*** Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.

Each working surface is divided into a number of concentric rings called ***tracks.*** The collection of all tracks that are the same distance from the edge of the platter, (i.e. all tracks immediately above one another in the following diagram) is called a ***cylinder***.

Each track is further divided into ***sectors,*** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. (Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors.)

The data on a hard drive is read by read-write ***heads.*** The standard configuration (shown below) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another. (Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties.)

The storage capacity of a traditional disk drive is equal to the number of heads (i.e. the number of working surfaces), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

In operation the disk rotates at high speed, such as 7200 rpm (120 revolutions per second.) The rate at which data can be transferred from the disk to the computer is composed of several steps:

The ***positioning time***, a.k.a. the ***seek time*** or ***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.

The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head. This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. (For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.

The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer.

Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to ***park*** the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.

Floppy disks are normally ***removable***. Hard drives can also be removable, and some are even ***hot-swappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.

Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.

The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

2 Magnetic Tapes

Magnetic tapes were once used for common secondary storage before the days of hard disk drives, but today are used primarily for backups.

Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.

Capacities of tape drives can range from 20 to 200 GB, and compression can double that capacity.

Disk Structure

The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:

The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.

All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.

Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.

Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:

With ***Constant Linear Velocity, CLV,*** the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.

With ***Constant Angular Velocity, CAV,*** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

Disk Attachment

Disk drives can be attached either directly to a particular host (a local disk) or to a network.

1 Host-Attached Storage

Local disks are accessed through I/O Ports as described earlier.

The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.

SATA is similar with simpler cabling.

High end workstations or other systems in need of larger number of disks typically use SCSI disks:

The SCSI standard supports up to 16 ***targets*** on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.

A SCSI target is usually a single drive, but the standard also supports up to 8 ***units*** within each target. These would generally be used for accessing individual disks within a RAID array.

The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.

Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.

SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.

FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:

A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the ***storage-area networks, SANs,*** to be discussed in a future section.

The ***arbitrated loop, FC-AL,*** that can address up to 126 devices (drives and controllers.)

2 Network-Attached Storage

Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.

NAS can be implemented using SCSI cabling, or ***ISCSI*** uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.

NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

3 Storage-Area Network

A ***Storage-Area Network, SAN,*** connects computers and storage devices in a network, using storage protocols instead of network protocols.

One advantage of this is that storage access does not tie up regular networking bandwidth.

SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.

SAN is also controllable, allowing restricted access to certain hosts and devices.

Disk Scheduling

As mentioned earlier, disk transfer speeds are limited primarily by ***seek times*** and ***rotational latency.*** When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.

BANDWIDTH is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, (for a series of disk requests.)

Both bandwidth and access time can be improved by processing requests in a good order.

Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

1 FCFS Scheduling

First-Come First-Serve is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:

2 SSTF Scheduling

Shortest Seek Time First scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.

SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

3 SCAN Scheduling

The ***SCAN*** algorithm, a.k.a. the ***elevator*** algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.

Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

4 C-SCAN Scheduling

The ***Circular-SCAN*** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

5, LOOK Scheduling

LOOK scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:

Selection of a Disk-Scheduling Algorithm

With very low loads all algorithms are equal, since there will normally only be one request to process at a time.

For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.

For busier systems, SCAN and LOOK algorithms eliminate starvation problems.

The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.

Some improvement to overall file system access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.

On modern disks the rotational latency can be almost as significant as the seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, (particularly when bad blocks have been remapped to spare sectors.)

Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, (which do know the actual geometry of the disk as well as any remapping), so that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.

Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason OSes may elect to spoon-feed requests to the disk controller one at a time in certain situations.

Disk Management

1. Disk Formatting

Before a disk can be used, it has to be ***low-level formatted***, which means laying down all of the headers and trailers demarking the beginning and ends of each sector. Included in the header and trailer are the linear sector numbers, and ***error-correcting codes, ECC,*** which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.

ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a ***soft error*** has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS.

Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.

After partitioning, then the filesystems must be ***logically formatted,*** which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the filesystem. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the filesystem structure, but requires that the application program manage its own disk storage requirements.)

2 Boot Block

Computer ROM contains a ***bootstrap*** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)

The first sector on the hard drive is known as the ***Master Boot Record, MBR,*** and contains a very small amount of code in addition to the ***partition table.*** The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the ***active*** or ***boot*** partition.

The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.

In a ***dual-boot*** (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.

Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include ***single-user*** a.k.a. ***maintenance*** or ***safe*** modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.

3 Bad Blocks

No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.

In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)

Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (Recall that blocks are tested with every write as well as with every read, so often errors can be detected before the write operation is complete, and the data simply written to a different sector instead.)

Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimization of the OS, especially if the replacement sector is physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder. Whenever possible a bad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. ***Sector slipping*** may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.

If the data on a bad block cannot be recovered, then a ***hard error*** has occurred., which requires replacing the file(s) from backups, or rebuilding them from scratch.

Swap-Space Management

Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.

Managing swap space is obviously an important task for modern OSes.

1 Swap-Space Use

The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!

Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

2 Swap-Space Location

Swap space can be physically located in one of two locations:

As a large file which is part of the regular filesystem. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.

As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

3 Swap-Space Management: An Example

Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back in from there than to write them out to swap space and then read them back.)

In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)

RAID Structure

The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)

RAID originally stood for ***Redundant Array of Inexpensive Disks,*** and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to ***Independent*** disks.

1 Improvement of Reliability via Redundancy

The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually ***decreases*** the **Mean Time To Failure, MTTF** of the system.

If, however, the same data was copied onto multiple disks, then the data would not be lost unless **both** (or all) copies of the data were damaged simultaneously, which is a **MUCH** lower probability than for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the **Mean Time To Repair** into play. For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the **Mean Time to Data Loss** would be 500 * 10^6 hours, or 57,000 years!

This is the basic idea behind disk ***mirroring***, in which a system contains identical data on two or more disks.

Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

2 Improvement in Performance via Parallelism

There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.)

Another way of improving disk access time is with ***striping***, which basically means spreading data out across multiple disks that can be accessed simultaneously.

With ***bit-level striping*** the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.

BLOCK-LEVEL STRIPING spreads a filesystem across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when filesystems are accessed in ***clusters*** of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

3 RAID Levels

Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different ***RAID levels***, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)

RAID LEVEL 0 - This level includes striping only, with no mirroring.

RAID LEVEL 1 - This level includes mirroring only, no striping.

RAID LEVEL 2 - This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)

RAID LEVEL 3 - This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

RAID LEVEL 4 - This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.

RAID LEVEL 5 - This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

RAID LEVEL 6 - This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the ***Reed-Solomon codes***), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.

RAID level 0 + 1 disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.

RAID level 1 + 0 mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.

In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.

If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.

However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.

In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.

If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.

Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.

In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

4 Selecting a RAID Level

Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.

Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost, but increases the likelihood that a second disk will fail before the first bad disk is repaired.

5 Extensions

RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data.

ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.

ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.

Stable-Storage Implementation

The concept of stable storage (first presented in chapter 6) involves a storage medium in which data is ***never*** lost, even in the face of equipment failure in the middle of a write operation.

To implement this requires two (or more) copies of the data, with separate failure modes.

An attempted disk write results in one of three possible outcomes:

The data is successfully and completely written.

The data is partially written, but not completely. The last block written may be garbled.

No writing takes place at all.

Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:

Write the data to the first physical block.

After step 1 had completed, then write the data to the second physical block.

Declare the operation complete only after both physical writes have completed successfully.

During recovery the pair of blocks is examined.

If both blocks are identical and there is no sign of damage, then no further action is necessary.

If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. (This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged.)

If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. (Undo the operation.)

Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.

Tertiary-Storage Structure

Primary storage refers to computer memory chips; Secondary storage refers to fixed-disk storage systems (hard drives); And ***Tertiary Storage*** refers to ***removable media,*** such as tape drives, CDs, DVDs, and to a lesser extend floppies, thumb drives, and other detachable devices.

Tertiary storage is typically characterized by large capacity, low cost per MB, and slow access times, although there are exceptions in any of these categories.

Tertiary storage is typically used for backups and for long-term archival storage of completed work. Another common use for tertiary storage is to swap large little-used files (or groups of files) off of the hard drive, and then swap them back in as needed in a fashion similar to secondary storage providing swap space for primary storage.

Tertiary-Storage Devices

1 Removable Disks

Removable magnetic disks (e.g. floppies) can be nearly as fast as hard drives, but are at greater risk for damage due to scratches. Variations of removable magnetic disks up to a GB or more in capacity have been developed. (Hot-swappable hard drives?)

A ***magneto-optical*** disk uses a magnetic disk covered in a clear plastic coating that protects the surface.

The heads sit a considerable distance away from the magnetic surface, and as a result do not have enough magnetic strength to switch bits ***at normal room temperature.***

For writing, a laser is used to heat up a specific spot on the disk, to a temperature at which the weak magnetic field of the write head is able to flip the bits.

For reading, a laser is shined at the disk, and the ***Kerr effect*** causes the polarization of the light to become rotated either clockwise or counter-clockwise depending on the orientation of the magnetic field.

Optical disks do not use magnetism at all, but instead use special materials that can be altered (by lasers) to have relatively light or dark spots.

For example the ***phase-change disk*** has a material that can be frozen into either a crystalline or an amorphous state, the latter of which is less transparent and reflects less light when a laser is bounced off a reflective surface under the material.

Three powers of lasers are used with phase-change disks: (1) a low power laser is used to read the disk, without effecting the materials. (2) A medium power erases the disk, by melting and re-freezing the medium into a crystalline state, and (3) a high power writes to the disk by melting the medium and re-freezing it into the amorphous state.

The most common examples of these disks are ***re-writable*** CD-RWs and DVD-RWs.

An alternative to the disks described above are ***Write-Once Read-Many, WORM*** drives.

The original version of WORM drives involved a thin layer of aluminum sandwiched between two protective layers of glass or plastic.

Holes were burned in the aluminum to write bits.

Because the holes could not be filled back in, there was no way to re-write to the disk. (Although data could be erased by burning more holes.)

WORM drives have important legal ramifications for data that must be stored for a very long time and must be provable in court as unaltered since it was originally written. (Such as long-term storage of medical records.)

Modern CD-R and DVD-R disks are examples of WORM drives that use organic polymer inks instead of an aluminum layer.

Read-only disks are similar to WORM disks, except the bits are pressed onto the disk at the factory, rather than being burned on one by one.

2 Tapes

Tape drives typically cost more than disk drives, but the cost per MB of the tapes themselves is lower.

Tapes are typically used today for backups, and for enormous volumes of data stored by certain scientific establishments. (E.g. NASA's archive of space probe and satellite imagery, which is currently being downloaded from numerous sources faster than anyone can actually look at it.)

Robotic tape changers move tapes from drives to archival tape libraries upon demand.

3 Future Technology

Solid State Disks, SSDs, are becoming more and more popular.

Holographic storage uses laser light to store images in a 3-D structure, and the entire data structure can be transferred in a single flash of laser light.

Micro-Electronic Mechanical Systems, MEMS, employs the technology used for computer chip fabrication to create VERY tiny little machines. One example packs 10,000 read-write heads within a square centimeter of space, and as media are passed over it, all 10,000 heads can read data in parallel.

I/O Hardware and Application of I/O Interface

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

Block devices − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.

Character devices − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

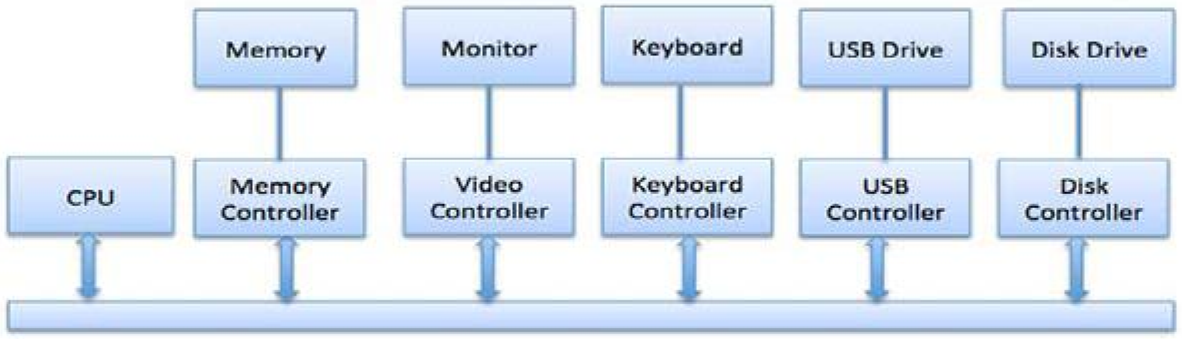
Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



Synchronous vs asynchronous I/O

Synchronous I/O – In this scheme CPU execution waits while I/O proceeds

Asynchronous I/O – I/O proceeds concurrently with CPU execution

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

Special Instruction I/O

Memory-mapped I/O

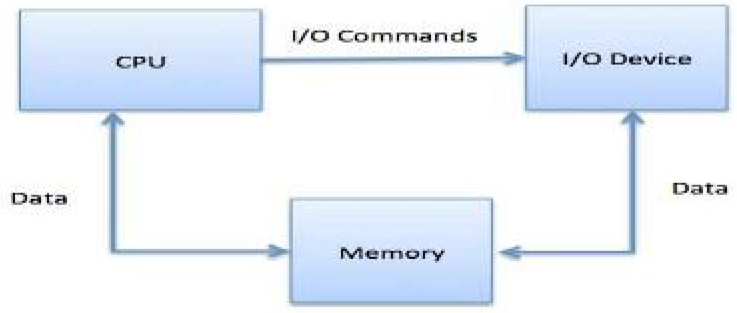
Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

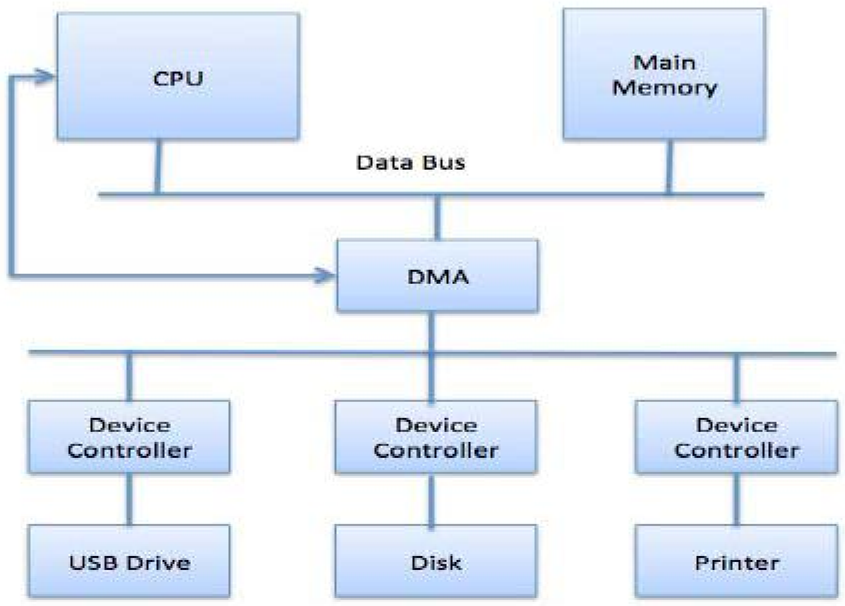
The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

Direct Memory Access (DMA)

Slow devices like keyboards will generate an interrupt to the main CPU after each byte is transferred. If a fast device such as a disk generated an interrupt for each byte, the operating system would spend most of its time handling these interrupts. So a typical computer uses direct memory access (DMA) hardware to reduce this overhead.

Direct Memory Access (DMA) means CPU grants I/O module authority to read from or write to memory without involvement. DMA module itself controls exchange of data between main memory and the I/O device. CPU is only involved at the beginning and end of the transfer and interrupted only after entire block has been transferred.

Direct Memory Access needs a special hardware called DMA controller (DMAC) that manages the data transfers and arbitrates access to the system bus. The controllers are programmed with source and destination pointers (where to read/write the data), counters to track the number of transferred bytes, and settings, which includes I/O and memory types, interrupts and states for the CPU cycles.



The operating system uses the DMA hardware as follows –

Step	Description
1	Device driver is instructed to transfer disk data to a buffer address X.
2	Device driver then instruct disk controller to transfer data to buffer.
3	Disk controller starts DMA transfer.
4	Disk controller sends each byte to DMA controller.
5	DMA controller transfers bytes to buffer, increases the memory address, decreases the counter C until C becomes zero.
6	When C becomes zero, DMA interrupts CPU to signal transfer completion.

Polling vs Interrupts I/O

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as **polling** and **interrupts**. Both of these techniques allow the processor to deal with events that can happen at any time and that are not related to the process it is currently running.

Polling I/O

Polling is the simplest way for an I/O device to communicate with the processor. The process of periodically checking status of the device to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

Most of the time, devices will not require attention and when one does it will have to wait until it is next interrogated by the polling program. This is an inefficient method and much of the processors time is wasted on unnecessary polls.

Compare this method to a teacher continually asking every student in a class, one after another, if they need help. Obviously the more efficient method would be for a student to inform the teacher whenever they require assistance.

Interrupts I/O

An alternative scheme for dealing with I/O is the interrupt-driven method. An interrupt is a signal to the microprocessor from a device that requires attention.

A device controller puts an interrupt signal on the bus when it needs CPU's attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

Kernel I/O Subsystem

Prerequisite – Microkernel
The kernel provides many services related to I/O. Several services such as scheduling, caching, spooling, device reservation, and error handling – are provided by the kernel, s I/O subsystem built on the hardware and device-driver infrastructure. The I/O subsystem is also responsible for protecting itself from the errant processes and malicious users.

I/O Scheduling –
To schedule a set of I/O request means to determine a good order in which to execute them. The order in which application issues the system call are the best choice. Scheduling can improve the overall performance of the system, can share device access permission fairly to all the processes, reduce the average waiting time, response time, turnaround time for I/O to complete.

OS developers implement scheduling by maintaining a wait queue of the request for each device. When an application issue a blocking I/O system call, The request is placed in the queue for that device. The I/O scheduler rearrange the order to improve the efficiency of the system.

Buffering –
A *buffer* is a memory area that stores data being transferred between two devices or between a device and an application. Buffering is done for three reasons.

First is to cope with a speed mismatch between producer and consumer of a data stream.

The second use of buffering is to provide adaptation for that have different data-transfer size.

Third use of buffering is to support copy semantics for the application I/O. "copy semantic " means, suppose that an application wants to write data on disk that is stored in its buffer. it calls the write() system s=call, providing a pointer to the buffer and the integer specify the number of bytes to write.

After the system call returns, what happens if the application of the buffer changes the content of the buffer? With copy semantic, the version of the data written to the disk is guaranteed to be the version at the time of the application system call.

Caching –

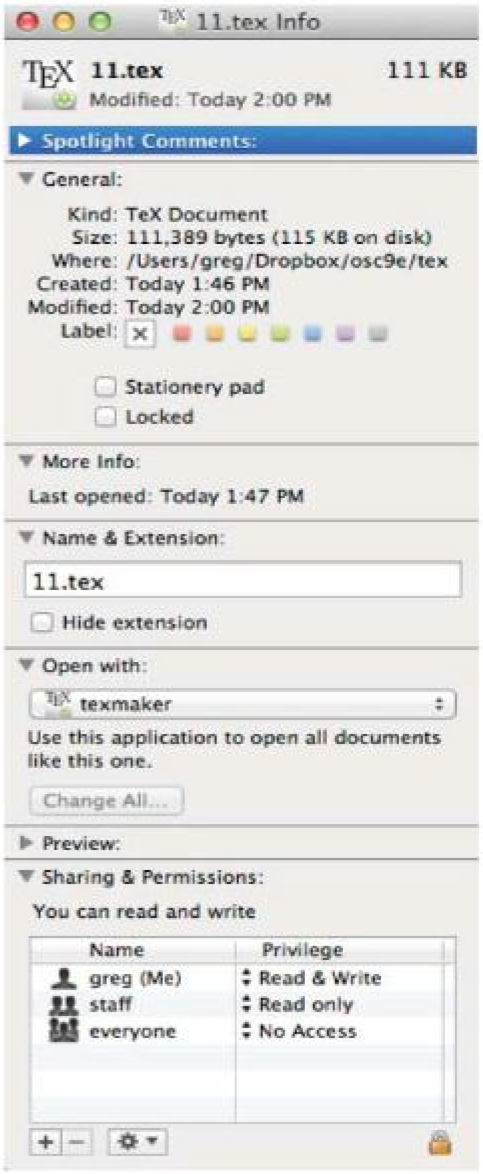
A *cache* is a region of fast memory that holds copy of data. Access to the cached copy is much easier than the original file. For instance, the instruction of the currently running process is stored on the disk, cached in physical memory, and copies again in the CPU's secondary and primary cache.

The main difference between a buffer and a cache is that a buffer may hold only the existing copy of data item, while cache, by definition, holds a copy on faster storage of an item that resides elsewhere.

Spooling and Device Reservation –

A *spool* is a buffer that holds the output of a device, such as a printer that cannot accept interleaved data stream. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixes together.

The OS solves this problem by preventing all output continuing to the printer. The output of all application is spooled in a separate disk file. When an application finishes printing then the spooling system queues the corresponding spool file for output to the printer.



11.tex Info

11.tex 111 KB
Modified: Today 2:00 PM

Spotlight Comments:

General:

Kind: TeX Document
Size: 111,389 bytes (115 KB on disk)
Where: /Users/greg/Dropbox/osc9e/tex
Created: Today 1:46 PM
Modified: Today 2:00 PM
Label:

Stationery pad
Locked

More Info:
Last opened: Today 1:47 PM

Name & Extension:
11.tex
Hide extension

Open with:
texmaker
Use this application to open all documents like this one.
Change All...

Preview:

Sharing & Permissions:
You can read and write

Name	Privilege
greg (Me)	Read & Write
staff	Read only
everyone	No Access

Error Handling –

An Os that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical glitch, Devices, and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded or for permanent reasons, as when a disk controller becomes defective.

I/O Protection –

Errors and the issue of protection are closely related. A user process may attempt to issue illegal I/O instruction to disrupt the normal function of a system. we can use the various mechanisms to ensure that such disruption cannot take place in the system.

To prevent illegal I/O access, we define all I/O instruction to be privileged instructions. The user cannot issue I/O instruction directly.

File Concept:

File Attributes:

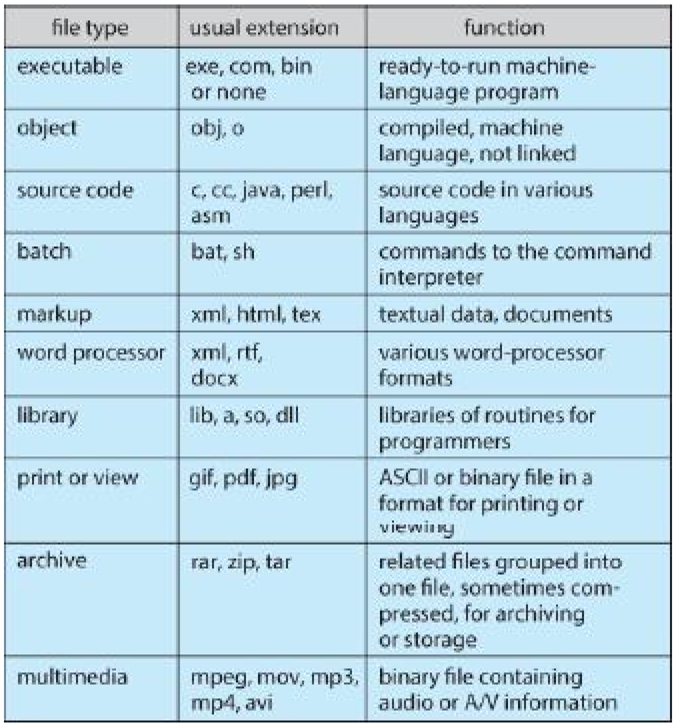
- Different OSes keep track of different file attributes, including:
    - **Name** - Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
    - **Identifier** (e.g. inode number)
    - **Type** - Text, executable, other binary, etc.
    - **Location** - on the hard drive.
    - **Size**
    - **Protection**
    - **Time & Date**
    - **User ID**

File Operations:

- The file ADT supports many common operations:
    - Creating a file
    - Writing a file
    - Reading a file
    - Repositioning within a file
    - Deleting a file
    - Truncating a file.
- Most Operating Systems require that files be ***opened*** before access and ***closed*** after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an ***open file table***, containing for example:
    - **File pointer** - records the current position in the file, for the next read or write access.
    - **File-open count** - How many times has the current file been opened (simultaneously by different processes) and not yet closed? When this counter reaches zero the file can be removed from the table.
    - **Disk location of the file.**
    - **Access rights**
- Some systems provide support for ***file locking.***
    - A ***shared lock*** is for reading only.
    - A ***exclusive lock*** is for writing as well as reading.

- An ***advisory lock*** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)
- A ***mandatory lock*** is enforced. (A truly locked door.)
- UNIX used advisory locks, and Windows uses mandatory locks.

File Types:



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

- Windows (and some other systems) use special file extensions to indicate the type of each file
- Macintosh stores a creator attribute for each file, according to the program that first created it with the create( ) system call.
- UNIX stores magic numbers at the beginning of certain files. (Experiment with the "file" command, especially in directories such as /bin and /dev)

File Structure:

- Some files contain an internal structure, which may or may not be known to the OS.
- For the OS to support particular file formats increases the size and complexity of the OS.
- UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)
- Macintosh files have two ***forks*** - a ***resource fork***, and a ***data fork***. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

Internal File Structure

- Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer.)
- Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.
- The number of logical units which fit into one physical block determines its ***packing***, and has an impact on the amount of internal fragmentation (wasted space) that occurs.
- As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

<u>Access Methods</u>

1 Sequential Access:

- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
    - read next - read a record and advance the tape to the next position.
    - write next - write a record and advance the tape to the next position.
    - rewind
    - skip n records - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

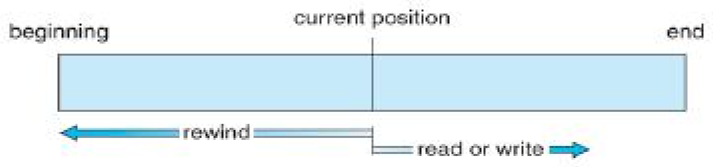
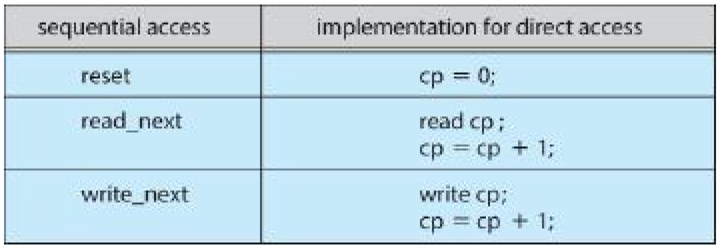


Fig - Sequential-access file.

2 Direct Access:

- Jump to any record and read that record. Operations supported include:
    - read n - read record number n. (Note an argument is now required.)
    - write n - write record number n. (Note an argument is now required.)
    - jump to record n - could be 0 or the end of file.
    - Query current record - used to return back to this record later.
    - Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.



sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

Fig - Simulation of sequential access on a direct-access file.

3 Other Access Methods:

- An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

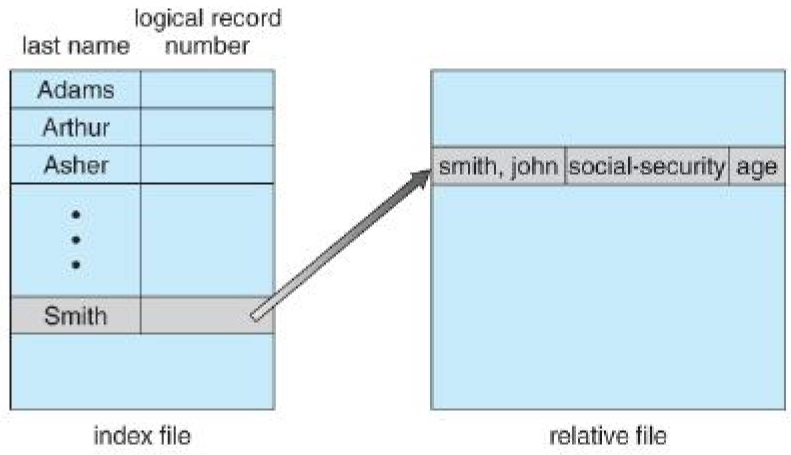
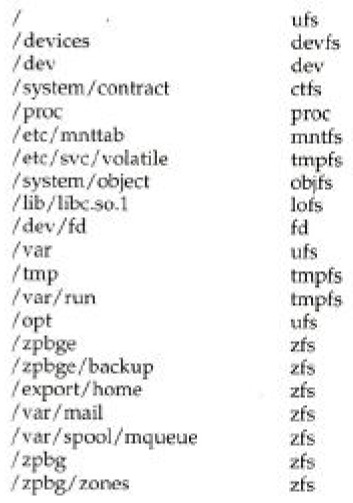


Fig - Example of index and relative files.

<u>Directory Structure</u>

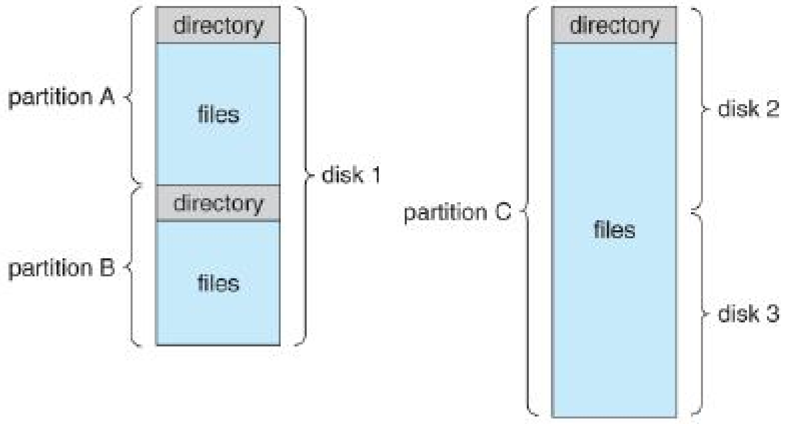
1 Storage Structure

- A disk can be used in its entirety for a file system.
- Alternatively a physical disk can be broken up into multiple ***partitions, slices, or mini-disks***, each of which becomes a virtual disk and can have its own file system. (or be used for raw storage, swap space, etc.)
- Or, multiple physical disks can be combined into one ***volume***, i.e. a larger virtual disk, with its own file system spanning the physical disks.



/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

Figure 11.8 Solaris file systems.



2 Directory Overview

- Directory operations to be supported include:
  - Search for a file
  - Create a file - add to the directory
  - Delete a file - erase from the directory
  - List a directory - possibly ordered in different ways.
  - Rename a file - may change sorting order
  - Traverse the file system.

3. Single-Level Directory

- Simple to implement, but each file must have a unique name.

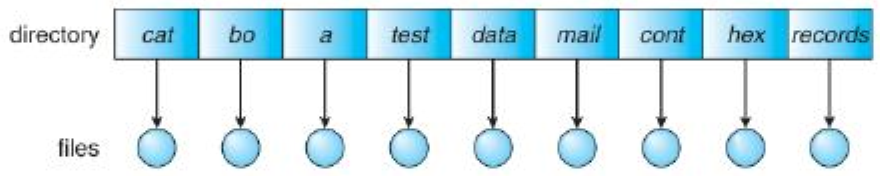


Fig - Single-level directory.

4 Two-Level Directory

- Each user gets their own directory space.
- File names only need to be unique within a given user's directory.
- A master file directory is used to keep track of each user's directory, and must be maintained when users are added to or removed from the system.
- A separate directory is generally needed for system (executable) files.

- Systems may or may not allow users to access other directories besides their own
    - If access to other directories is allowed, then provision must be made to specify the directory being accessed.
    - If access is denied, then special consideration must be made for users to run programs located in system directories. A ***search path*** is the list of directories in which to search for executable programs, and can be set uniquely for each user.

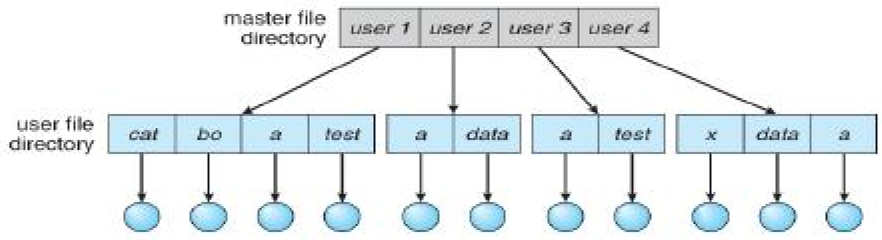


Fig - Two-level directory structure.

5 Tree-Structured Directories

- An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar.
- Each user / process has the concept of a ***current directory*** from which all (relative) searches take place.
- Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)
- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.
- One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

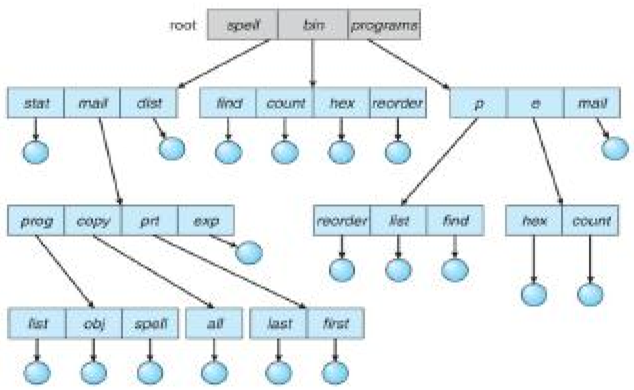


Fig - Tree-structured directory structure.

6 Acyclic-Graph Directories

- When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the ***directed*** arcs from parent to child.)
- UNIX provides two types of ***links*** for implementing the acyclic-graph structure.
    - A ***hard link*** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.
    - A ***symbolic link*** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.
- Windows only supports symbolic links, termed ***shortcuts.***
- Hard links require a ***reference count***, or ***link count*** for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.
- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:
    - One option is to find all the symbolic links and adjust them also.
    - Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
    - What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

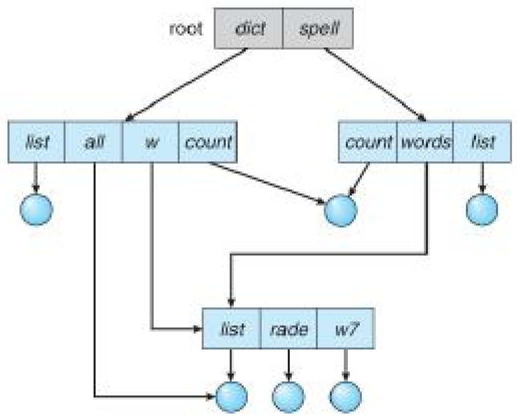


Fig - Acyclic-graph directory structure.

7 General Graph Directory

- If cycles are allowed in the graphs, then several problems can arise:
    - Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)
    - Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted.)

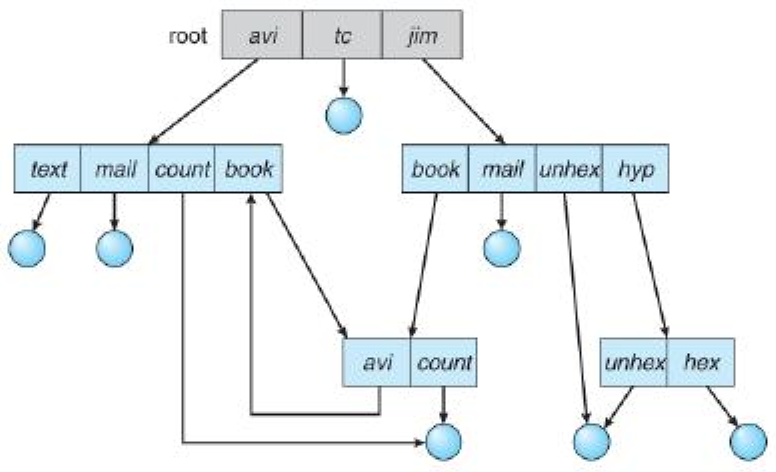


Fig - General graph directory.

File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a filesystem to mount and a ***mount point*** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.
- File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. (E.g. root may

allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted.

- File systems may be mounted read-only, or have other restrictions imposed.

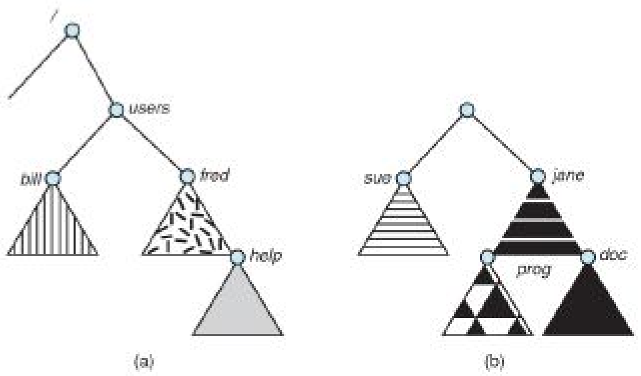


Fig- File system. (a) Existing system. (b) Unmounted volume.

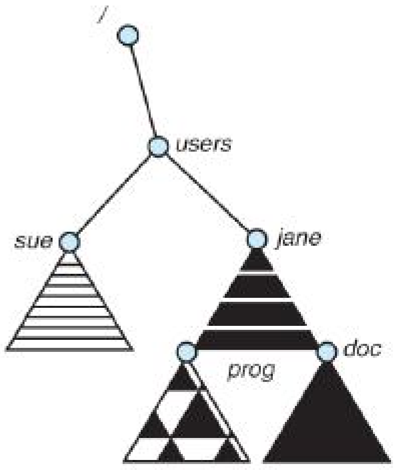


Fig - Mount point.

- The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.
- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow file systems to be mounted to any directory in the file system, much like UNIX.

File Sharing

1 Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner (user) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner (**U**ser), the **G**roup, and to the rest of the world (the universe, a.k.a. **O**thers.)
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

2 Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
  - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account or password controlled, or ***anonymous***, not requiring any user name or password.
  - Various forms of ***distributed file systems*** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
  - The WWW has made it easy once again to access files on remote systems without mounting their file systems, generally using (anonymous) ftp as the underlying file transport mechanism.

2.1 The Client-Server Model

- When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a ***server***, and the system which mounts them is the ***client.***
- User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)
- The same computer can be both a client and a server. (E.g. cross-linked file systems.)
- There are a number of security concerns involved in this model:

    - Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
    - Servers may restrict remote access to read-only.
    - Servers restrict which file systems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.
- The NFS (Network File System) is a classic example of such a system.

2.2 Distributed Information Systems

- The ***Domain Name System, DNS,*** provides for a unique naming system across all of the Internet.
- Domain names are maintained by the ***Network Information System, NIS***, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.
- Microsoft's ***Common Internet File System, CIFS***, establishes a ***network login*** for each user on a networked system with shared file access. Older Windows systems used ***domains***, and newer systems (XP, 2000), use ***active directories.*** User names must match across the network for this system to be valid.
- A newer approach is the ***Lightweight Directory-Access Protocol, LDAP,*** which provides a ***secure single sign-on*** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

2.3 Failure Modes

- When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

Protection

- Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies. This section discusses the latter.
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

1 Types of Access

- The following low-level operations are often controlled:
  - Read - View the contents of the file
  - Write - Change the contents of the file.
  - Execute - Load the file onto the CPU and follow the instructions contained therein.
  - Append - Add to the end of an existing file.
  - Delete - Remove a file from the system.
  - List -View the name and other attributes of files on the system.
- Higher-level operations, such as copy, can generally be performed through combinations of the above.

2 Access Control

- One approach is to have complicated ***Access Control Lists, ACL,*** which specify exactly what access is allowed or denied for specific users or groups.
  - The AFS uses this system for distributed access.
  - Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

bit	**Files**	**Directories**
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write (change)	Change directory contents. Required to create or delete files.

	file contents.	
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

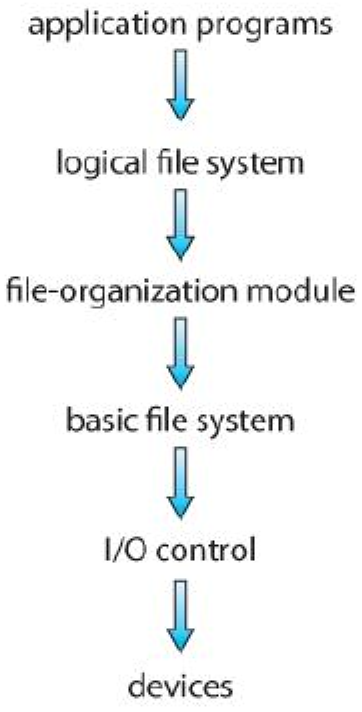
- In addition there are some special bits that can also be applied:
    - The set user ID (SUID) bit and/or the set group ID (SGID) bits applied to executable files temporarily change the identity of whoever runs the program to match that of the owner / group of the executable program. This allows users running specific programs to have access to files (***while running that program***) to which they would normally be unable to access. Setting of these two bits is usually restricted to root, and must be done with caution, as it introduces a potential security leak.
    - The sticky bit on a directory modifies write permission, allowing users to only delete files for which they are the owner. This allows everyone to create files in /tmp, for example, but to only delete files which they have created, and not anyone else's.
    - The SUID, SGID, and sticky bits are indicated with an S, S, and T in the positions for execute permission for the user, group, and others, respectively. If the letter is lower case, (s, s, t), then the corresponding execute permission is not also given. If it is upper case, (S, S, T), then the corresponding execute permission IS given.

File-System Implementation

File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
    - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.

- ***I/O Control*** consists of ***device drivers***, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ***ports***) that it listens to, and a unique set of command codes and results codes that it understands.
- The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.



- The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
- The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.

- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be file system specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)

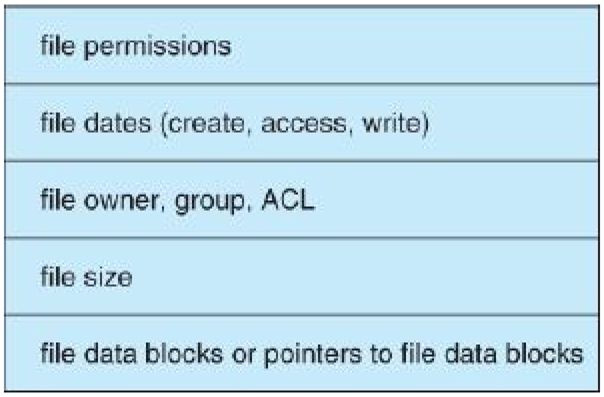
<u>File-System Implementation</u>

1 Overview

- File systems store several important data structures on the disk:
  - A ***boot-control block***, (per volume) a.k.a. the ***boot block*** in UNIX or the ***partition boot sector*** in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the

volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.

- A ***volume control block,*** (per volume) a.k.a. the ***master file table*** in UNIX or the ***superblock*** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
- A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a ***master file table.***
- The ***File Control Block, FCB,*** (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.



file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Fig - A typical file-control block.

- There are also several key data structures stored in memory:
    - An in-memory mount table.
    - An in-memory directory cache of recently accessed directory information.
    - ***A system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
    - ***A per-process open file table,*** containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)
- Figure illustrates some of the interactions of file system components when files are created and/or used:
    - When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.

- When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a ***file descriptor***, and Windows refers to it as a ***file handle***.
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

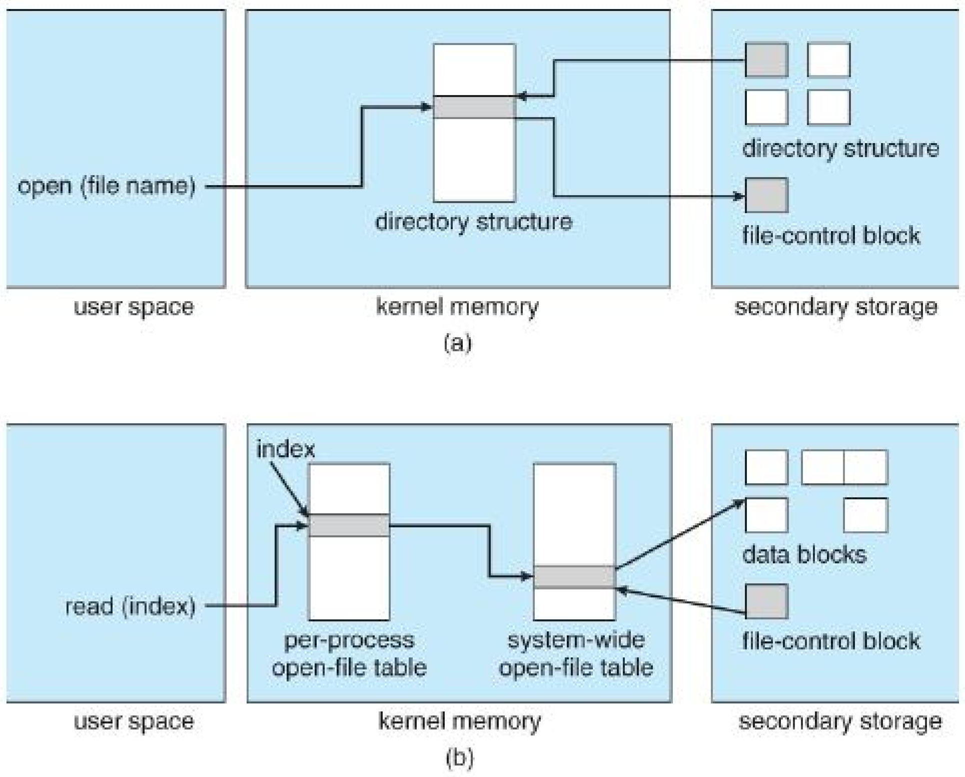


Fig- In-memory file-system structures. (a) File open. (b) File read.

2 Partitions and Mounting

- Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.
- Partitions can either be used as raw devices (with no structure imposed upon them), or they can be formatted to hold a filesystem (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.
- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of which of several available systems to boot.
- The ***root partition*** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)
- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. Filesystems may be mounted either automatically or manually. In UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

3 Virtual File Systems

- ***Virtual File Systems, VFS***, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)
- The VFS in Linux is based upon four key object types:
  - The ***inode*** object, representing an individual file
  - The ***file*** object, representing an open file.
  - The ***superblock*** object, representing a filesystem.

    - The ***dentry*** object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. See /usr/include/linux/fs.h for full details. Common operations provided include open( ), read( ), write( ), and mmap( ).

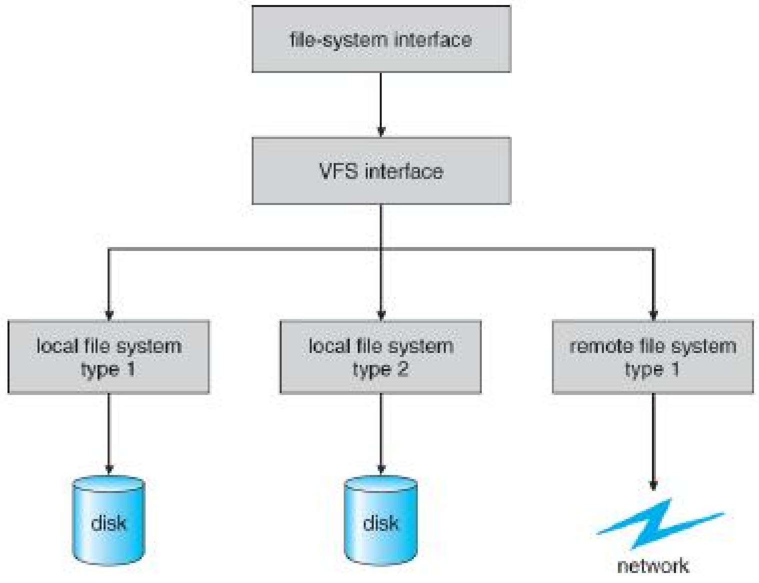


Fig - Schematic view of a virtual file system.

3 Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

1 Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file (or verifying one does not already exist upon creation) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.

- More complex data structures, such as B-trees, could also be considered.

2 Hash Table

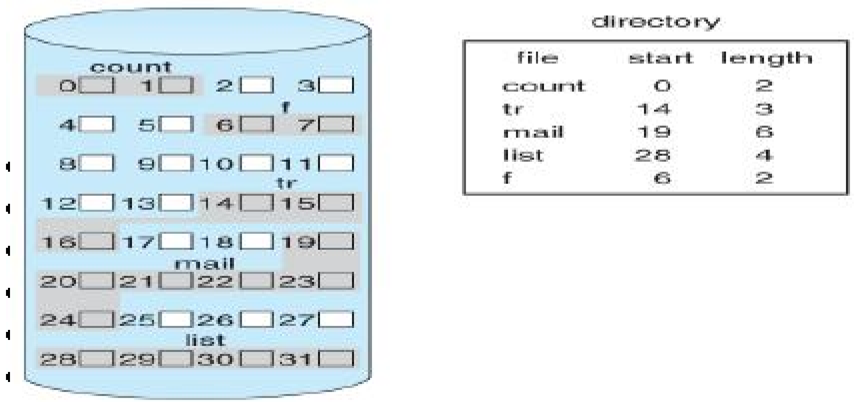
- A hash table can also be used to speed up searches.
- Hash tables are generally implemented ***in addition to*** a linear or other structure

Allocation Methods

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

1 Contiguous Allocation

- ***Contiguous Allocation*** requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.
- Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.
- (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
    - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
    - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
    - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.



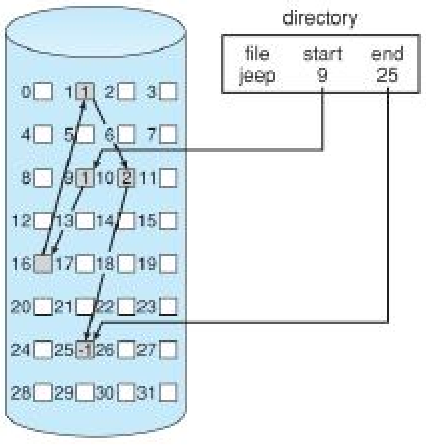
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Fig - Contiguous allocation of diskspace.

- A variation is to allocate file space in large contiguous chunks, called ***extents.*** When a file outgrows its original extent, then an additional one is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary.) The high-performance files system Veritas uses extents to optimize performance.

2 Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating ***clusters*** of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.



file	start	end
jeep	9	25

Fig - Linked allocation of disk space.

- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

The ***File Allocation Table, FAT,*** used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.

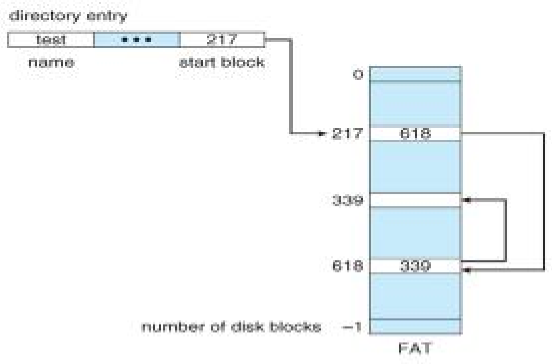


Fig- File-allocation table.

3 Indexed Allocation

- ***Indexed Allocation*** combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.

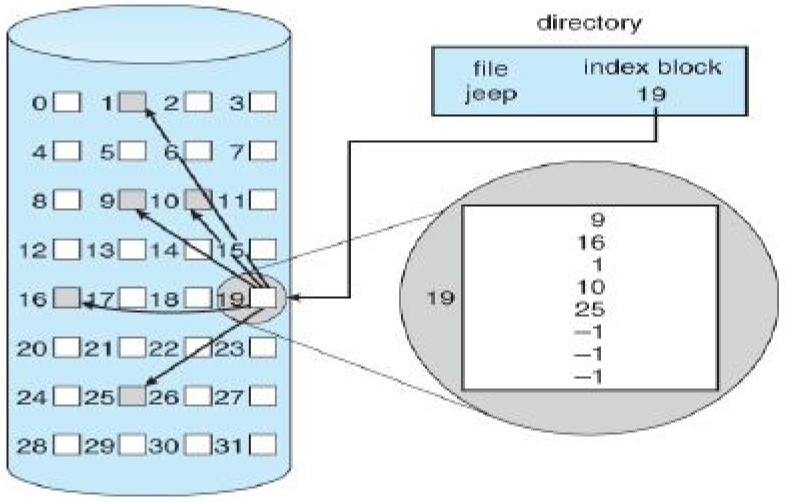


Fig - Indexed allocation of disk space.

- Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

- **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.)

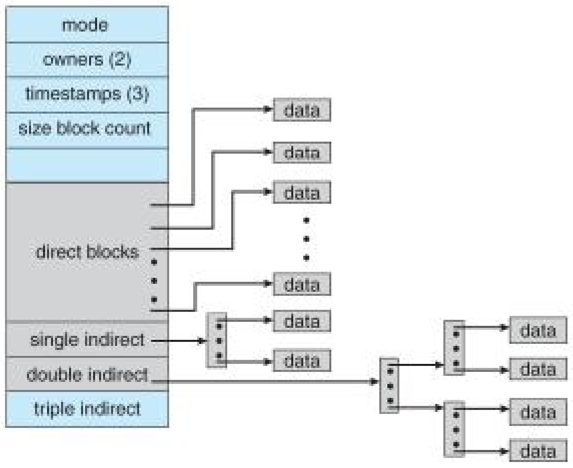


Fig - The UNIX inode.

4 Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.
- Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

- And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

<u>Free-Space Management</u>

- Another important aspect of disk management is keeping track of and allocating free space.

1 Bit Vector

- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap. (For example.)

2 Linked List

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.

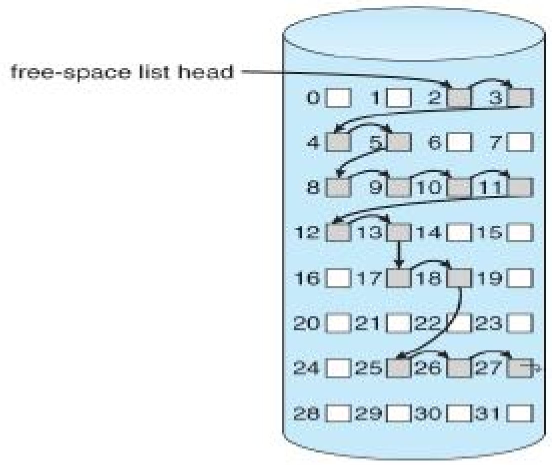


Fig - Linked free-space list on disk.

3 Grouping

- A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

4 Counting

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

5 Space Maps

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) ***metaslabs*** of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

Efficiency and Performance

1 Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created.
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.

- As technology advances, addressing schemes have had to grow as well.
    - Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!)
- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

2 Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.
- Some OSes cache disk blocks they expect to need again in a ***buffer cache.***
- A ***page cache*** connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.
- Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a ***unified virtual memory.***
- The following figures show the advantages of the ***unified buffer cache*** found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.

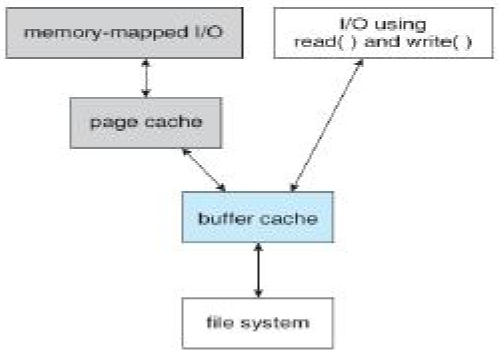


Fig - I/O without a unified buffer cache.

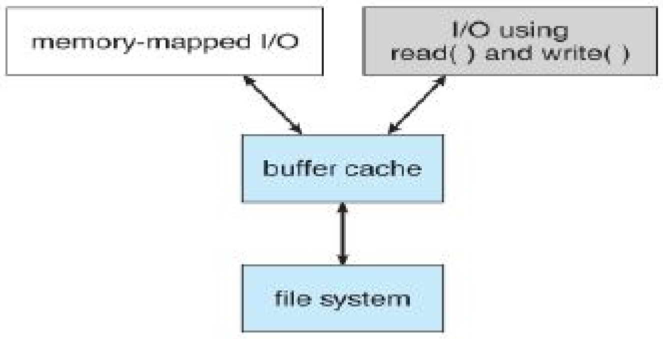


Fig - I/O using a unified buffer cache.

- Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in ***priority paging*** giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.

- Another issue affecting performance is the question of whether to implement ***synchronous writes*** or ***asynchronous writes.*** Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order (See Chapter 12.) Metadata writes are often done synchronously. Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.
- The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.) On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:
    - ***Free-behind*** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
    - ***Read-ahead*** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.
- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times. (See Chapter 12.) Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

Unit –V Operating System in Distributed Processing

Introduction: Manages a collection of independent computers and makes them appear to the users of the system as if it were a single computer. Distributed Computing Systems commonly use two types of Operating Systems. Network Operating Systems Distributed Operating System.

- A distributed system is A collection of independent computers that appears to its users as a single coherent system.

- Distributed computing is a field of computer science that studies distributed systems. A distributed system consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal.

- A computer program that runs in a distributed system is called a distributed program, and distributed programming is the process of writing such programs.
- Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers.
- A distributed system is a collection of independent computers, interconnected via a network, capable of collaborating on a task.
- Distributed computing is computing performed in a distributed system. Distributed computing has become increasingly common due advances that have made both machines and networks cheaper and faster.

Some examples of distributed systems:

♦ Local Area Network and Intranet

♦ Database Management System

♦ Automatic Teller Machine Network

♦ Internet/World-Wide Web

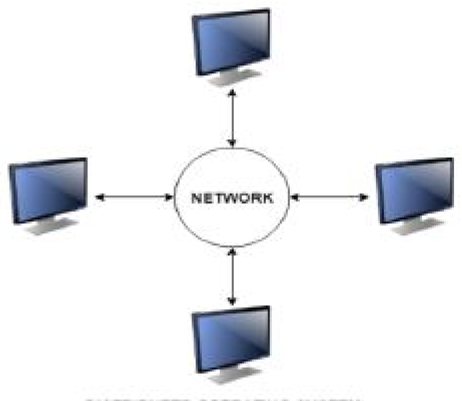
♦ Mobile and Ubiquitous Computing

Microcomputers are small computers having microprocessors as their central processor. Some of its advantages are portability, less costly, user-friendliness, thus making them ideal as home computers. There are mainframe computers, mini computers and microcomputers. Mainframes are large scale computers, (search IBM System/360) mini computer are smaller, a lot smaller than mainframes (search IBM AS/300) and minis are what's on your desktop like a DELL, HP, Sony, etc.

They are many different things between Micro and mini computers Micro Computer support one user at the same time desktop computers PDA laptops these are called Micro computers or Carry Along computers Mini computers support many users at the same time powerful than micro and expensive.

A distributed system contains multiple nodes that are physically separate but linked together using the network. All the nodes in this system communicate with each other and handle processes in tandem. Each of these nodes contains a small part of the distributed operating system software.

A diagram to better explain the distributed system is:



DISTRIBUTED OPERATING SYSTEM

Types of Distributed Systems

The nodes in the distributed systems can be arranged in the form of client/server systems or peer to peer systems. Details about these are as follows:

Client/Server Systems

In client server systems, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server. Both the client and server usually communicate via a computer network and so they are a part of distributed systems.

Peer to Peer Systems

The peer to peer systems contains nodes that are equal participants in data sharing. All the tasks are equally divided between all the nodes. The nodes interact with each other as required as share resources. This is done with the help of a network.

Advantages of Distributed Systems

Some advantages of Distributed Systems are as follows:

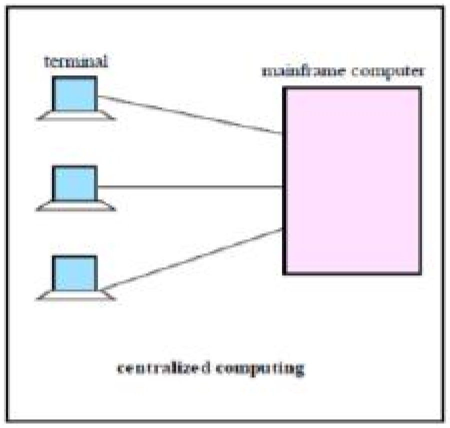
- All the nodes in the distributed system are connected to each other. So nodes can easily share data with other nodes.
- More nodes can easily be added to the distributed system i.e. it can be scaled as required.
- Failure of one node does not lead to the failure of the entire distributed system. Other nodes can still communicate with each other.
- Resources like printers can be shared with multiple nodes rather than being restricted to just one.

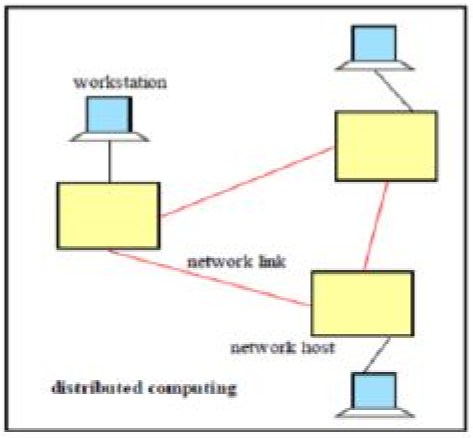
Disadvantages of Distributed Systems

Some disadvantages of Distributed Systems are as follows:

- It is difficult to provide adequate security in distributed systems because the nodes as well as the connections need to be secured.
- Some messages and data can be lost in the network while moving from one node to another.
- The database connected to the distributed systems is quite complicated and difficult to handle as compared to a single user system.
- Overloading may occur in the network if all the nodes of the distributed system try to send data at once.

Centralized and Distributed Computing





Centralized System Characteristics

- One component with non-autonomous parts.
- Component shared by users all the time
- All resources accessible
- Software runs in a single process
- Single point of control
- Single point of failure

Distributed System Characteristics

- Multiple autonomous components
- Components are shared by all users
- Resources may not be accessible
- Software runs in concurrent processes on different processes

- Multiple points of control
- Multiple points of failure

Advantages of Distributed Systems over Centralized System

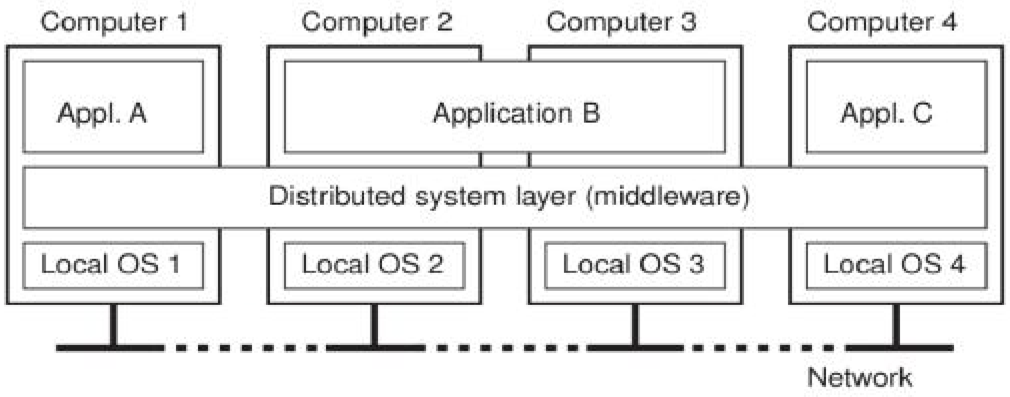
• Economics: a collection of microprocessors offer a better price/performance than mainframes. Low price/performance ratio: cost effective way to increase computing power.

• Speed: a distributed system may have more total computing power than a mainframe. • Inherent distribution: Some applications are inherently distributed. Ex. a supermarket chain. • Reliability: If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.

• Incremental growth: Computing power can be added in small increments. Modular expandability

• Another deriving force: the existence of large number of personal computers, the need for people to collaborate and share information.

The figure below shows a simple distributed systems for a number of applications running through different operating system where the middleware takes responsibility for the heterogeneity of the communications.



Why we need a distributed system is mainly for the following reasons:

- **Economics:** a collection of microprocessors offer a better price/performance than mainframes.
- **Speed:** a distributed system may have more total computing power than a mainframe. Enhanced performance through load distribution.

- **Inherent distribution:** Some applications are inherently distributed. Ex. a supermarket chain.
- **Reliability:** If one machine crashes, the system as a whole can still survive. Higher availability and improved reliability.
- **Incremental growth:** Computing power can be added in small increments. Modular expandability
- **Data sharing:** allow many users to access to a common data base
- **Resource Sharing:** expensive peripherals like color printers
- **Communication:** enhance human-to-human communication, e.g., email, chat
- **Flexibility:** spread the workload over the available machines
- Mobility: Access the system, data or resources from any place or device.
- …..

A distributed system can be much larger and more powerful given the combined capabilities of the distributed components, than combinations of stand-alone systems. But it must be reliable. This is a challenging goal to achieve because of the complexity of the interactions between simultaneously running components. To be truly reliable, a distributed system must have the following **characteristics:**

- **Fault-Tolerant:** It can recover from component failures without performing incorrect actions.
- **Highly Available:** It can restore operations, permitting it to resume providing services even when some components have failed.
- **Recoverable:** Failed components can restart themselves and re-join the system, after the cause of failure has been repaired.
- **Consistent:** The system can coordinate actions by multiple components often in the presence of concurrency and failure. This underlies the ability of a distributed system to act like a non-distributed system.
- **Scalable:** It can operate correctly even as some aspect of the system is scaled to a larger size.
- **Predictable Performance:** The ability to provide desired responsiveness in a timely manner.
- **Secure:** The system authenticates access to data and services Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or in the same room. Our definition of distributed systems has the following significant consequences:

- **Concurrency:** concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network.

- **No global clock:** When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the only communication is by sending messages through a network.
- **Independent failures:** All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures.

Common Examples for distributed systems include:

- network file system, network printer etc
- ATM (cash machine)
- Distributed databases
- Network computing
- Global positioning systems
- Retail point-of-sale terminals
- Air-traffic control
- Enterprise computing
- WWW

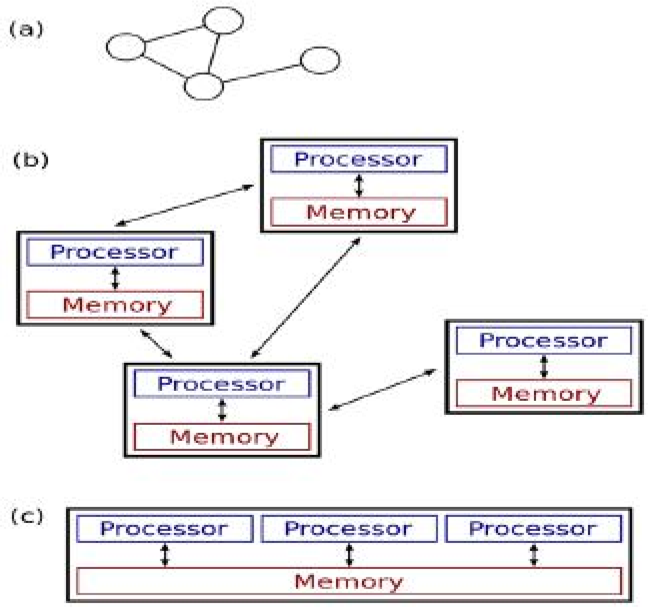
Parallel and distributed computing

Distributed systems are groups of networked computers which share a common goal for their work. The terms "concurrent computing", "parallel computing", and "distributed computing" have a lot of overlap, and no clear distinction exists between them. The same system may be characterized both as "parallel" and "distributed"; the processors in a typical distributed system run concurrently in parallel. Parallel computing may be seen as a particular tightly coupled form of distributed computing, and distributed computing may be seen as a loosely coupled form of parallel computing. Nevertheless, it is possible to roughly classify concurrent systems as "parallel" or "distributed" using the following criteria:

- In parallel computing, all processors may have access to a shared memory to exchange information between processors.
- In distributed computing, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

The figure on the right illustrates the difference between distributed and parallel systems. Figure (a) is a schematic view of a typical distributed system; the system is represented as a network topology in which each node is a computer and each line connecting the nodes is a communication link. Figure (b) shows the same distributed system in more detail: each computer has its own local memory, and information can be exchanged only by passing messages from one node to another by using the available communication links. Figure (c) shows a parallel system in which each processor has a direct access to a shared memory.

The situation is further complicated by the traditional uses of the terms parallel and distributed *algorithm* that do not quite match the above definitions of parallel and distributed *syste*. Nevertheless, as a rule of thumb, high-performance parallel computation in a shared-memory multiprocessor uses parallel algorithms while the coordination of a large-scale distributed system uses distributed algorithms.



Prime difference between network and distributed operating system is all of the above.

A network operating system is made up of software and associated protocols that allow a set of computer network to be used together. A distributed operating system is an ordinary centralized operating system but runs on multiple independent CPUs. Environment users are aware of multiplicity of machines.

A distributed operating system is a software over a collection of independent, networked, communicating, and physically separate computational nodes. They handle jobs which are serviced by multiple CPUs. Each individual node holds a specific software subset of the global aggregate operating system.

A distributed system is a system whose components are located on different networked computers, which then communicate and coordinate their actions by passing messages to one other. ... Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications.

Network OS and Distributed OS have a common hardware base, but the difference lies in the Software. ... Control over file placement is done manually by the user in network OS.But, in distributed operating system it can be done automatically by the system itself.

A process is the unit of work in a computer system. A process must be in main memory during execution. To improve the utilization of central processing unit (CPU) as well as the speed of its response to its users, the computer must keep several processes in memory. ... A process can be thought of as a program in execution.

Process management is an integral part of any modern-day operating system (OS). To meet these requirements, the OS must maintain a data structure for each process, which describes the state and resource ownership of that process, and which enables the OS to exert control over each process.

Challenges in distributed systems.

1 – Heterogeneity

Heterogeneity – "Describes a system consisting of multiple distinct components" Of course heterogeneity applies to pretty much anything which is made up of many different items or objects including Food!

Anyway, in many systems in order to overcome heterogeneity a software layer known as **Middleware** is often used to hide the differences amongst the components underlying layers.

2 – Openness

Openness –"Property of each subsystem to be open for interaction with other systems"

So once something has been published it cannot be taken back or reversed. Furthermore in open distributed systems there is often no central authority, as different systems may have their own intermediary.

3 – Security

The issues surrounding security are those of

- Confidentiality
- Integration
- Availability

To combat these issues encryption techniques such as those of cryptography can help but they are still not absolute. Denial of Service attacks can still occur, where a server or service is bombarded with false requests usually by botnets (zombie computers).

4 – Scalability

Basically a system is described as scalable if:

"As the system, number of resources, or users increase the performance of the system is not lost and remains effective in accomplishing its goals"

That's a fairly self explanatory description, but there are a number of important issues that arise as a result of increasing scalability, such as increase in cost and physical resources. It is also important to avoid performance bottlenecks by using caching and replication.

5 – Fault handling

Failures are inevitable in any system; some components may stop functioning while others continue running normally. So naturally we need a way to:

Detect Failures – Various mechanisms can be employed such as checksums.

Mask Failures – retransmit upon failure to receive acknowledgement

Recover from failures – if a server crashes roll back to previous state

Build Redundancy – Redundancy is the best way to deal with failures. It is achieved by replicating data so that if one sub system crashes another may still be able to provide the required information.

6 – Concurrency

Concurrency issues arise when several clients attempt to request a shared resource at the same time. This is problematic as the outcome of any such data may depend on the execution order, and so synchronization is required.

7 – Transparency

A distributed system must be able to offer transparency to its users. As a user of a distributed system you do not care if we are using 20 or 100's of machines, so we hide this information, presenting the structure as a normal centralized system.

- Access Transparency – where resources are accessed in a uniform manner regardless of location
- Location Transparency – the physical location of a resource is hidden from the user
- Failure Transparency – Always try and Hide failures from users (see challenge No.5)