UNIT I

INTRODUCTION

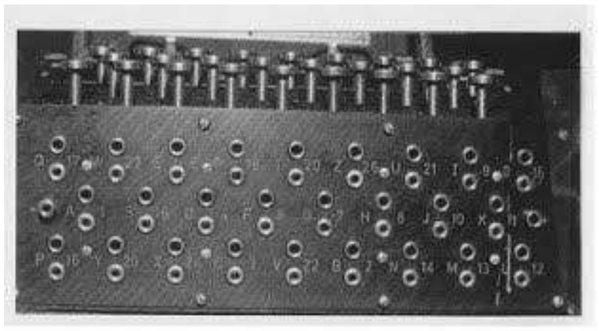
OPERATING SYSTEM

- An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.
- An OS act as an intermediary between user and hardware.
- An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users.
- The OS also manages secondary memory and I/O (input/output) devices on behalf of its users.

HISTORY OF OPERATING SYSTEM

The First Generation (1940's to early 1950's)

- When electronic computers where first introduced in the 1940's they were created without any operating systems.
- All programming was done in absolute machine language, often by wiring up plugboards to control the machine's basic functions.
- During this generation computers were generally used to solve simple math calculations, operating systems were not necessarily needed.



The Second Generation (1955-1965)

- The first operating system was introduced in the early 1950's, it was called GMOS and was created by General Motors for IBM's machine the 701.
- Operating systems in the 1950's were called single-stream batch processing systems because the data was submitted in groups.
- These new machines were called mainframes, and they were used by professional operators in large computer rooms. Since there was such as high price tag on these machines, only government agencies or large corporations were able to afford them.

The Third Generation (1965-1980)

- By the late 1960's operating systems designers were able to develop the system of multiprogramming in which a computer program will be able to perform multiple jobs at the same time.
- The introduction of multiprogramming was a major part in the development of operating systems because it allowed a CPU to be busy nearly 100 percent of the time that it was in operation.
- Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18-bit

words, but at $120,000 per machine (less than 5 percent of the price of a 7094), it sold like hotcakes.

- These microcomputers help create a whole new industry and the development of more PDP's. These PDP's helped lead to the creation of personal computers which are created in the fourth generation.

The Fourth Generation (1980-Present Day)

- The fourth generation of operating systems saw the creation of personal computing. Although these computers were very similar to the minicomputers developed in the third generation, personal computers cost a very small fraction of what minicomputers cost.
- A personal computer was so affordable that it made it possible for a single individual could be able to own one for personal use while minicomputers where still at such a high price that only corporations could afford to have them.
- One of the major factors in the creation of personal computing was the birth of Microsoft and the Windows operating system.
- The windows Operating System was created in 1975 when Paul Allen and Bill Gates had a vision to take personal computing to the next level.
- They introduced the MS-DOS in 1981 although it was effective it created much difficulty for people who tried to understand its cryptic commands. Windows went on to become the largest operating system used in techonology today with releases of Windows 95, Windows 98, WIndows XP (Which is currently the most used operating system to this day), and their newest operating system Windows 7.
- Along with Microsoft, Apple is the other major operating system created in the 1980's. Steve Jobs, co founder of Apple, created the Apple Macintosh which was a huge success due to the fact that it was so user friendly.
- Windows development throughout the later years were influenced by the Macintosh and it created a strong competition between the two companies.
- Today all of our electronic devices run off of operating systems, from our computers and smartphones, to ATM machines and motor vehicles. And as technology advances, so do operating systems.

DIFFERENT KINDS OF OPERATING SYSTEM (OR) TYPES OF OPERATING SYSTEMS

Following are some of the most widely used types of Operating system.

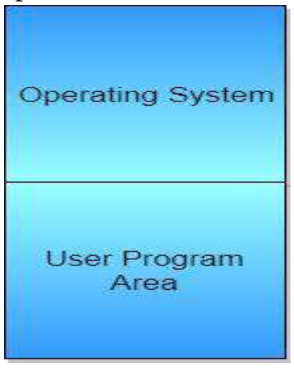
1. Simple Batch System
2. Multiprogramming Batch System
3. Multiprocessor System
4. Desktop System
5. Distributed Operating System
6. Clustered System
7. Realtime Operating System
8. Handheld System

1. Simple Batch Systems

- In this type of system, there is **no direct interaction between user and the computer**.
- The user has to submit a job (written on cards or tape) to a computer operator.
- Then computer operator places a batch of several jobs on an input device.
- Jobs are batched together by type of languages and requirement.
- Then a special program, the monitor, manages the execution of each program in the batch.
- The monitor is always in the main memory and available for execution.

Advantages of Simple Batch Systems

1. No interaction between user and computer.
2. No mechanism to prioritise the processes.

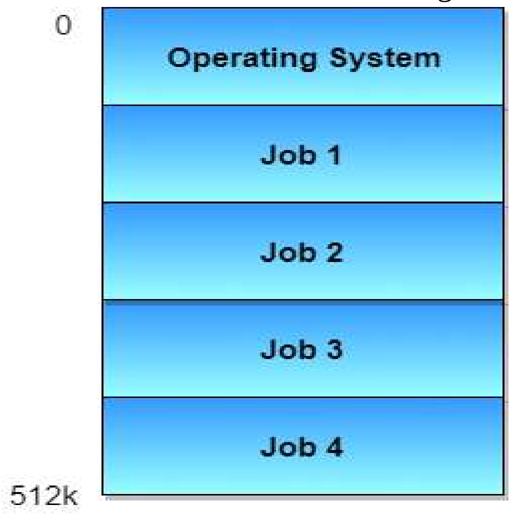


2. Multiprogramming Batch Systems

- In this the operating system picks up and begins to execute one of the jobs from memory.
- Once this job needs an I/O operation operating system switches to another job (CPU and OS always busy).
- Jobs in the memory are always less than the number of jobs on disk(Job Pool).
- If several jobs are ready to run at the same time, then the system chooses which one to run through the process of **CPU Scheduling**.

- In Non-multiprogrammed system, there are moments when CPU sits idle and does not do any work.
- In Multiprogramming system, CPU will never be idle and keeps on processing.

Time Sharing Systems are very similar to Multiprogramming batch systems. In fact time sharing systems are an extension of multiprogramming systems.
In Time sharing systems the prime focus is on **minimizing the response time**, while in multiprogramming the prime focus is to maximize the CPU usage.



3. **Multiprocessor Systems**
   - A Multiprocessor system consists of several processors that share a common physical memory. Multiprocessor system provides higher computing power and speed. In multiprocessor system all processors operate under single operating system. Multiplicity of the processors and how they do act together are transparent to the others.

Advantages of Multiprocessor Systems

1. Enhanced performance
2. Execution of several tasks by different processors concurrently, increases the system's throughput without speeding up the execution of a single task.
3. If possible, system divides task into many subtasks and then these subtasks can be executed in parallel in different processors. Thereby speeding up the execution of single tasks.

4. Desktop Systems

- Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs. PC operating systems therefore were neither **multiuser** nor **multitasking**. However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness. These systems are called **Desktop Systems** and include PCs running Microsoft Windows and the Apple Macintosh. Operating systems for these computers have benefited in several ways from the development of operating systems for **mainframes**.
- **Microcomputers** were immediately able to adopt some of the technology developed for larger operating systems. On the other hand, the hardware costs for microcomputers are sufficiently **low** that individuals have sole use of the computer, and CPU utilization is no longer a prime concern. Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

5. Distributed Operating System

- The motivation behind developing distributed operating systems is the availability of powerful and inexpensive microprocessors and advances in communication technology.
- These advancements in technology have made it possible to design and develop distributed systems comprising of many computers that are inter connected by communication networks. The main benefit of distributed systems is its low price/performance ratio.

Advantages Distributed Operating System

1. As there are multiple systems involved, user at one site can utilize the resources of systems at other sites for resource-intensive tasks.
2. Fast processing.
3. Less load on the Host Machine.

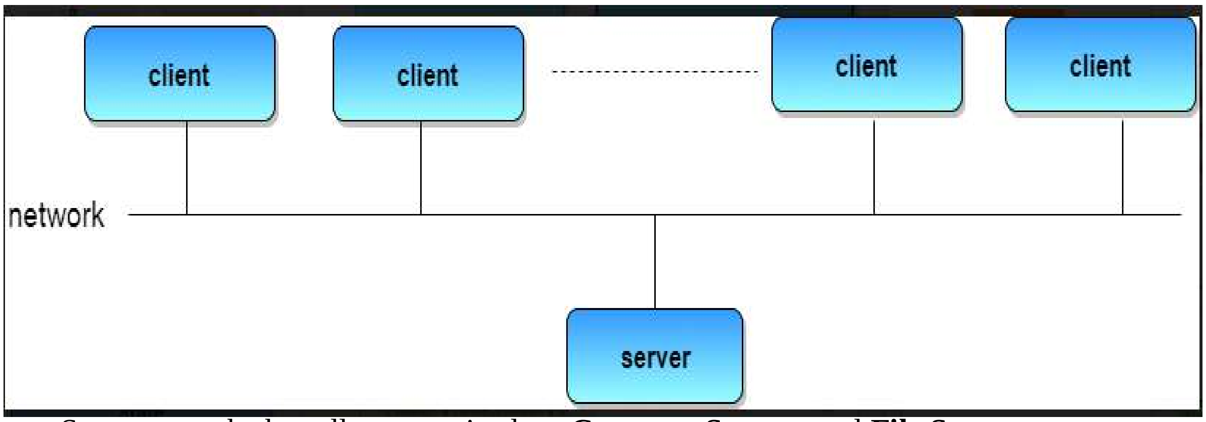
Types of Distributed Operating Systems

Following are the two types of distributed operating systems used:

1. Client-Server Systems
2. Peer-to-Peer Systems

Client-Server Systems

Centralized systems today act as **server systems** to satisfy requests generated by **client systems**. The general structure of a client-server system is depicted in the figure below:



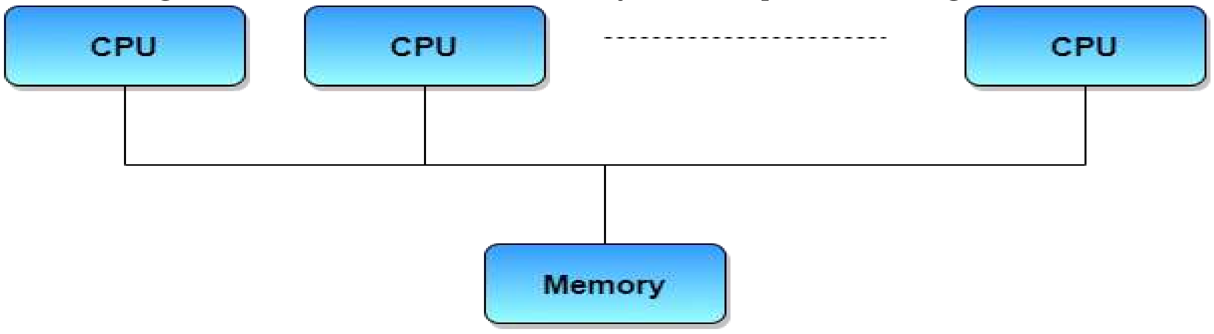
Server Systems can be broadly categorized as: **Compute Servers** and **File Servers**.

- **Compute Server systems**, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.
- **File Server systems**, provide a file-system interface where clients can create, update, read, and delete files.

Peer-to-Peer Systems

- The growth of computer networks - especially the Internet and World Wide Web (WWW) – has had a profound influence on the recent development of operating systems. When PCs were introduced in the 1970s, they were designed for **personal** use and were generally considered standalone computers. With the beginning of widespread public use of the Internet in the 1990s for electronic mail and FTP, many PCs became connected to computer networks.
- In contrast to the **Tightly Coupled** systems, the computer networks used in these applications consist of a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.

These systems are usually referred to as loosely coupled systems (or distributed systems). The general structure of a client-server system is depicted in the figure below:



6. Clustered Systems

- Like parallel systems, clustered systems gather together multiple CPUs to accomplish computational work.
- Clustered systems differ from parallel systems, however, in that they are composed of two or more individual systems coupled together.
- The definition of the term clustered is **not concrete;** the general accepted definition is that clustered computers share storage and are closely linked via LAN networking.
- Clustering is usually performed to provide **high availability**.
- A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others. If the monitored machine fails, the monitoring machine can take ownership of its storage, and restart the application(s) that were running on the failed machine. The failed machine can remain down, but the users and clients of the application would only see a brief interruption of service.
- **Asymmetric Clustering -** In this, one machine is in hot standby mode while the other is running the applications. The hot standby host (machine) does nothing but monitor the active server. If that server fails, the hot standby host becomes the active server.
- **Symmetric Clustering -** In this, two or more hosts are running applications, and they are monitoring each other. This mode is obviously more efficient, as it uses all of the available hardware.
- **Parallel Clustering -** Parallel clusters allow multiple hosts to access the same data on the shared storage. Because most operating systems lack support for this simultaneous data access by multiple hosts, parallel clusters are usually accomplished by special versions of software and special releases of applications.

Clustered technology is rapidly changing. Clustered system's usage and it's features should expand greatly as **Storage Area Networks(SANs)**. SANs allow easy attachment of multiple hosts to multiple storage units. Current clusters are usually limited to two or four hosts due to the complexity of connecting the hosts to shared storage.

7. Real Time Operating System

- It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.
- The Real-Time Operating system which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems.**
- While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but no assurity of completeing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

8. Handheld Systems

Handheld systems include **Personal Digital Assistants(PDAs)**, such as Palm-Pilots or Cellular Telephones with connectivity to a network such as the Internet. They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.

- Many handheld devices have between **512 KB** and **8 MB** of memory. As a result, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager once the memory is no longer being used.
- Currently, many handheld devices do **not use virtual memory** techniques, thus forcing program developers to work within the confines of limited physical memory.
- Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require **more power**. To include a faster processor in a handheld device would require a **larger battery** that would have to be replaced more frequently.
- The last issue confronting program designers for handheld devices is the small display screens typically available. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

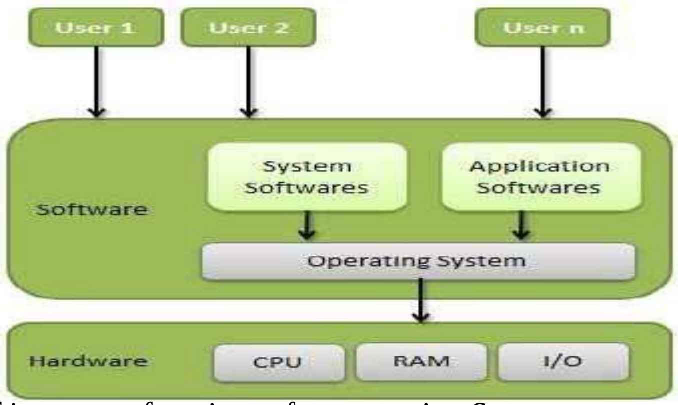
Some handheld devices may use wireless technology such as **BlueTooth**, allowing remote access to e-mail and web browsing. **Cellular telephones** with connectivity to the Internet fall into this category. Their use continues to expand as network connections become more available and other options such as cameras and MP3 players, expand their utility.

AN OPERATING SYSTEM CONCEPTS

- An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.
- Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

Definition

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



Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security

- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

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

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

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

Processor Management

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

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

Device Management

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

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

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

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

Other Important Activities

Following are some of the important activities that an Operating System performs −

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.

- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

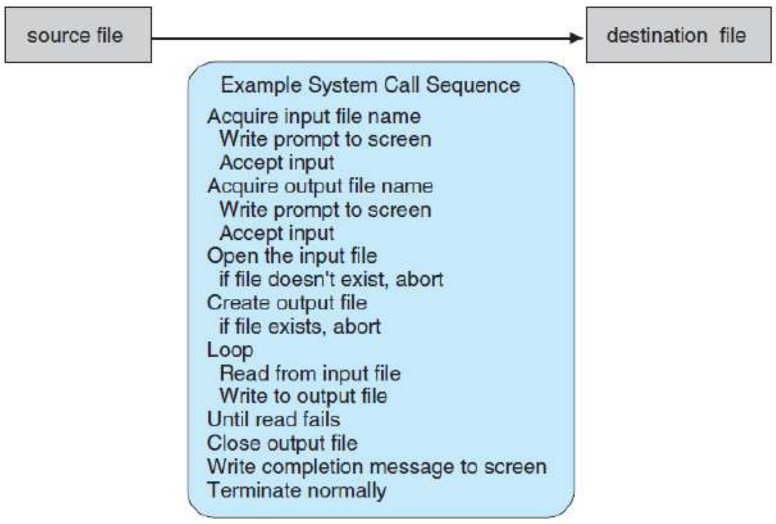
SYSTEM CALLS

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++.

For example, simple program to read data from one file and copy them to another file.

- The first input that the program will need is the names of the two files: the input file and the output file.
- Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call.
- Possible error conditions for each operation can require additional system calls.
- When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call).
- Each read and write must return status information regarding various possible error conditions.
- Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call).

This system-call sequence is shown in the Figure. Example system calls are shown here.



```
Example System Call Sequence
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

- Three of the common APIs available to application programmers are
- Windows API for Windows systems,
- POSIX API for POSIX-based systems (virtually all versions of UNIX, Linux, and Mac OSX),

- Java API for programs that run on the Java virtual machine.
- A programmer accesses an API via a library of code provided by the operating system.
- An application programmer prefer programming according to an API rather than invoking actual system calls. One benefit concerns program portability.

System call Implementation

- The run-time support system provides a **system call interface** that serves as the link to system calls made available by the operating system.
- The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system.
- A number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.
- The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call.
- Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.
- The relationship between an API, the system-call interface, and the operating system is shown in Figure, which illustrates how the operating system handles a user application invoking the open() system call.

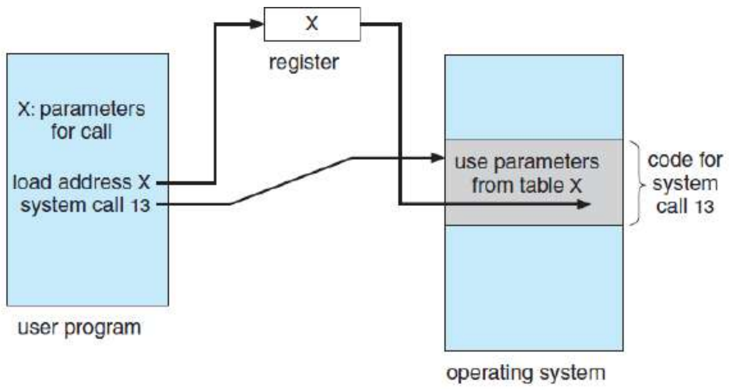
System calls – Parameter passing

- System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call.

Three general methods are used to pass parameters to the operating system.

The simplest approach is to pass the parameters in registers.

If more parameters are present than registers, then the parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register (Figure). This is the approach taken by Linux and Solaris.

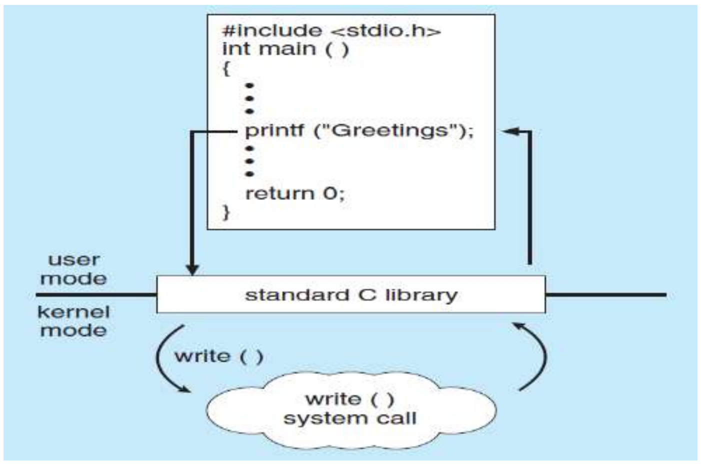


Passing parameters as Table

- Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.

Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

Example System call



TYPES OF SYSTEM CALLS

System calls can be grouped roughly into six major categories.

1 . Process control
2. File manipulation
3. Device manipulation
4. Information maintenance,
5. Communications
6. Protection

1. Process Control

- A running program needs to be able to halt its execution either normally (**end**()) or abnormally (**abort**()).
- If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.
- Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.
- In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.
- In a GUI system, a pop-up window might alert the user to the error and ask for guidance.
- In a batch system, the command interpreter usually terminates the entire job and continues with the next job.
- A process or job executing one program may want to **load**() and **execute**() another program.

- If more programs continue concurrently, a new job or process created is to be multiprogrammed. Often, there is a system call specifically for this purpose(**create process**() or **submit job**()).
- If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (**get process attributes**() and **set process attributes**())
- To terminate a job or process that we created (**terminate process**()) if it is incorrect or is no longer needed.
- Having created new jobs or processes, we need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (**wait time**()).
- More probably, we will want to wait for a specific event to occur (**wait event**()).
- The jobs or processes should then signal when that event has occurred (**signal vent**()). The system calls of Process control are
    - end, abort
    - load, execute
    - create process, terminate process
    - get process attributes, set process attributes
    - wait for time
    - wait event, signal event
    - allocate and free memory

2. **File Management**

- It is needed to create() and delete() files. Either system call requires the name of the file and perhaps some of the file's attributes.
- Once the file is created, we need to open() it and to use it.
- We may also read(), write(), or reposition() (rewind or skip to the end of the file, for example).
- Finally, we need to close() the file, indicating that we are no longer using it.
- To determine the values of various attributes and to reset them if necessary, two system calls, get file attributes() and set file attributes(), are required. File attributes include the file name, file type, protection codes, accounting information, and so on.
- The system calls of File management are
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes

3. Device Management

- The various resources—main memory, disk drives, access to files, and so on, controlled by the operating system can be thought of as devices.
- Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).
- A system with multiple users may require us to first request() a device, to ensure exclusive use of it.
- After we are finished with the device, we release() it.
- We can read(), write(), and (possibly) reposition() the device.
- The system calls of Device management are
    - request device, release device

  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

4. Information Maintenance

- Many system calls exist simply for the purpose of transferring information between the user program and the operating system.
- For example, most systems have a system call to return the current time() and date(). Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.
- Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump() memory. This provision is useful for debugging.
- The operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes() and set process attributes
- The system calls of Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes set process, file, or device attributes

5. Communication

There are two common models of inter process communication:

- message passing model and
- shared-memory model.

Message-passing model

- The communicating processes exchange messages with one another to transfer information.
- Messages can be exchanged between the processes either directly or indirectly through a common mailbox.
- Before communication can take place, a connection must be opened.
- They execute a wait for connection() call and are awakened when a connection is made.
- The source c**lient**, and the receiving daemon **server,** then exchange messages by using read message() and write message() system calls.
- The close connection() call terminates the communication.

Shared-memory model

The processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes.

The system calls of Communications

- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

6. Protection

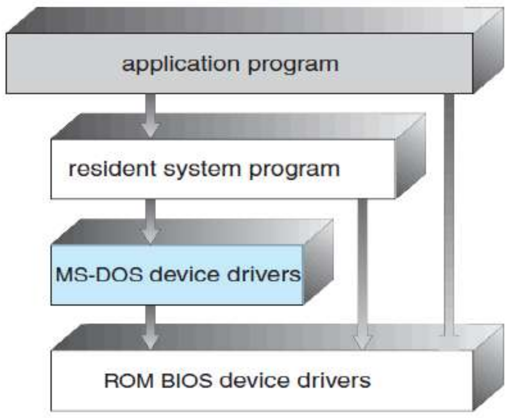
- Protection provides a mechanism for controlling access to the resources provided by a computer system.
- System calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as files and disks.
- The allow user() and deny user() system calls specify whether particular users can—or cannot— be allowed access to certain resources.

OPERATING-SYSTEM STRUCTURE

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

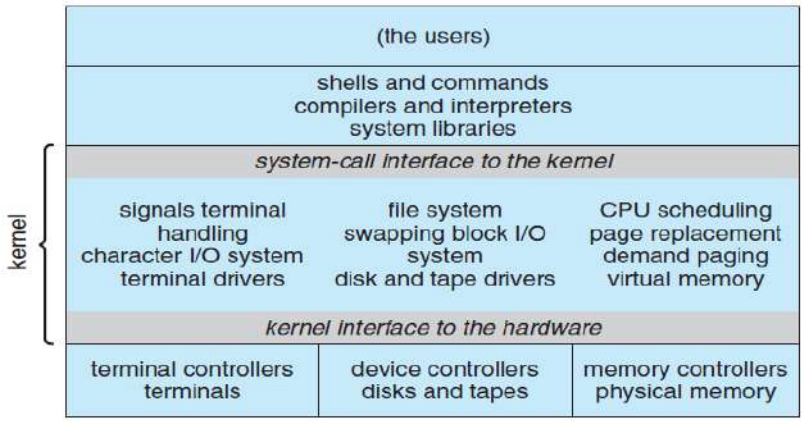
1. Simple Structure

- Many operating systems do not have well-defined structures.
- MS-DOS is an example of such a system. In MS-DOS, the interfaces and levels of functionality are not well separated.
- Application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era.



MS-DOS Layers

- Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality.
- It consists of two separable parts: o Kernel and System programs.
- The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.
- The traditional UNIX operating system is layered to some extent, as shown in the Figure



Traditional UNIX system structure.

- Everything below the system-call interface and above the physical hardware is the kernel.
- The kernel provides the file system, CPU scheduling, memory management, and otheroperating-system functions through system calls.
- Taken in sum, that is an enormous amount of functionality to be combined into one level.
- This monolithic structure was difficult to implement and maintain. It had a distinct
- performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel.

2. Layered Approach

In l**ayered approach**, the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer *N*) is the user interface. This layering structure is depicted in Figure.

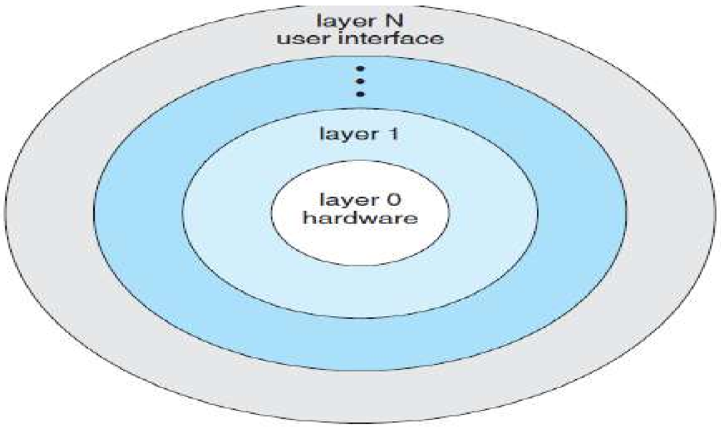


Figure: A layered operating system.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.

- A typical operating-system layer—say, layer *M*—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M*, in turn, can invoke operations on lower-level layers.
- The main advantage of the layered approach is simplicity of construction and debugging.

- The layers are selected so that each uses functions (operations) and services of only lower-
- level layers.
- This approach simplifies debugging and system verification. Each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
- A final problem with layered implementations is that they tend to be less efficient than other types.

3 Microkernels

- The **microkernel** approach structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.
- Microkernels provide minimal process and memory management, in addition to a communication facility. The Figure illustrates the architecture of a typical microkernel.

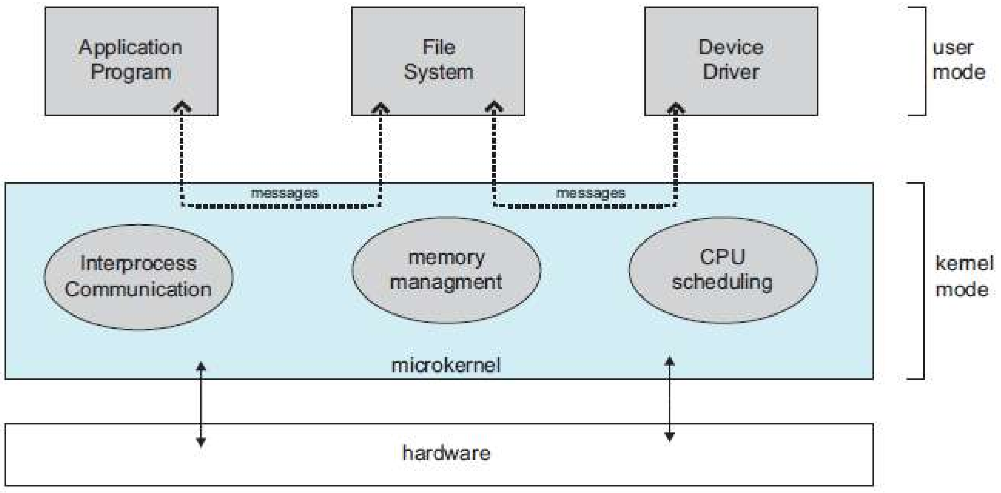


Figure- Architecture of a typical microkernel.

- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
- Communication is provided through **message passing.** For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.
- One benefit of the microkernel approach is that it makes extending the operating system easier.
- All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

4. Modules

- The best current methodology for operating-system design involves using **loadable kernel modules**.
- The kernel has a set of core components and links in additional services via modules, either at boot time or during run time.
- The idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running.
- Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
- Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

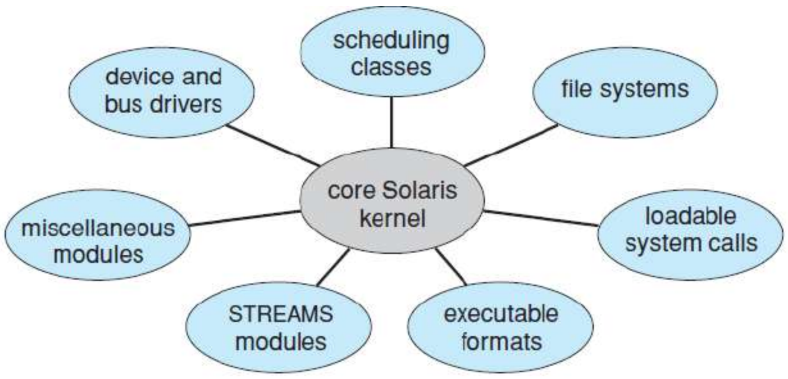


Figure- Solaris loadable modules

The Solaris operating system structure, shown in Figure, is organized around a core kernel with seven types of loadable kernel modules:

- Scheduling classes
- File systems
- Loadable system calls
- Executable formats
- STREAMS modules
- Miscellaneous
- Device and bus drivers

Linux also uses loadable kernel modules, primarily for supporting device drivers and file systems.

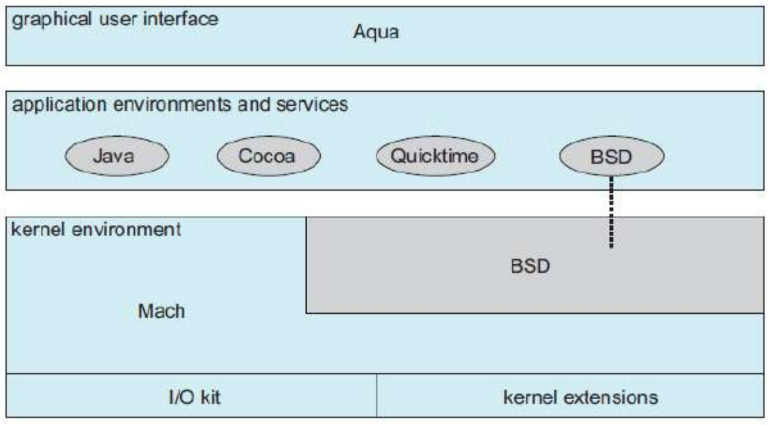
5 Hybrid Systems

Very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. Three hybrid systems are as follows.

- Apple Mac OS X operating system
- Two most prominent mobile operating systems—iOS and Android.

i) Mac OS X

The Apple Mac OS X operating system uses a hybrid structure.



- It is a layered system. The top layers include the ***Aqua*** user interface and a set of application environments and services.
- The **Cocoa** environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications.
- Below these layers is the ***kernel environment***, which consists primarily of the Mach microkernel and the BSD UNIX kernel.
- Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.
- In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules

ii) iOS

- iOS is a mobile operating system designed by Apple to run its smartphone, the ***iPhone***, as well as its tablet computer, the ***iPad***.
- iOS is structured on the MacOS X operating system, with added functionality pertinent to mobile devices,but does not directly run Mac OS X applications.
- **Cocoa Touch** is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices.
- Cocoa Touch provides support for hardware features unique to mobile devices, such as touch screens.
- The **media services** layer provides services for graphics, audio, and video.
- The **core services** layer provides a variety of features, including support for cloud computing and databases.
- The bottom layer represents the core operating system, which is based on the kernel environment
- The structure of iOS appears in Figure.

iii) Android

- The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers.
- Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity.
- Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks for developing mobile applications.
- The structure of Android appears in Figure

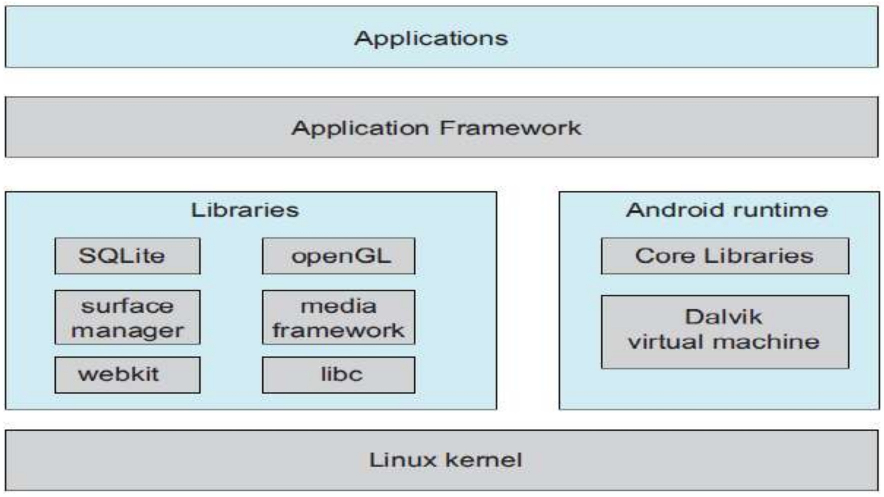


Figure - Architecture of Google's Android

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

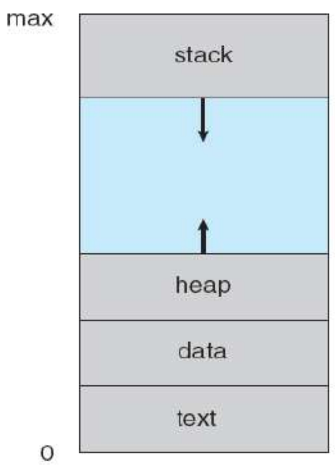
------------------END OF THE UNIT I------------------------

UNIT II

WHAT IS A PROCESS?

A process is a program in execution. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

Process memory is divided into four sections for efficient working:

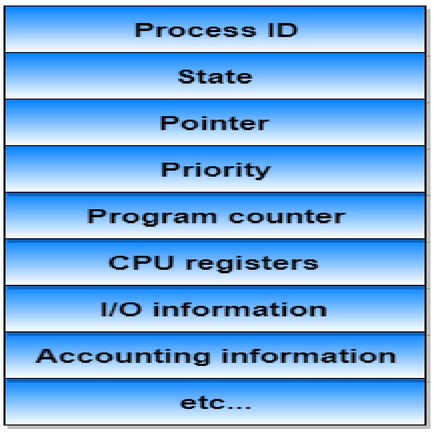


- The **Text section** is made up of the compiled program code, read in from non-volatile storage when the program is launched.
- The **Data section** is made up the global and static variables, allocated and initialized prior to executing the main.
- The **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for local variables when they are declared.

PROCESS CONTROL BLOCK

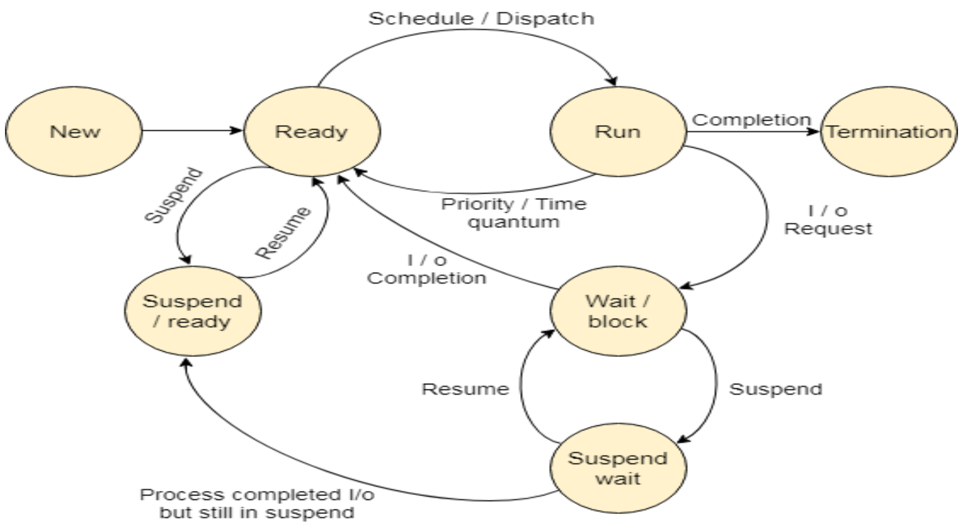
There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:

- **Process State**: It can be running, waiting etc.
- **Process ID** and the **parent process ID**.
- CPU registers and Program Counter. **Program Counter** holds the address of the next instruction to be executed for that process.
- **CPU Scheduling** information: Such as priority information and pointers to scheduling queues.
- **Memory Management information**: For example, page tables or segment tables.
- **Accounting information**: The User and kernel CPU time consumed, account numbers, limits, etc.
- **I/O Status information**: Devices allocated, open file tables, etc.



PROCESS STATES

State Diagram



The process, from its creation to completion, passes through various states. The minimum number of states is five. The names of the states are not standardized although the process may be in one of the following states during execution.

1. New

- A program which is going to be picked up by the OS into the main memory is called a new process.

2. Ready

- Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.
- The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state.

3. Running

- One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system then we can have n processes running simultaneously.

4. Block or wait

- From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process.

- When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

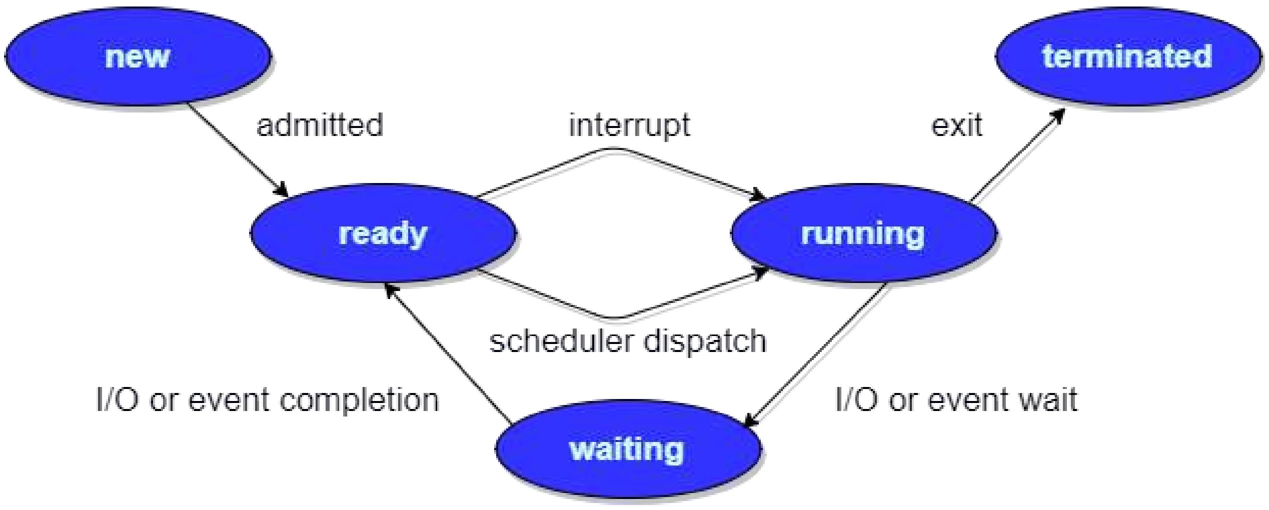
5. **Completion or termination**
   - When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.
6. **Suspend ready**
   - A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.
   - If the main memory is full and a higher priority process comes for the execution then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.
7. **Suspend wait**

- Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.

Different Process States

Processes in the operating system can be in any of the following states:

- NEW- The process is being created.
- READY- The process is waiting to be assigned to a processor.
- RUNNING- Instructions are being executed.
- WAITING- The process is waiting for some event to occur(such as an I/O completion or reception of a signal).
- TERMINATED- The process has finished execution.



WHAT IS PROCESS SCHEDULING?

The act of determining which process is in the **ready** state, and should be moved to the **running** state is known as **Process Scheduling**.

The prime aim of the process scheduling system is to keep the CPU busy all the time and to deliver minimum response time for all programs. For achieving this, the scheduler must apply appropriate rules for swapping processes IN and OUT of CPU.

Scheduling fell into one of the two general categories:

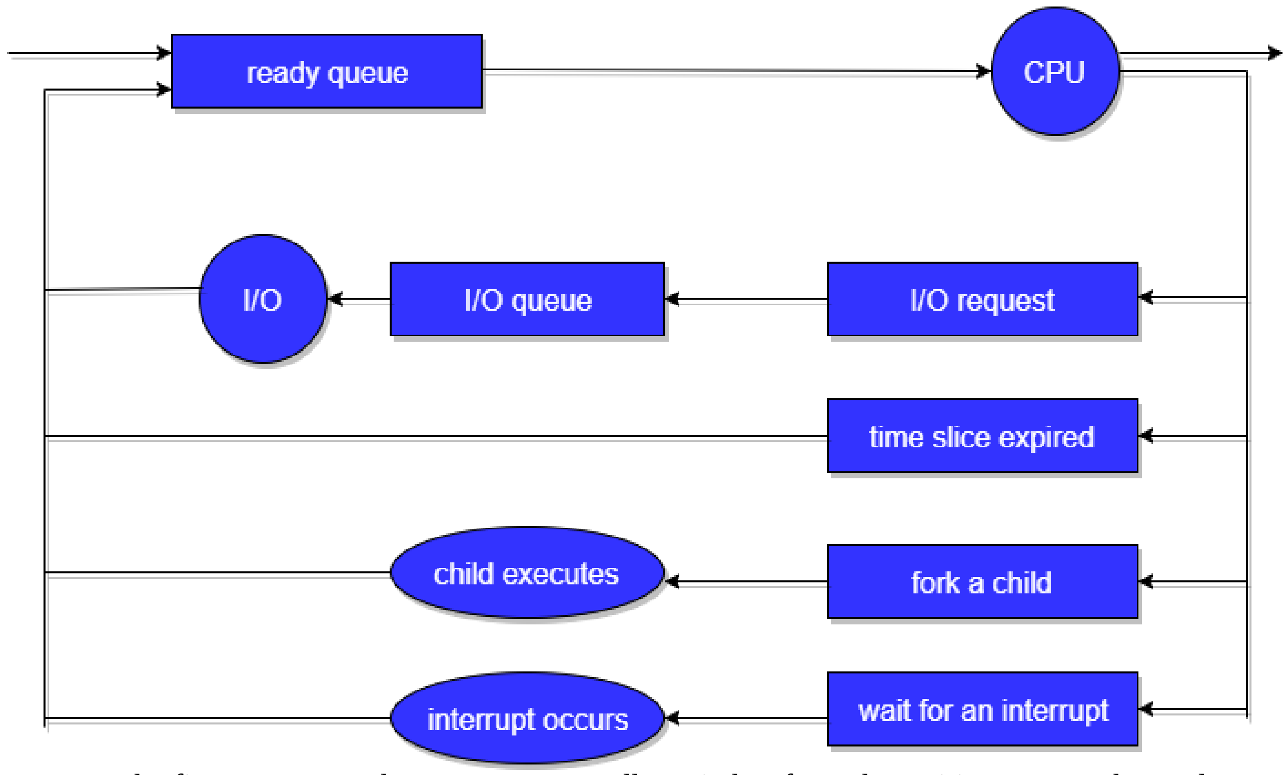
- **Non Pre-emptive Scheduling:** When the currently executing process gives up the CPU voluntarily.
- **Pre-emptive Scheduling:** When the operating system decides to favour another process, pre-empting the currently executing process.

What are Scheduling Queues?

- All processes, upon entering into the system, are stored in the **Job Queue**.
- Processes in the Ready state are placed in the **Ready Queue**.
- Processes waiting for a device to become available are placed in **Device Queues**. There are unique device queues available for each I/O device.

A new process is initially put in the **Ready queue**. It waits in the ready queue until it is selected for execution(or dispatched). Once the process is assigned to the CPU and is executing, one of the following several events can occur:

- The process could issue an I/O request, and then be placed in the **I/O queue**.
- The process could create a new subprocess and wait for its 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.

Types of Schedulers

There are three types of schedulers available:

1. Long Term Scheduler
2. Short Term Scheduler
3. Medium Term Scheduler

Let's discuss about all the different types of Schedulers in detail:

Long Term Scheduler

- Long term scheduler runs less frequently. Long Term Schedulers decide which program must get into the job queue. From the job queue, the Job Processor, selects processes and loads them into the memory for execution. Primary aim of the Job Scheduler is to maintain a good degree of Multiprogramming.

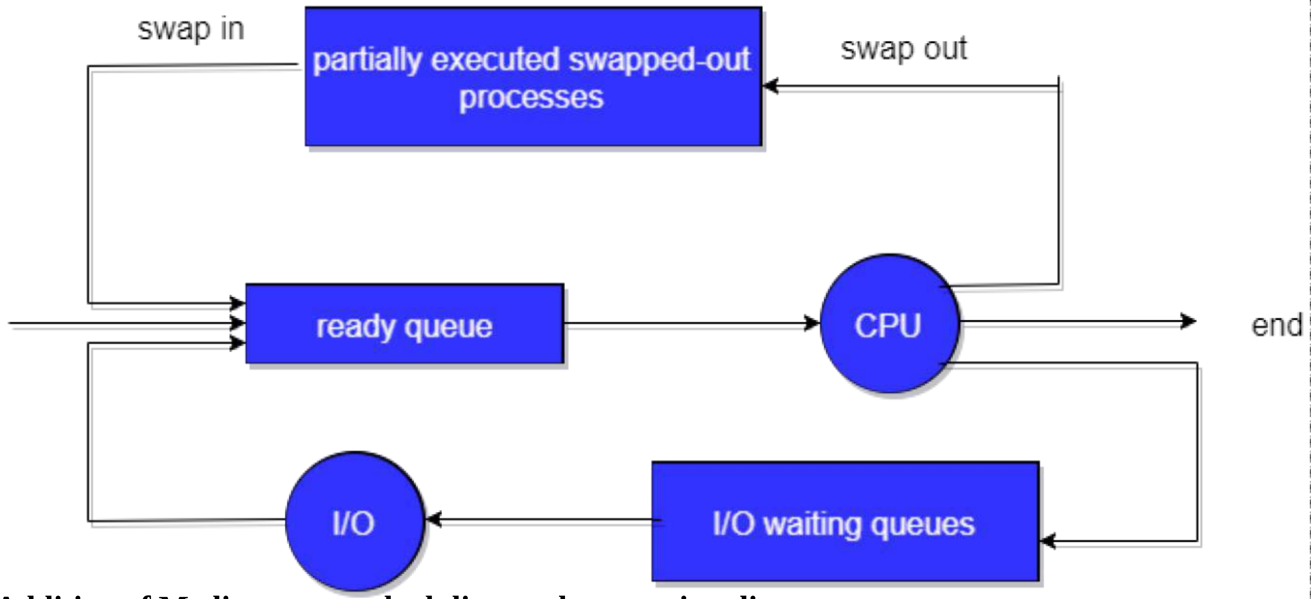
- An optimal degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

Short Term Scheduler

- This is also known as CPU Scheduler and runs very frequently. The primary aim of this scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

- This scheduler removes the processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution van 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. This complete process is descripted in the below diagram:



Addition of Medium-term scheduling to the queueing diagram.

What is Context Switch?

1. Switching the CPU to another process requires **saving** the state of the old process and **loading** the saved state for the new process. This task is known as a **Context Switch**.
2. The **context** of a process is represented in the **Process Control Block(PCB)** of a process; it includes the value of the CPU registers, the process state and memory-management information. 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.
3. Context switch time is **pure overhead**, because the **system does no useful work while switching**. Its 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). Typical speeds range from 1 to 1000 microseconds.

Context Switching has become such a performance **bottleneck** that programmers are using new structures(threads) to avoid it whenever and wherever possible.

OPERATIONS ON PROCESSES

Process Creation

- A process may create several new processes, during the course of execution.

- The creating process is called a parent process, whereas the new processes are called the children of that process.
- When a process creates a new process, two possibilities exist in terms of execution:
    - i. The parent continues to execute concurrently with its children.
    - ii. The parent waits until some or all of its children have terminated.
- There are also two possibilities in terms of the address space of the new process:
    - i. The child process is a duplicate of the parent process.
    - ii. The child process has a program loaded into it.
- In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork system call.

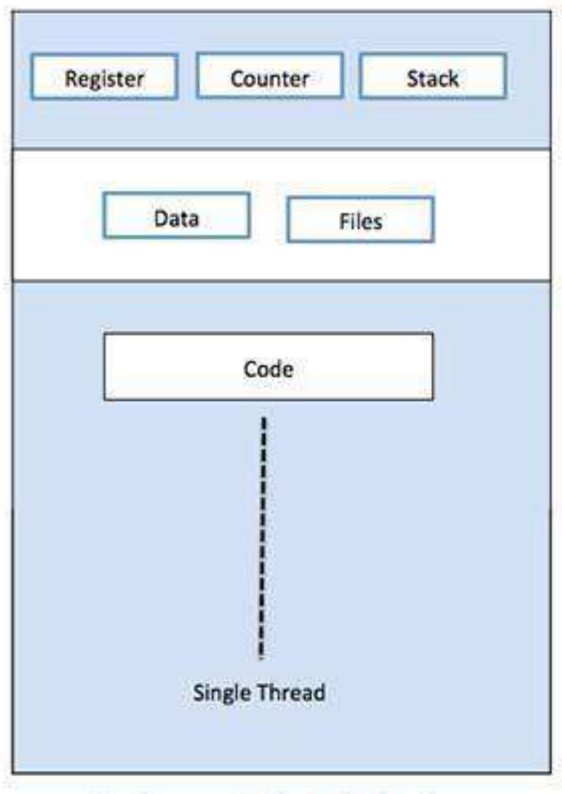
Process Termination

- A process terminates when it finishes executing itsfinal statement and asks the operating system to delete it by using the exit system call.

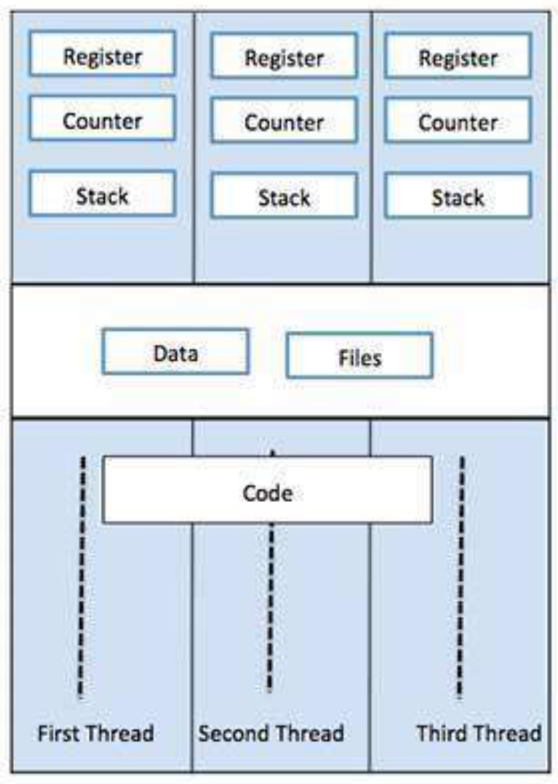
- At that point, the process may return data (output) to its parent process (via the wait system call).
- A process can cause the termination of another process via an appropriate system call.
- A parent may terminate the execution of one of its children for a variety of reasons, such as these:
- The child has exceeded its usage of some of the resources that it has been allocated.
- 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. On 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.

WHAT IS 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.



Single Process P with single thread



Single Process P with three threads

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	One thread can read, write or change another thread's data.

	others.	

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.

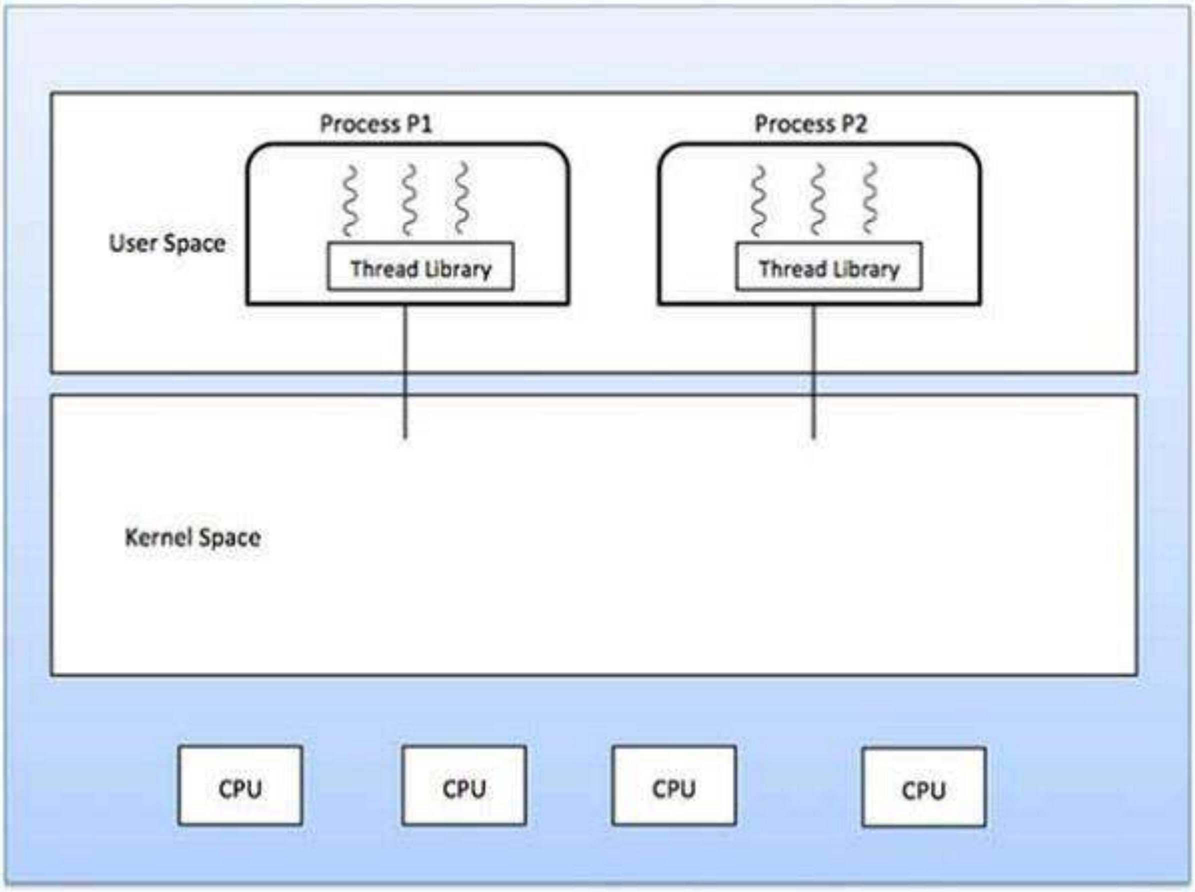
Types of Thread

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.

User Level Threads

- In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

- In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.

Disadvantages

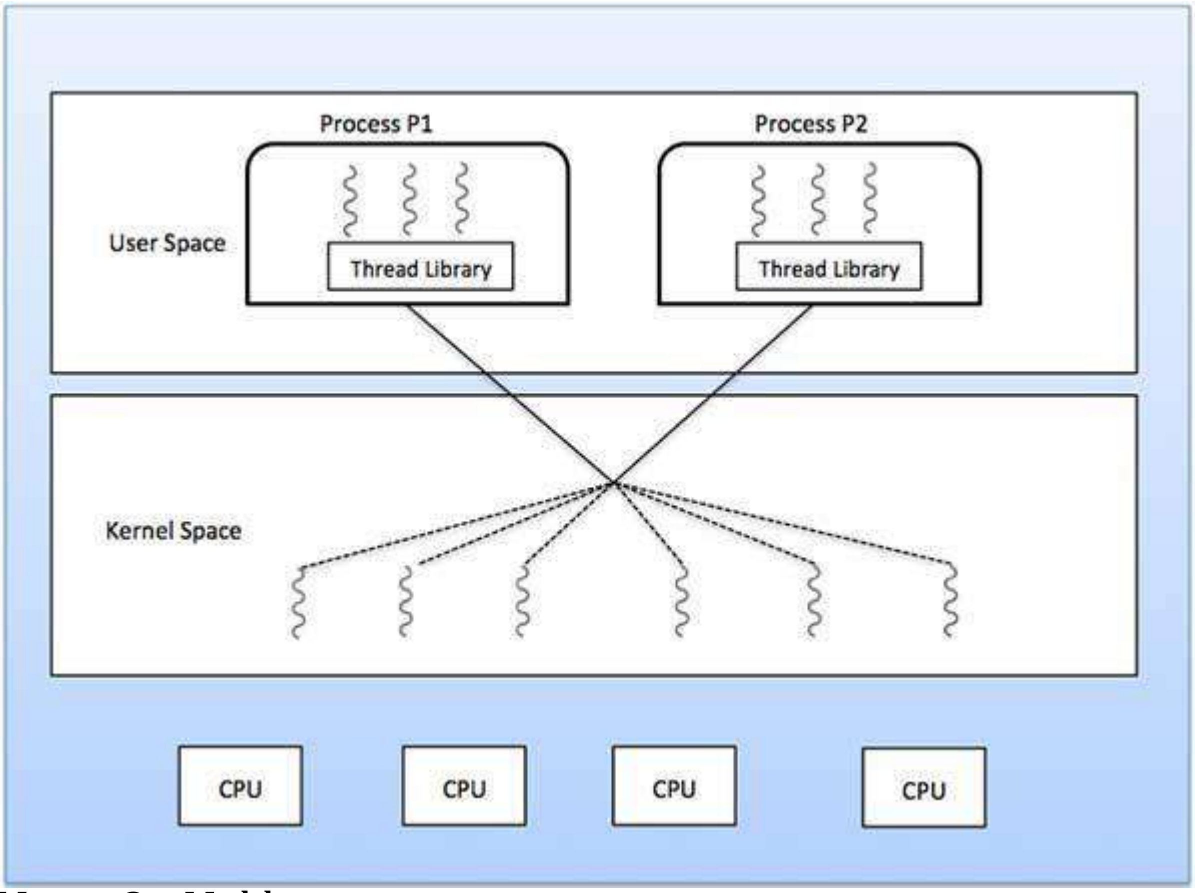
- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

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.

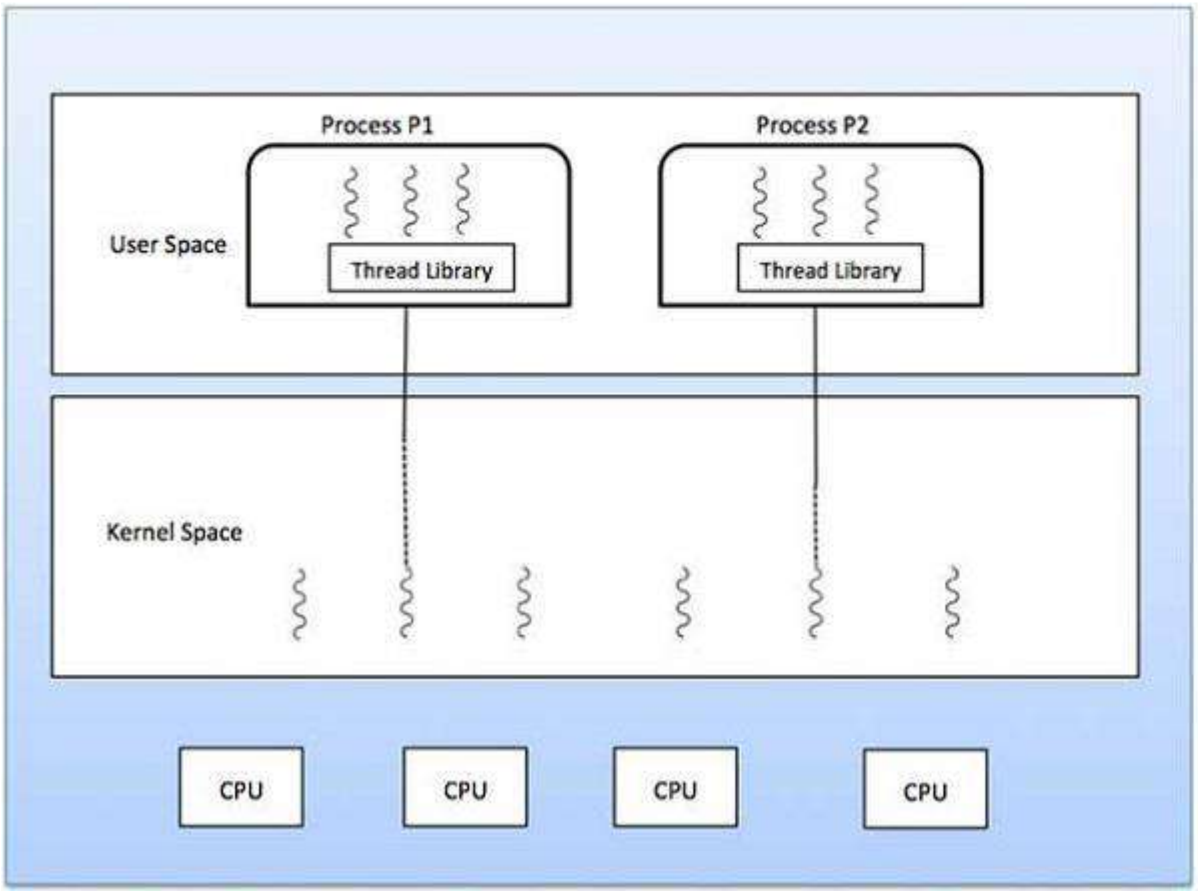
Many to Many Model

- 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.



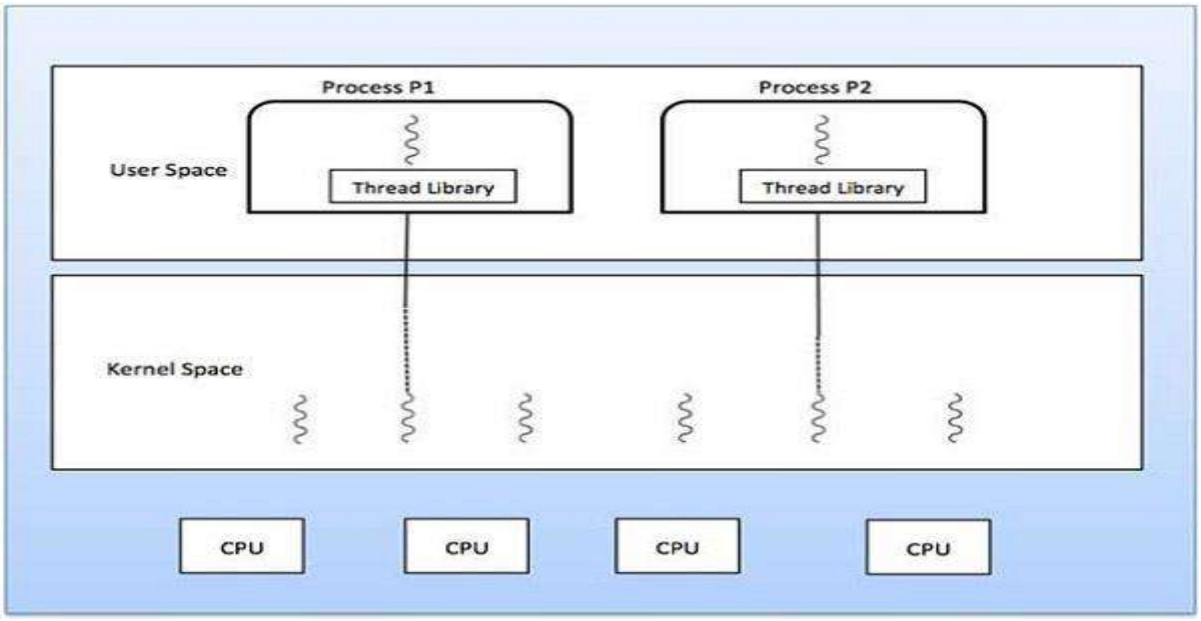
Many to One Model

- 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

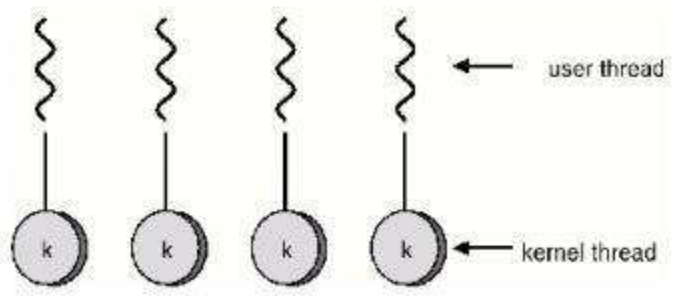
- There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.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

THREAD LIBRARIES



- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.

There are three main thread libraries in use today:

1. **POSIX Pthreads** - may be provided as either a user or kernel library, as an extension to the POSIX standard.
2. **Win32 threads** - provided as a kernel-level library on Windows systems.
3. **Java threads** - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

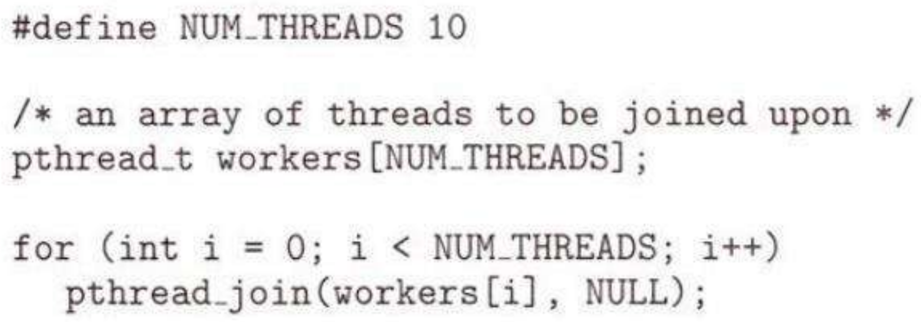
The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

1. Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for pThreads, not the implementation.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.

- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.

- pThreads begin execution in a specified function, in this example the runner( ) function:



```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

2. Java Threads

- ALL Java programs use Threads - even "common" single-threaded ones.
- The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)
- Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)
- Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.
- Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

3. Windows Threads

Similar to pThreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature:

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

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

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

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

      // close the thread handle
      CloseHandle(ThreadHandle);

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

INTERPROCESS COMMUNICATION

- Operating systems provide the means for cooperating processes to communicate with each other via an interprocess communication (PC) facility.
- IPC provides a mechanism to allow processes to communicate and to synchronize their actions.IPC is best provided by a message passing system.

Basic Structure:

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.

Physical implementation of the link is done through a hardware bus , network etc, There are several methods for logically implementing a link and the operations:

1. Direct or indirect communication
2. Symmetric or asymmetric communication
3. Automatic or explicit buffering
4. Send by copy or send by reference
5. Fixed-sized or variable-sized messages

1. Naming

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

1.1. Direct Communication

Each process that wants to communicate must explicitly name the recipient or sender of the communication.
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.
- Exactly one link exists between each pair of processes.

There are two ways of addressing namely

♣ Symmetry in addressing

♣ Asymmetry in addressing

In symmetry in addressing, the send and receive primitives are defined as:

♣ send(P, message) Send a message to process P
♣ receive(Q, message) Receive a message from Q

In asymmetry in addressing , the send & receive primitives are defined as:

♣ send (p, message) send a message to process p
♣ receive(id, message) receive message from any process,

id is set to the name of the process with which communication has taken place

1.2. Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or ports.
The send and receive primitives are defined as follows:

♣ send (A, message) Send a message to mailbox A.
♣ receive (A, message) Receive a message from mailbox A.

A communication link has the following properties:

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

1.3. Buffering

A link has some capacity that determines the number of message that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link.

There are three ways that such a queue can be implemented.

1. **Zero capacity:** Queue length of maximum is 0. No message is waiting in a queue. The sender must wait until the recipient receives the message. (Message system with no buffering)
2. **Bounded capacity:** The queue has finite length n. Thus at most n messages can reside in it.
3. **Unbounded capacity:** The queue has potentially infinite length. Thus any number of messages can wait in it. The sender is never delayed.

1.4 Synchronization

Message passing may be either blocking or non-blocking.

1. **Blocking Send** - The sender blocks itself till the message sent by it is received by the receiver.
2. **Non-blocking Send** - The sender does not block itself after sending the message but continues with its normal operation.
3. **Blocking Receive** - The receiver blocks itself until it receives the message.

4. **Non-blocking Receive –** The receiver does not block itself.

There are two levels of communication

- Low – level form of communication – eg. Socket
- High – level form of communication – eg.RPC , RMI

PROCESS SYNCHRONIZATION

1. Background

Producer code

```
item nextProduced;
while( true )   {
            /* Produce an item and store it in nextProduced */ nextProduced =
            makeNewItem( . . . );
            /* Wait for space to become available */ while( ( ( in + 1 ) %
            BUFFER_SIZE ) == out )
                ; /* Do nothing */
            /* And then store the item and repeat the loop. */ buffer[ in ] =
            nextProduced;
            in = ( in + 1 ) % BUFFER_SIZE;

        }
```

Consumer code

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

      }
```

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

Producer Process:

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

Consumer Process:

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

- Unfortunately we have now introduced a new problem, because both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a race condition. In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first, and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process. (Bank balance example discussed in class.)
- The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.
- But, you might say, "Each of those are single instructions - How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are actually three steps each at the hardware level:
  (1)Fetch counter from memory into a register,
  (2)increment or decrement the register, and
  (3)Store the new value of counter back to memory. If the instructions from the two processes get interleaved, there could be serious problems, such as illustrated by the following:

Producer:

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

Consumer:

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

Interleaving:

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

CRITICAL-SECTION PROBLEM

The producer-consumer problem described above is a specific example of a more general situation known as the ***critical section*** problem. The general idea is that in a number of cooperating processes, each has a critical section of code, with the following conditions and terminologies:

- Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
- The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a carefully controlled locking door.
- The code following the critical section is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
- The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

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

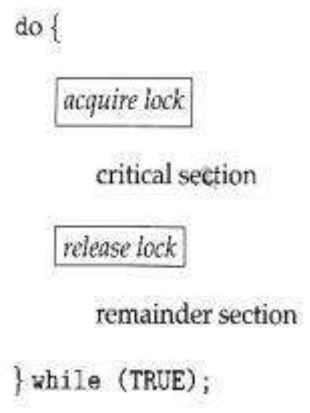
1. **Mutual Exclusion -** Only one process at a time can be executing in their critical section.

2. **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (i.e. processes cannot be blocked forever waiting to get into their critical sections.)
3. **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)

- We assume that all processes proceed at a non-zero speed, but no assumptions can be made regarding the ***relative*** speed of one process versus another.
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms:
    - **Non-preemptive kern**els do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel-mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.
    - **Preemptive kernels** allow for real-time operations, but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

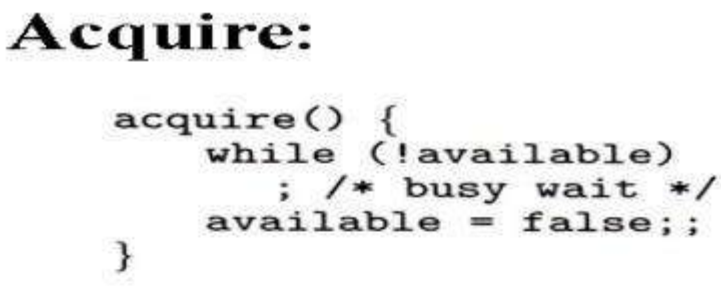
Mutex Locks

- 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.



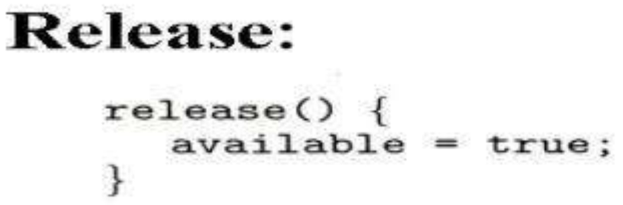
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

- 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;;
}
```



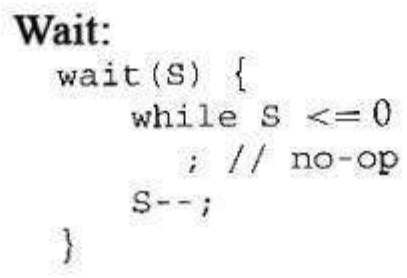
Release:

```
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.

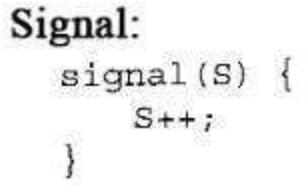
SEMAPHORES

- A more robust alternative to simple mutexes is to use ***semaphores***, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.
- Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It IS okay, however,forthebusylooptobe interruptedwhenthetesistrue,which prevents the system from hanging forever.



Wait:

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



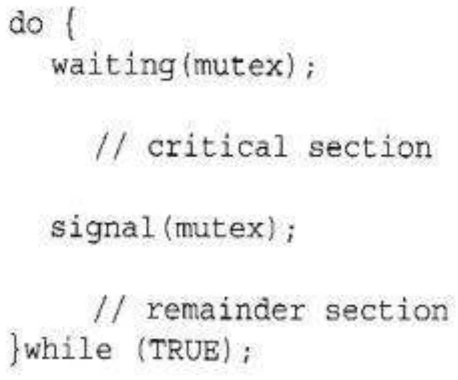
Signal:

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

1. Semaphore Usage

In practice, semaphores can take on one of two forms:

- **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism.. The use of mutexes for this purpose is shown in Figure below.



```
do {
  waiting(mutex);

     // critical section

  signal(mutex);

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

Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.

First we create a semaphore named synch that is shared by the two processes, and initialize it to zero.

Then in process P1 we insert the code:

```
S1;

signal( synch );
```

and in process P2 we insert the code:

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

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

2. Semaphore Implementation

The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.

An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.)

The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

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

Wait Operation:

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

Signal Operation:

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

3. Deadlocks and Starvation

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

```
      P0                P1

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

Another problem to consider is that of starvation, in which one or more processes get blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call.

If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

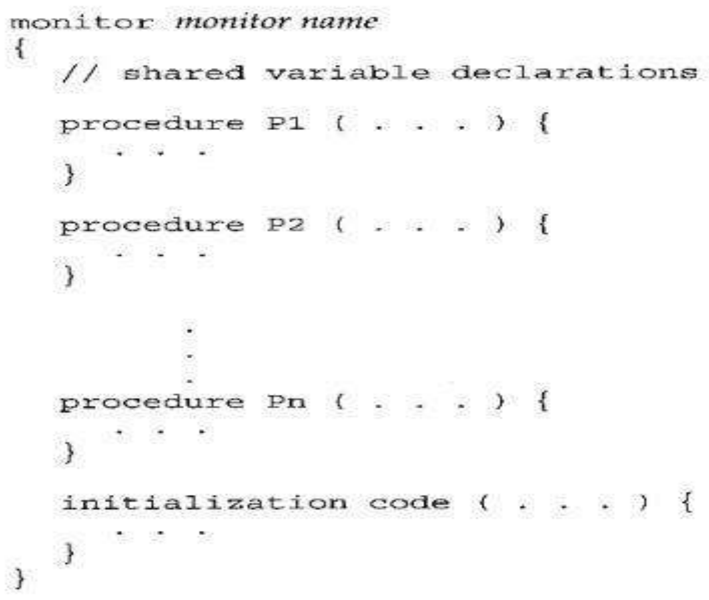
MONITORS

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.)

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

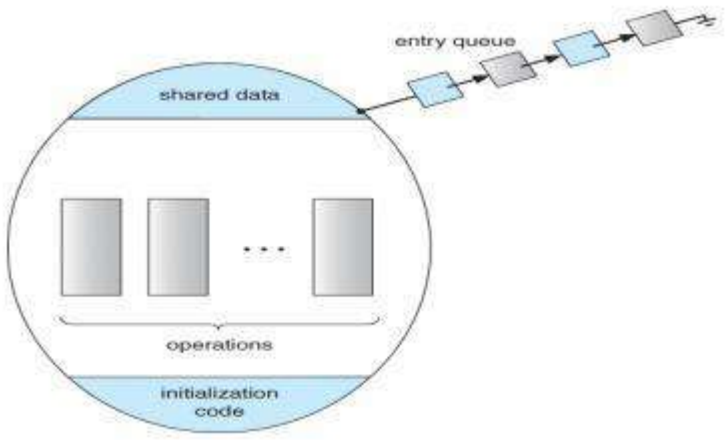
Monitor Usage

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



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

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

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

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

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

- 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.)



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.

Implementing a Monitor Using Semaphores

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

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

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

-------------------- END OF THE UNIT II--------------------

CPU SCHEDULING

- ✓ CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.
- ✓ 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 from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

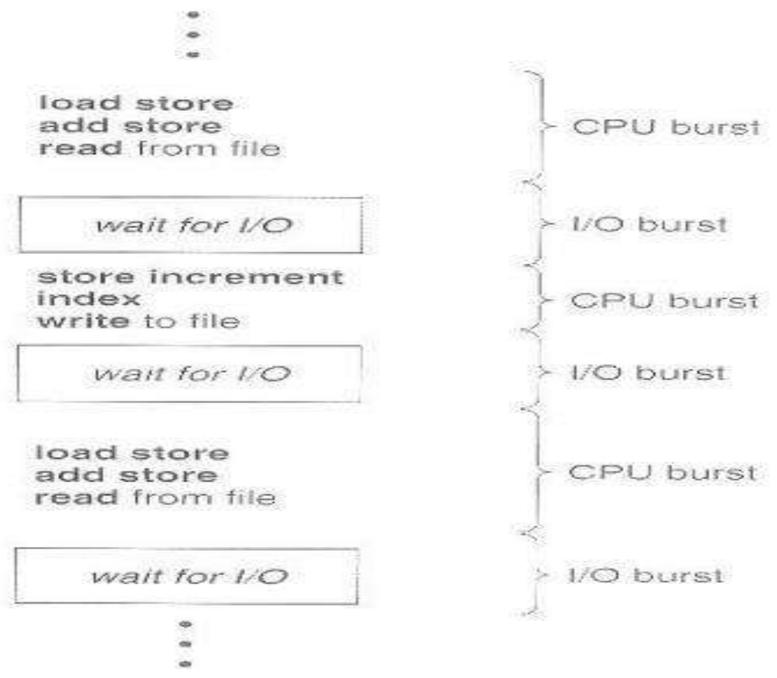
Basic Concepts

Almost all programs have some alternating cycle of CPU number crunching and waiting for I/O of some kind. (Even a simple fetch from memory takes a long time relative to CPU speeds.)

- ✓ In a simple system running a single process, the time spent waiting for I/O is wasted, and those CPU cycles are lost forever.
- ✓ A scheduling system allows one process to use the CPU while another is waiting for I/O, thereby making full use of otherwise lost CPU cycles.
- ✓ The challenge is to make the overall system as "efficient" and "fair" as possible, subject to varying and often dynamic conditions, and where "efficient" and "fair" are somewhat subjective terms, often subject to shifting priority policies.

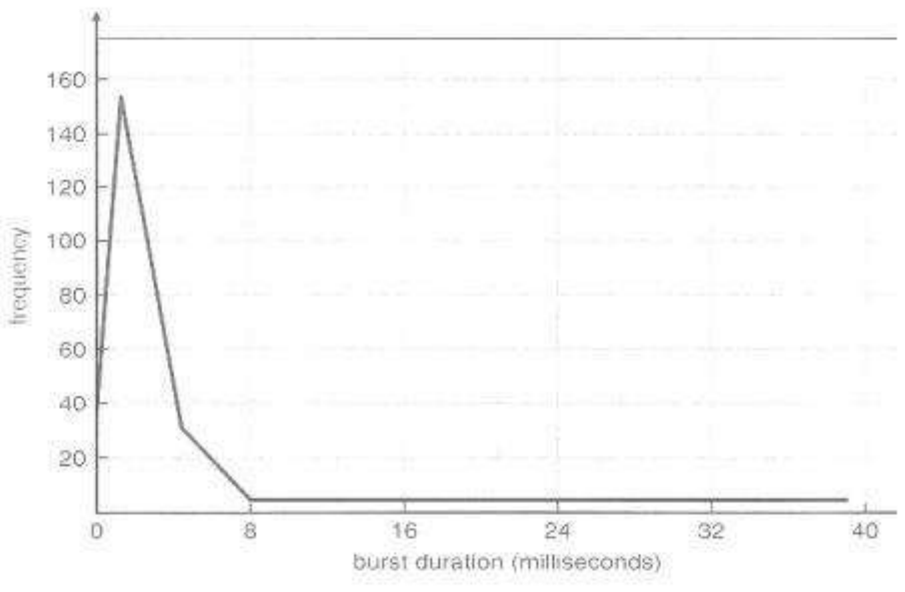
CPU-I/O Burst Cycle

Almost all processes alternate between two states in a continuing ***cycle***, as shown in Figure below:



- ✓ A CPU burst of performing calculations, and An I/O burst, waiting for data transfer in or out of the system.

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



CPU Scheduler

- ✓ Whenever the CPU becomes idle, it is the job of the CPU Scheduler (a.k.a. the short-term scheduler) to select another process from the ready queue to run next.
- ✓ The storage structure for the ready queue and the algorithm used to select the next process are not necessarily a FIFO queue. There are several alternatives to choose from, as well as numerous adjustable parameters for each algorithm.

Preemptive Scheduling

CPU scheduling decisions take place under one of four conditions:

1. When a process switches from the running state to the waiting state, such as for an I/O request or invocation of the wait( ) system call.
2. When a process switches from the running state to the ready state, for example in response to an interrupt.
3. When a process switches from the waiting state to the ready state, say at completion of I/O or a return from wait( ).
4. When a process terminates.

- ✓ For conditions 1 and 4 there is no choice - A new process must be selected.
- ✓ For conditions 2 and 3 there is a choice - To either continue running the current process, or select a different one.
- ✓ If scheduling takes place only under conditions 1 and 4, the system is said to be ***non-preemptive***, or ***cooperative***. Under these conditions, once a process starts running it keeps running, until it either voluntarily blocks or until it finishes. Otherwise the system is said to be ***preemptive.***
- ✓ Windows used non-preemptive scheduling up to Windows 3.x, and started using pre-emptive scheduling with Win95. Macs used non-preemptive prior to OSX, and pre-emptive since then. Note that pre-emptive scheduling is only possible on hardware that supports a timer interrupt.

- ✓ Note that pre-emptive scheduling can cause problems when two processes share data, because one process may get interrupted in the middle of updating shared data structures.
- ✓ Preemption can also be a problem if the kernel is busy implementing a system call (e.g. updating critical kernel data structures) when the preemption occurs. Most modern UNIX deal with this problem by making the process wait until the system call has either completed or blocked before allowing the preemption Unfortunately this solution is problematic for real-time systems, as real-time response can no longer be guaranteed.
- ✓ Some critical sections of code protect themselves from concurrency problems by disabling interrupts before entering the critical section and re-enabling interrupts on exiting the section. Needless to say, this should only be done in rare situations, and only on very short pieces of code that will finish quickly, (usually just a few machine instructions.)

Dispatcher

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

Scheduling Algorithms

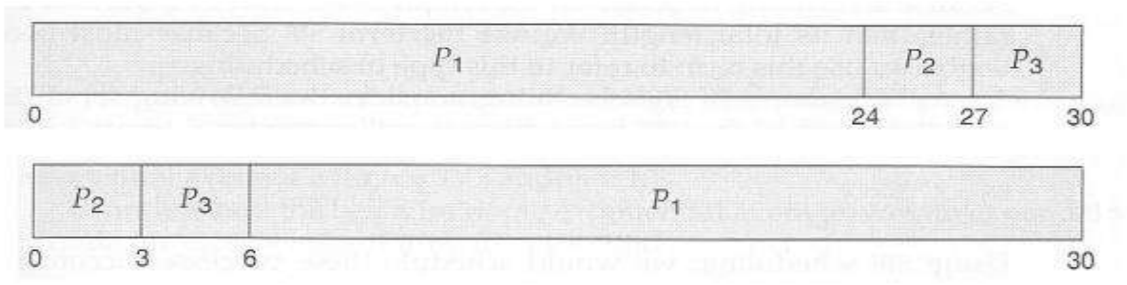
- ✓ The following subsections will explain several common scheduling strategies, looking at only a single CPU burst each for a small number of processes. Obviously real systems have to deal with a lot more simultaneous processes executing their CPU-I/O burst cycles. There are different types of scheduling algorithm
- ✓ **First-Come First-Serve Scheduling,(FCFS)**
- ✓ **Shortest-Job-First Scheduling, (SJF)**
- ✓ **Priority Scheduling (PS)**
- ✓ **Round Robin Scheduling (RRS)**
- ✓ **Multilevel Queue Scheduling (MQS)**
- ✓ **Multilevel Feedback-Queue Scheduling (MFQS)**

First-Come First-Serve Scheduling, FCFS

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

Process	Burst Time
P1	24
P2	3
P3	3

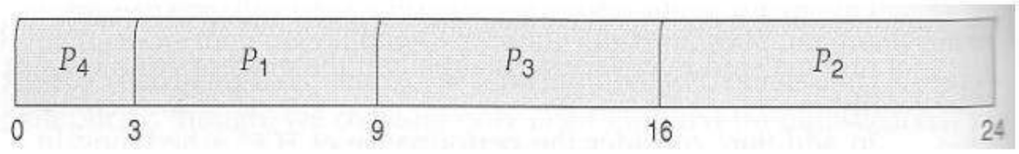
- In the first Gantt chart below, process P1 arrives first. The average waiting time for the three processes is (0 + 24 + 27) / 3 = 17.0 ms.
- In the second Gantt chart below, the same three processes have an average wait time of (0 + 3 + 6) / 3 = 3.0 ms. The total run time for the three bursts is the same, but in the second case two of the three finish much quicker, and the other process is only delayed by a short amount.



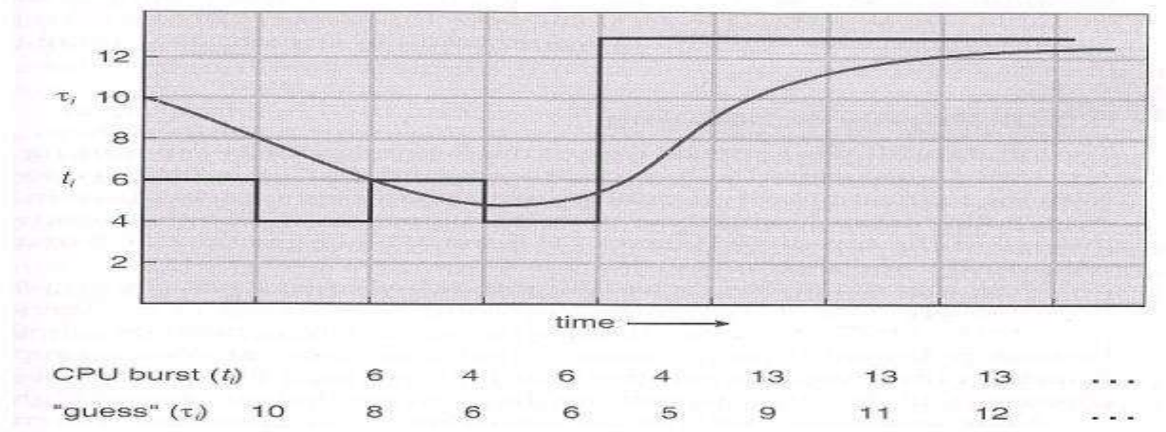
Shortest-Job-First Scheduling, SJF

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

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

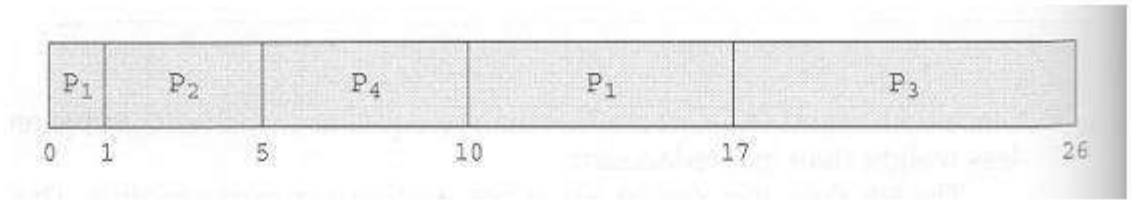


In the case above the average wait time is (0 + 3 + 9 + 16) / 4 = 7.0 ms, (as opposed to 10.25 ms for FCFS for the same processes.)



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

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



- ✓ The average wait time in this case is

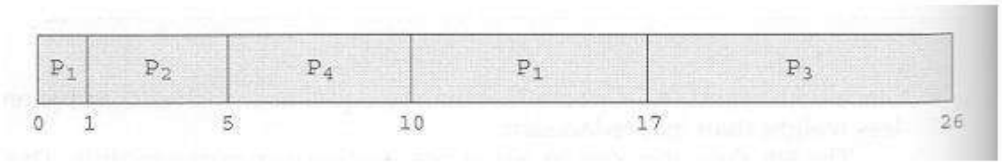
  ((5 - 3) + (10 - 1) + (17 - 2)) / 4 = 26 / 4 = 6.5 ms.

  (As opposed to 7.75 ms for non-preemptive SJF or 8.75 for FCFS.)

Priority Scheduling

- ✓ Priority scheduling is a more general case of SJF, in which each job is assigned a priority and the job with the highest priority gets scheduled first. (SJF uses the inverse of the next expected burst time as its priority - The smaller the expected burst, the higher the priority.)
- ✓ Note that in practice, priorities are implemented using integers within a fixed range, but there is no agreed-upon convention as to whether "high" priorities use large numbers or small numbers.
- ✓ For example, the following Gantt chart is based upon these process burst times and priorities, and yields an average waiting time of 8.2 ms:

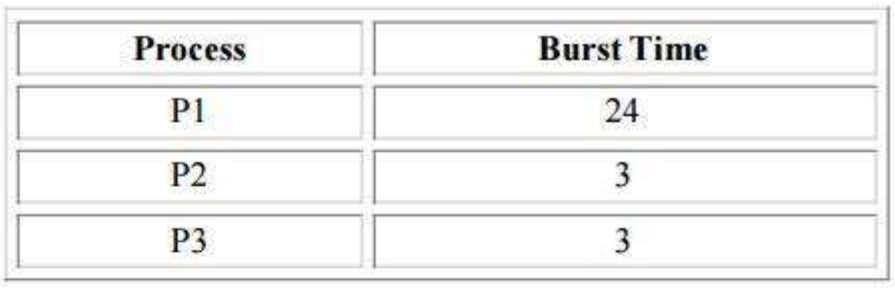
Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
p4	3	5



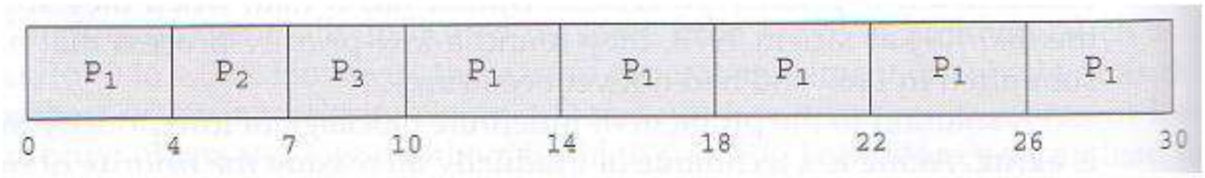
- ✓ Priorities can be assigned either internally or externally. Internal priorities are assigned by the OS using criteria such as average burst time, ratio of CPU to I/O activity, system resource use, and other factors available to the kernel. External priorities are assigned by users, based on the importance of the job, fees paid, politics, etc.

Round Robin Scheduling

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



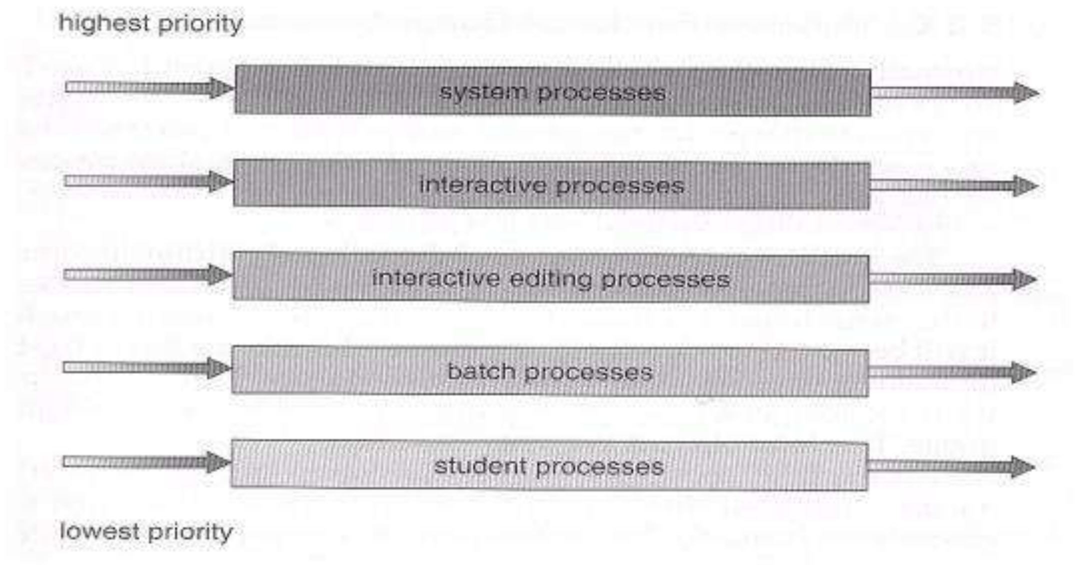
Process	Burst Time
P1	24
P2	3
P3	3



- ✓ The performance of RR is sensitive to the time quantum selected. If the quantum is large enough, then RR reduces to the FCFS algorithm; If it is very small, then each process gets 1/nth of the processor time and share the CPU equally.
- ✓ **BUT**, a real system invokes overhead for every context switch, and the smaller the time quantum the more context switches there are. (See Figure 5.4 below.) Most modern systems use time quantum between 10 and 100 milliseconds, and context switch times on the order of 10 microseconds, so the overhead is small relative to the time quantum.
- ✓ In general, turnaround time is minimized if most processes finish their next cpu burst within one time quantum. For example, with three processes of 10 ms bursts each, the average turnaround time for 1 ms quantum is 29, and for 10 ms quantum it reduces to 20.
- ✓ However, if it is made too large, then RR just degenerates to FCFS. A rule of thumb is that 80% of CPU bursts should be smaller than the time quantum.

Multilevel Queue Scheduling

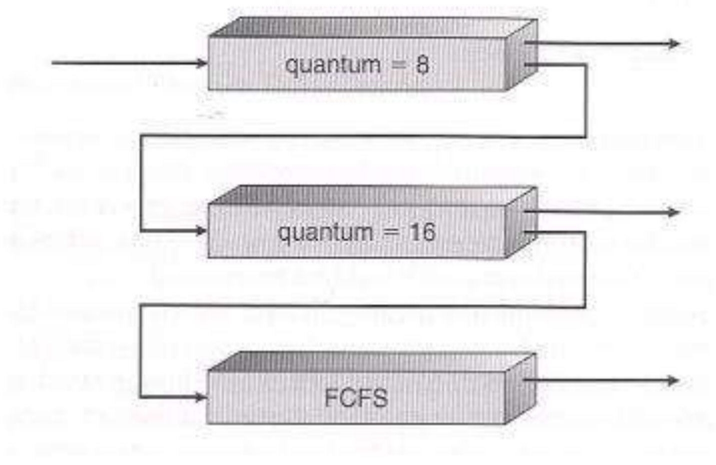
- ✓ When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.
- ✓ Scheduling must also be done between queues, that is scheduling one queue to get time relative to other queues. Two common options are strict priority (no job in a lower priority queue runs until all higher priority queues are empty) and round-robin (each queue gets a time slice in turn, possibly of different sizes.)
- ✓ Note that under this algorithm jobs cannot switch from queue to queue - Once they are assigned a queue, that is their queue until they finish



Multilevel Feedback-Queue Scheduling

- ✓ Multilevel feedback queue scheduling is similar to the ordinary multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons:
  - If the characteristics of a job change between CPU-intensive and I/O intensive, then it may be appropriate to switch a job from one queue to another.
  - Aging can also be incorporated, so that a job that has waited for a long time can get bumped up into a higher priority queue for a while.

- ✓ Some of the parameters which define one of these systems include:
  - The number of queues.
  - The scheduling algorithm for each queue.
  - The methods used to upgrade or demote processes from one queue to another. (Which may be different.)
  - The method used to determine which queue a process enters initially.



MEMORY MANAGEMENT

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution. Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

The basic concepts related to Memory Management.

Process Address Space

- ✓ The process address space is the set of logical addresses that a process references in its code. For example, when 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^31 possible numbers, for a total theoretical size of 2 gigabytes.
- ✓ The operating system takes care of mapping the logical addresses to physical addresses at the time of memory allocation to the program. There are three types of addresses used in a program before and after memory is allocated −

S.N.	Memory Addresses & Description
1	**Symbolic addresses** The addresses used in a source code. The variable names, constants, and instruction labels are the basic elements of the symbolic address space.
2	**Relative addresses** At the time of compilation, a compiler converts symbolic addresses into relative addresses.
3	**Physical addresses** The loader generates these addresses at the time when a program is loaded into main memory.

- ✓ Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.
- ✓ The set of all logical addresses generated by a program is referred to as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is referred to as a **physical address space.**
- ✓ The runtime mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.
    - ▪ The value in the base register is added to every address generated by a user process, which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
    - ▪ The user program deals with virtual addresses; it never sees the real physical addresses.

Static vs Dynamic Loading

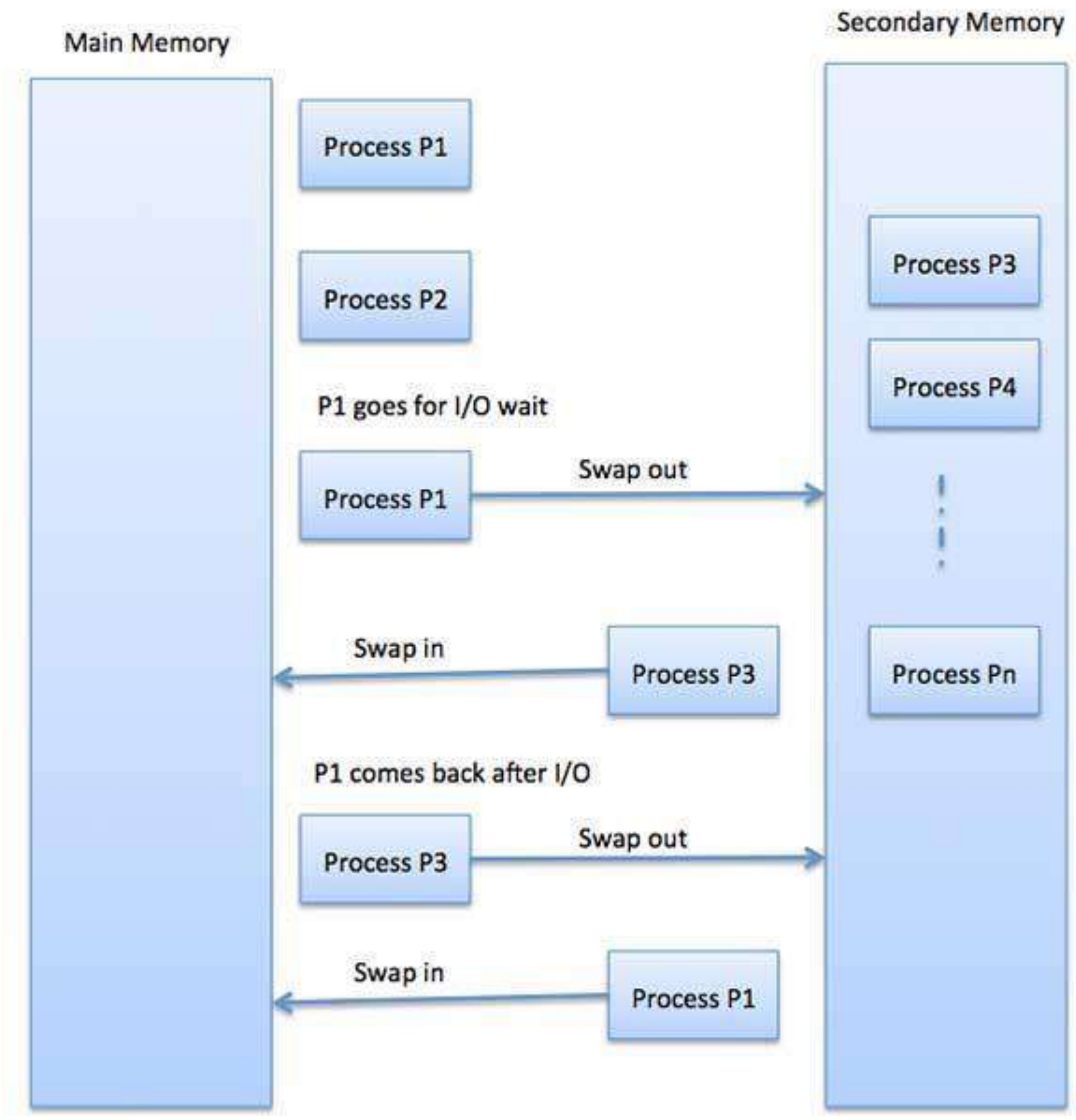
- ✓ The choice between Static or Dynamic Loading is to be made at the time of computer program being developed. If you have to load your program statically, then at the time of compilation, the complete programs will be compiled and linked without leaving any external program or module dependency. The linker combines the object program with other necessary object modules into an absolute program, which also includes logical addresses.
- ✓ If you are writing a Dynamically loaded program, then your compiler will compile the program and for all the modules which you want to include dynamically, only references will be provided and rest of the work will be done at the time of execution.
- ✓ At the time of loading, with **static loading**, the absolute program (and data) is loaded into memory in order for execution to start.
- ✓ If you are using **dynamic loading**, dynamic routines of the library are stored on a disk in relocatable form and are loaded into memory only when they are needed by the program.

Static vs Dynamic Linking

- ✓ As explained above, when static linking is used, the linker combines all other modules needed by a program into a single executable program to avoid any runtime dependency.
- ✓ When dynamic linking is used, it is not required to link the actual module or library with the program, rather a reference to the dynamic module is provided at the time of compilation and linking. Dynamic Link Libraries (DLL) in Windows and Shared Objects in Unix are good examples of dynamic libraries.

Swapping

- ✓ Swapping is a mechanism in which a process can be swapped temporarily out of main memory (or move) to secondary storage (disk) and make that memory available to other processes. At some later time, the system swaps back the process from the secondary storage to main memory.
- ✓ Though performance is usually affected by swapping process but it helps in running multiple and big processes in parallel and that's the reason **Swapping is also known as a technique for memory compaction**.



- ✓ The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory.
- ✓ Let us assume that the user process is of size 2048KB and on a standard hard disk where swapping will take place has a data transfer rate around 1 MB per second. The actual transfer of the 1000K process to or from memory will take

  2048KB / 1024KB per second
  = 2 seconds
  = 2000 milliseconds

- ✓ Now considering in and out time, it will take complete 4000 milliseconds plus other overhead where the process competes to regain main memory.

Memory Allocation

Main memory usually has two partitions −

- **Low Memory** − Operating system resides in this memory.

- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

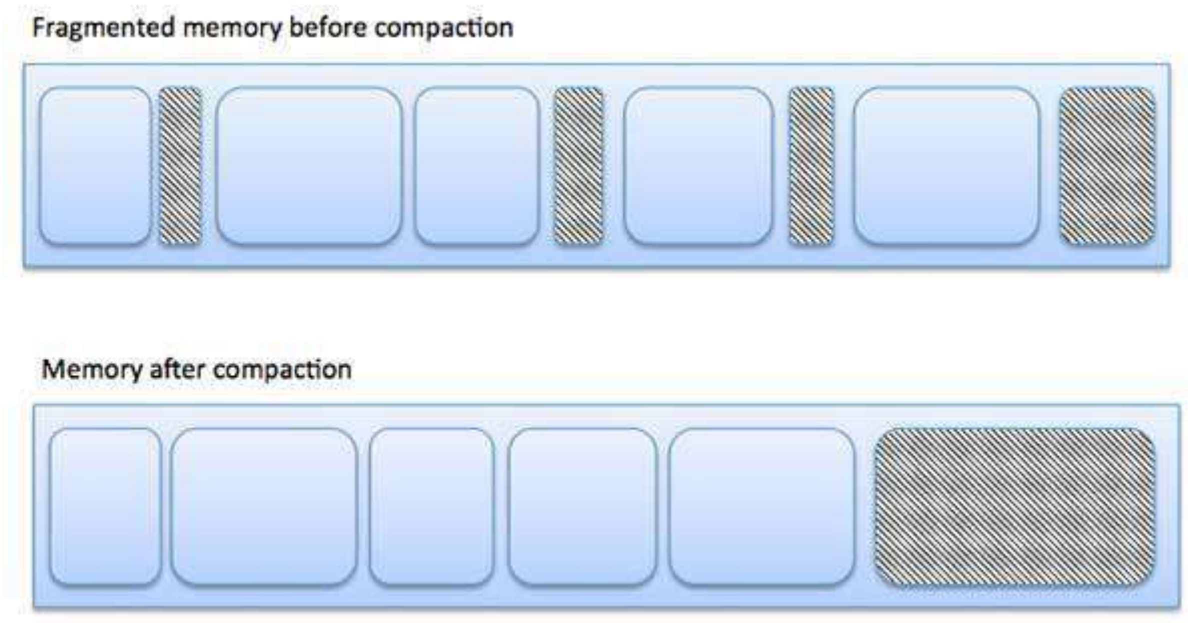
FRAGMENTATION

✓ As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

S.N.	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

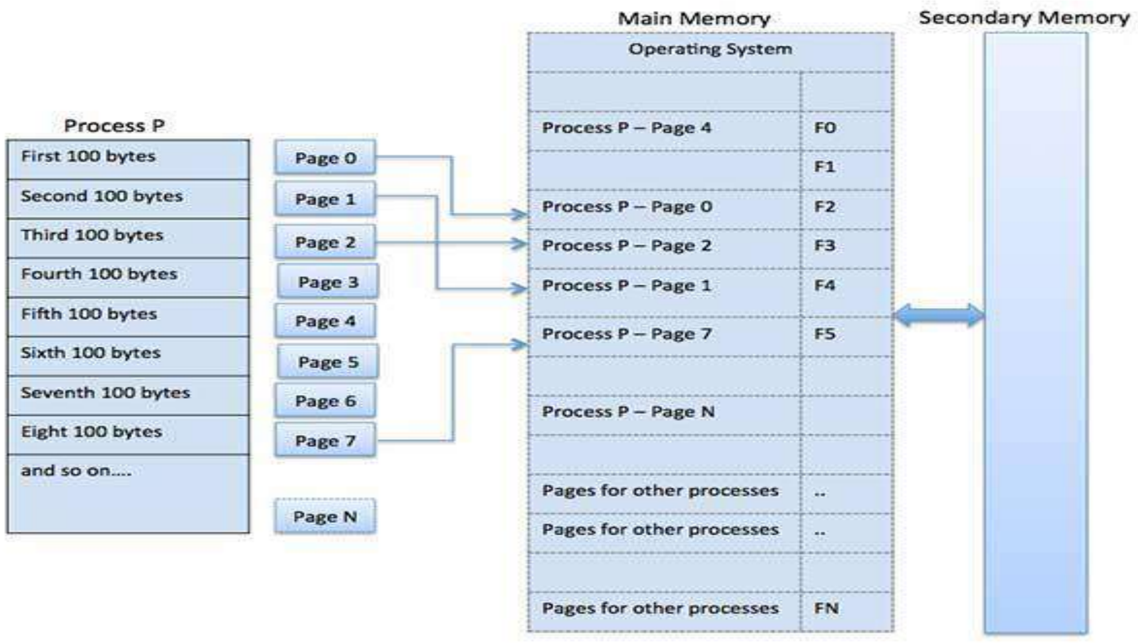
The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −



- ✓ External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.
- ✓ The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

PAGING

- ✓ A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.
- ✓ Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.
- ✓ Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

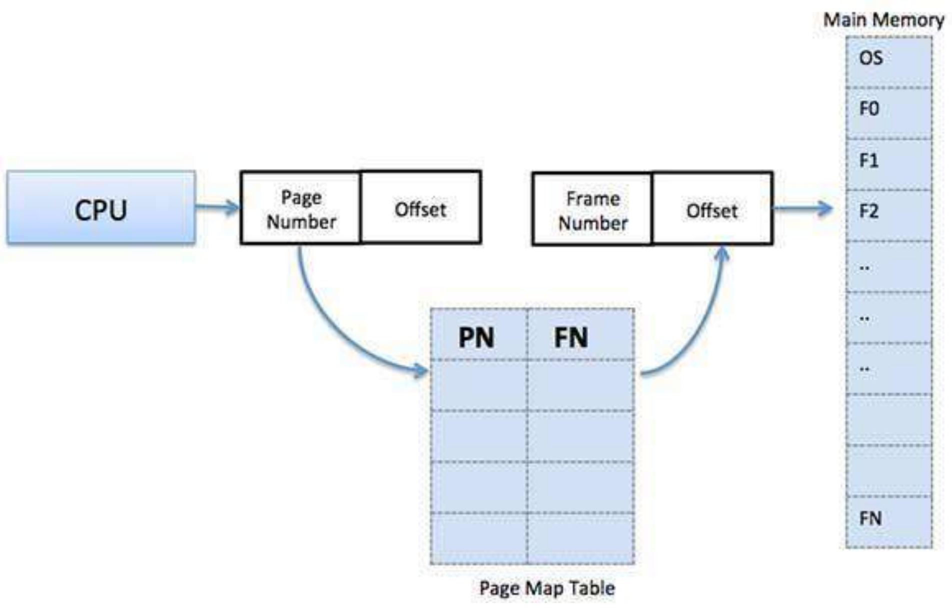


Address Translation

- ✓ Page address is called **logical address** and represented by **page number** and the **offset**.

  Logical Address = Page number + page offset
- ✓ Frame address is called **physical address** and represented by a **frame number** and the **offset**.

  Physical Address = Frame number + page offset
- ✓ A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



- ✓ When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.
- ✓ When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out

of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

✓ This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging −

- Paging reduces external fragmentation, but still suffer from internal fragmentation.
- Paging is simple to implement and assumed as an efficient memory management technique.
- Due to equal size of the pages and frames, swapping becomes very easy.
- Page table requires extra memory space, so may not be good for a system having small RAM.

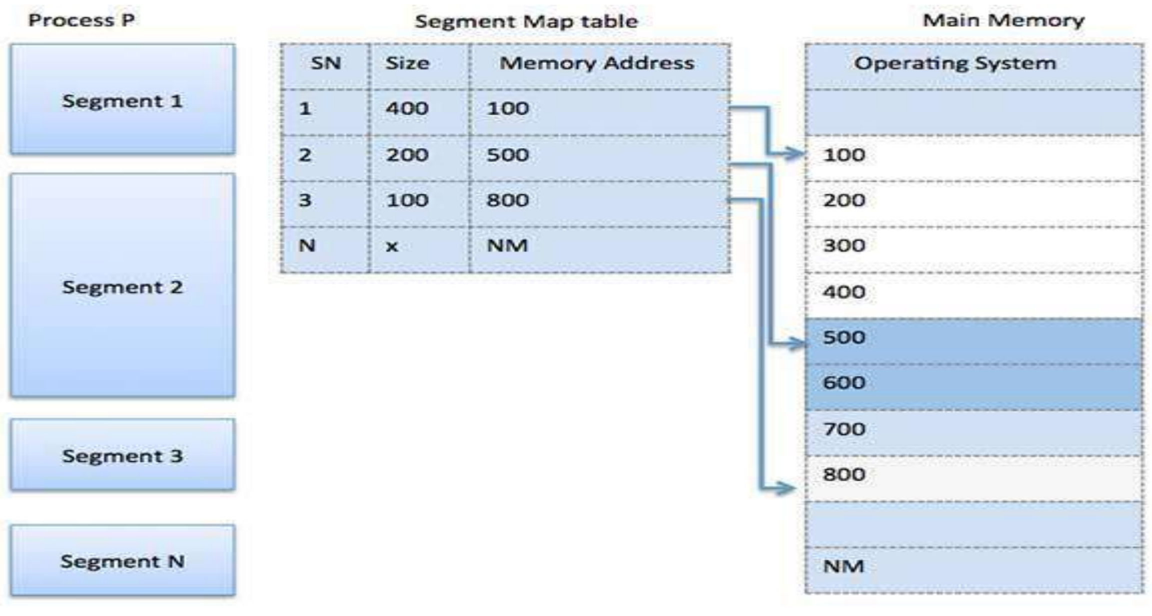
SEGMENTATION

✓ Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

✓ When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

✓ Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

✓ A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Segmentation with paging

- ✓ The IBM OS/ 2.32 bit version is an operating system running on top of the Intel 386 architecture. The 386 user segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.
- ✓ The local-address space of a process is divided into two partitions.
  1. The first partition consists of up to 8 KB segments that are private to that process.
  2. The second partition consists of up to 8KB segments that are shared among all the processes.
- ✓ Information about the first partition is kept in the local descriptor table (LDT), information about the second partition is kept in the global descriptor table (GDT).

- ✓ Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.
- ✓ The logical address is a pair (selector, offset) where the selector is a16-bit number:

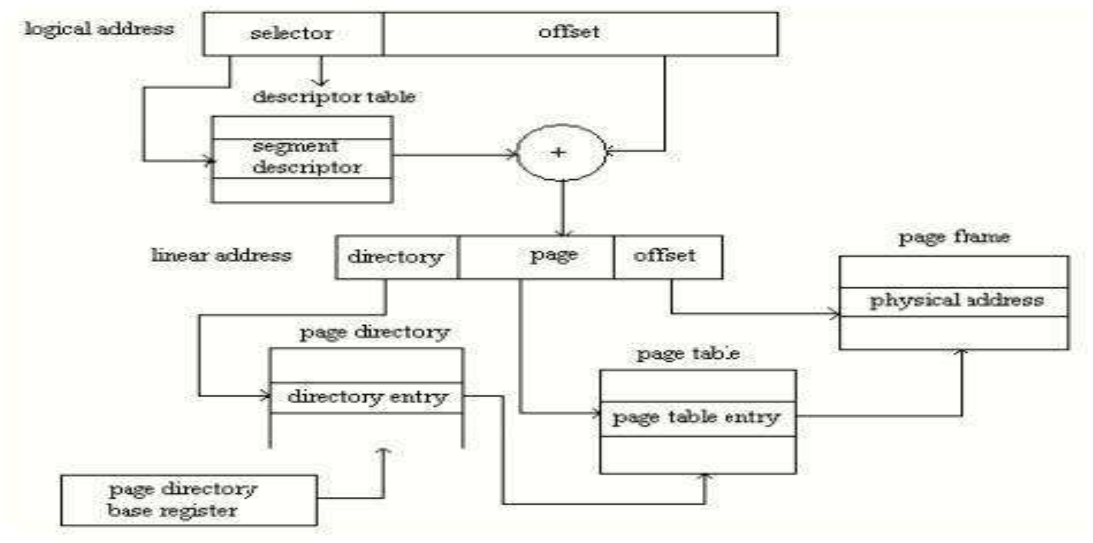
s	g	p
13	1	2

- ✓ Where s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection.

- The offset is a 32-bit number specifying the location of the byte within the segment in question.
- The base and limit information about the segment in question are used to generate a linear-address.
- ✓ First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.
- The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit page table pointer. The logical address is as follows.

s	g	p
10	10	12

- To improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.
- If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.



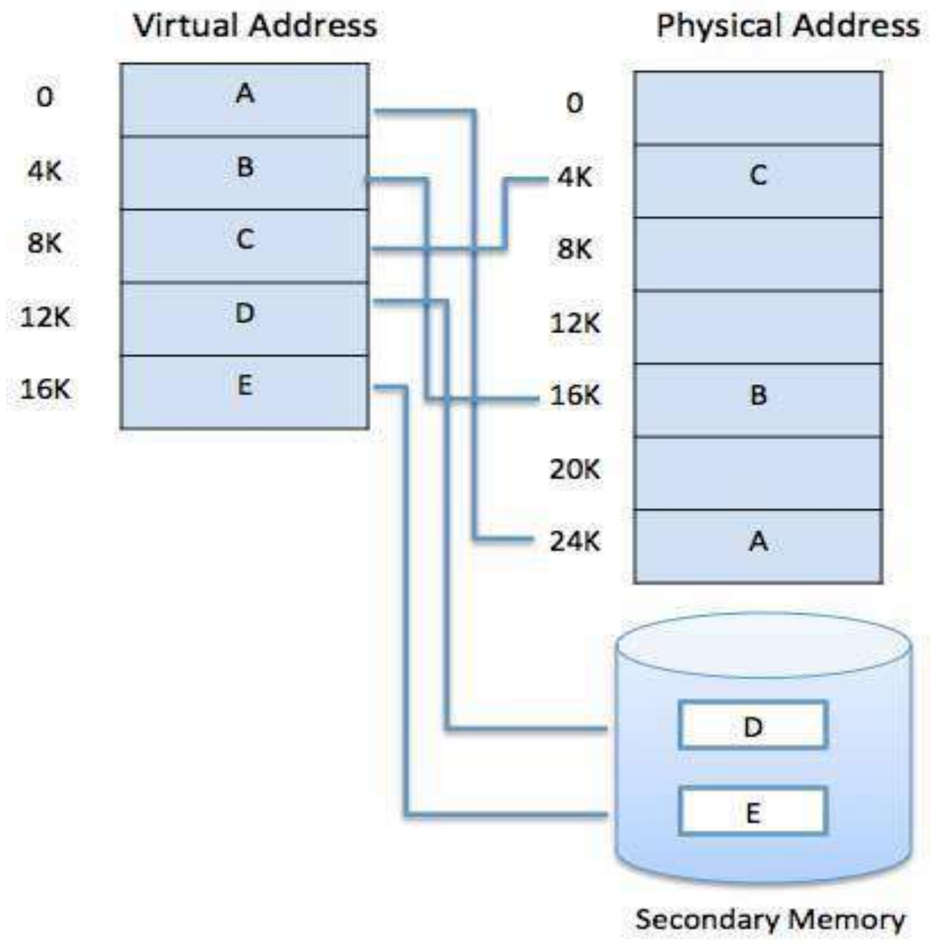
A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.

The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.

Following are the situations, when entire program is not required to be loaded fully in main memory.

- User written error handling routines are used only when an error occurred in the data or computation.
- Certain options and features of a program may be used rarely.
- Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
- The ability to execute a program that is only partially in memory would counter many benefits.
- Less number of I/O would be needed to load or swap each user program into memory.
- A program would no longer be constrained by the amount of physical memory that is available.
- Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

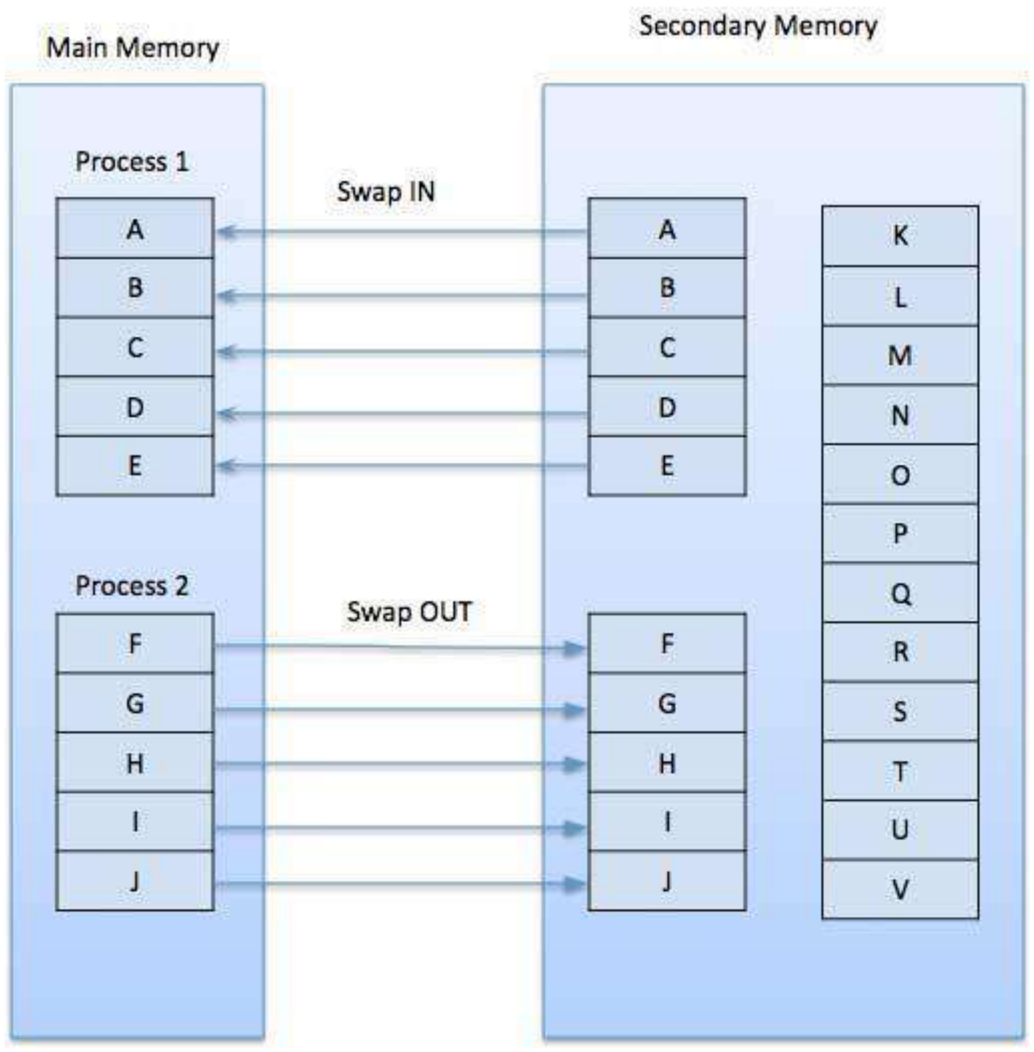
Modern microprocessors intended for general-purpose use, a memory management unit, or MMU, is built into the hardware. The MMU's job is to translate virtual addresses into physical addresses. A basic example is given below −



Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance. When a context switch occurs, the operating system does not copy any of the old program’s pages out to the disk or any of the new program’s pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program’s pages as they are referenced.



While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

Advantages

Following are the advantages of Demand Paging –

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

Disadvantages

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

PAGE REPLACEMENT

✓ If no frames are free, we could find one that is not currently being used & free it.

- ✓ We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory.
- ✓ Then we can use that freed frame to hold the page for which the process faulted.

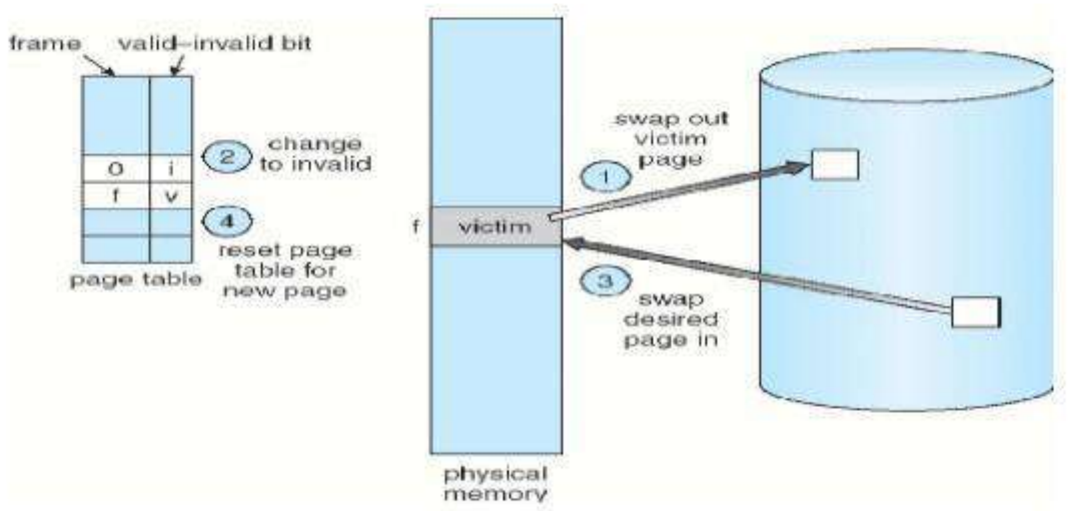
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   - If there is a free frame , then use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame
   - Write the victim page to the disk, change the page & frame tables accordingly.
3. Read the desired page into the (new) free frame. Update the page and frame tables.
4. Restart the process

Modify (dirty) bit:

- ✓ It indicates that any word or byte in the page is modified.
- ✓ When we select a page for replacement, we examine its modify bit.
  - If the bit is set, we know that the page has been modified & in this case we must write that page to the disk.
  - If the bit is not set, then if the copy of the page on the disk has not been overwritten, then we can avoid writing the memory page on the disk as it is already there.

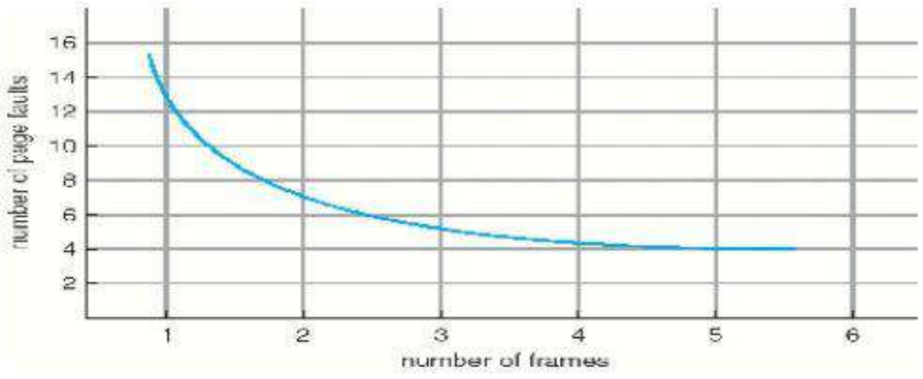
Page Replacement Algorithms



1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. LRU Approximation Page Replacement
5. Counting-Based Page Replacement

- ✓ We evaluate an algorithm by running it on a particular string of memory references & computing the number of page faults. The string of memory reference is called a reference string. The algorithm that provides less number of page faults is termed to be a good one.
- ✓ As the number of available frames increases , the number of page faults decreases. This is shown in the following graph:

(a) FIFO page replacement algorithm

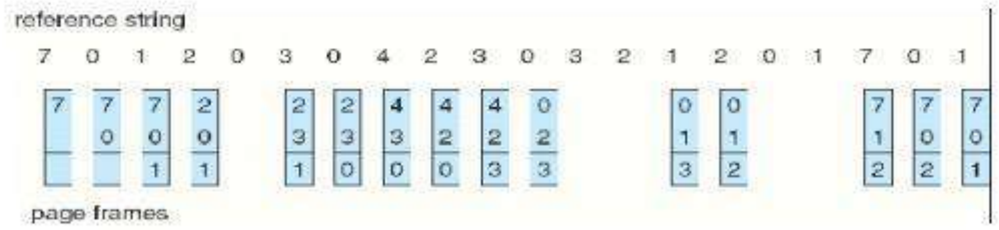


- ✓ Replace the oldest page.
- ✓ This algorithm associates with each page ,the time when that page was brought in.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3 (3 pages can be in memory at a time per process)



Drawback:

- ✓ FIFO page replacement algorithm =s performance is not always good.
- ✓ To illustrate this, consider the following example:
- ✓ Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

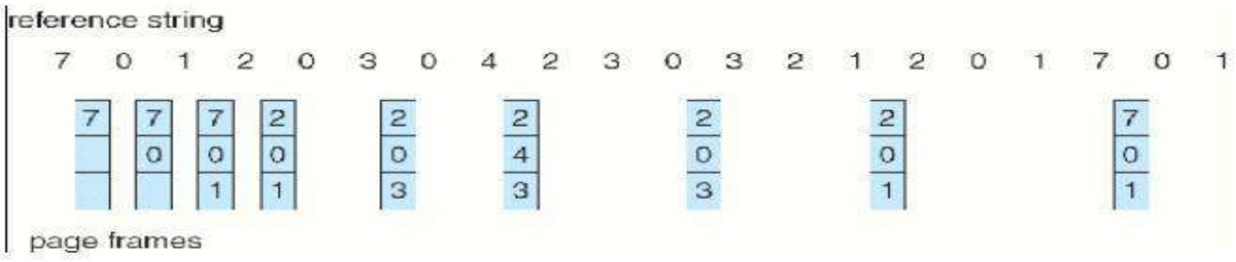
  If No.of available frames -= 3 then the no.of page faults =9

  If No.of available frames =4 then the no.of page faults =10
- ✓ Here the no. of page faults increases when the no.of frames increases .This is called as **Belady's Anomaly.**

(b) Optimal page replacement algorithm

Replace the page that will not be used for the longest period of time.

Example:



Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3

Drawback:

- ✓ It is difficult to implement as it requires future knowledge of the reference string.

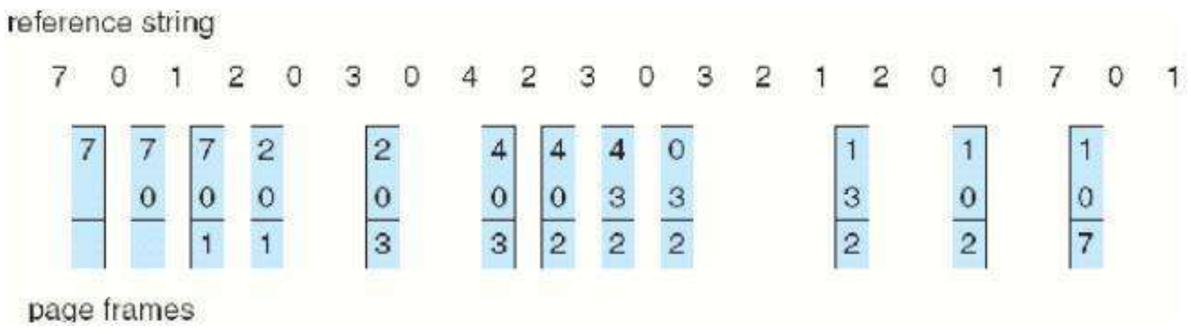
(c) LRU (Least Recently Used) page replacement algorithm

Replace the page that has not been used for the longest period of time.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No of Page Faults :12

LRU page replacement can be implemented using

1. Counters

- ✓ Every page table entry has a time-of-use field and a clock or counter is associated with the CPU.
- ✓ The counter or clock is incremented for every memory reference.
- ✓ Each time a page is referenced , copy the counter into the time-of-use field.
- ✓ When a page needs to be replaced, replace the page with the smallest counter value.

2. Stack

Keep a stack of page numbers

Whenever a page is referenced, remove the page from the stack and put it on top of the stack.

When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page).

Use of A Stack to Record The Most Recent Page References

(d) LRU Approximation Page Replacement

Reference bit

- · With each page associate a reference bit, initially set to 0
- · When page is referenced, the bit is set to 1
- ✓ When a page needs to be replaced, replace the page whose reference bit is 0
- ✓ The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- ✓ Keep an 8-bit byte for each page in a table in memory.
- ✓ At regular intervals , a timer interrupt transfers control to OS.
- ✓ The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

Example:

1. If reference bit is 00000000 then the page has not been used for 8 time periods.
2. If reference bit is 11111111 then the page has been used atleast once each time period.
3. If the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

- ✓ Basic algorithm is FIFO
- ✓ When a page has been selected , check its reference bit.
  - · If 0 proceed to replace the page
  - · If 1 give the page a second chance and move on to the next FIFO page.
- ✓ When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.
- ✓ Hence a second chance page will not be replaced until all other pages are replaced.

(iii) Enhanced Second Chance Algorithm

- ✓ Consider both reference bit and modify bit
- ✓ There are four possible classes
  1. (0,0) – neither recently used nor modified Best page to replace
  2. (0,1) – not recently used but modified page has to be written out before replacement.
  3. (1,0) - recently used but not modified page may be used again
  4. (1,1) – recently used and modified page may be used again and page has to be written to disk.

(e) Counting-Based Page Replacement

- ✓ Keep a counter of the number of references that have been made to each page
  - • Least Frequently Used (LFU)Algorithm: replaces page with smallest count
  - • Most Frequently Used (MFU)Algorithm: replaces page with largest count
- ✓ It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

-----------------------END OF UNIT III-----------------------------

UNIT IV
DEADLOCK

Definition: A process requests resources. If the resources are not available at that time ,the process enters a wait state. Waiting processes may never change state again because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

A process must request a resource before using it, and must release resource after using it.

1. **Request:** If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource
3. **Release:** The process releases the resource.

1. Deadlock Characterization

Four Necessary conditions for a deadlock

1. **Mutual exclusion:** At least one resource must be held in a non sharable mode. That is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait:** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**: Resources cannot be preempted.
4. **Circular wait:** P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2...Pn-1.

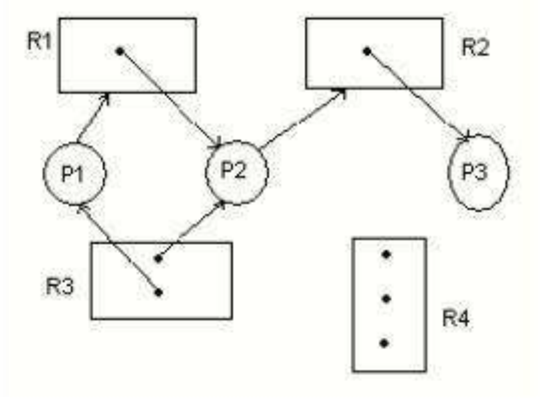
2. Resource-Allocation Graph

It is a Directed Graph with a set of vertices V and set of edges E.

V is partitioned into two types:

1. nodes P = {p1, p2,..pn}
2. Resource type R ={R1,R2,...Rm}
   Pi -->Rj - request => request edge

- ✓ Rj-->Pi - allocated => assignment edge. Pi is denoted as a circle and Rj as a square.
- ✓ Rj may have more than one instance represented as a dot with in the square.
- ✓ Sets P,R and E. P = {P1,P2,P3} R = {R1,R2,R3,R4}
- ✓ E= {P1->R1, P2->R3, R1->P2, R2->P1, R3->P3 }



Resource instances

One instance of resource type R1,Two instance of resource type R2,One instance of resource type R3,Three instances of resource type R4.

Process states

- ✓ Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.Resource Allocation Graph with a deadlock
- ✓ Process P2 is holding an instance of R1 and R2 and is waiting for an instance of resource type R3.Process P3 is holding an instance of R3.
    - P1->R1->P2->R3->P3->R2->P1
    - P2->R3->P3->R2->P2

Methods for handling Deadlocks

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection and Recovery

3. Deadlock Prevention:

- ✓ This ensures that the system never enters the deadlock state.
- ✓ Deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions cannot hold.
- ✓ By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

Denying Mutual exclusion

- ✓ Mutual exclusion condition must hold for non-sharable resources.
- ✓ Printer cannot be shared simultaneously shared by prevent processes.
- ✓ sharable resource - example Read-only files.
    - 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.

Denying Hold and wait

- ✓ Whenever a process requests a resource, it does not hold any other resource.
- ✓ One technique that can be used requires each process to request and be allocated
- ✓ all its resources before it begins execution.
- ✓ Another technique is before it can request any additional resources, it must release all the resources that it is currently allocated.

These techniques have two main disadvantages :

- First, resource utilization may be low, since many of the resources may be allocated but unused for a long time.
- We must request all resources at the beginning for both protocols.
- starvation is possible.

Denying No preemption

- ✓ 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 currently being held are preempted. (ALLOW PREEMPTION)
- ✓ These resources are implicitly released.
- ✓ The process will be restarted only when it can regain its old resources.

Denying Circular wait

- ✓ Impose a total ordering of all resource types and allow each process to request for resources in an increasing order of enumeration.

  Let R = {R1,R2,...Rm} be the set of resource types.
- ✓ Assign to each resource type a unique integer number.
- ✓ If the set of resource types R includes tapedrives, disk drives and printers.

  F(tapedrive)=1,
  F(diskdrive)=5,
  F(Printer)=12.
- ✓ Each process can request resources only in an increasing order of enumeration.

4. Deadlock Avoidance:

- ✓ Deadlock avoidance request that the OS be given in advance additional information concerning which resources a process will request and useduring its life time.
- ✓ With this information it can be decided for each request whether or not the process should wait.
- ✓ To decide whether the current request can be satisfied or must be delayed, a system must consider the resources currently available, the resources currently allocated to each process and future requests and releases of each process.

Safe State

- ✓ A state is safe if the system can allocate resources to each process in some order and still avoid a dead lock.
- ✓ A deadlock is an unsafe state.
- ✓ Not all unsafe states are dead locks
- ✓ An unsafe state may lead to a dead lock

Two algorithms are used for deadlock avoidance namely;

1. Resource Allocation Graph Algorithm - single instance of a resource type.

2. Banker's Algorithm – several instances of a resource type.

Resource allocation graph algorithm

Claim edge - Claim edge Pi---> Rj indicates that process Pi may request resource Rj at some time, represented by a dashed directed edge.

- When process Pi request resource Rj, the claim edge Pi -> Rj is converted to a request edge.
- Similarly, when a resource Rj is released by Pi the assignment edge Rj -> Pi is reconverted to a claim edge Pi -> Rj

Banker's algorithm

Available: indicates the number of available resources of each type.

Max: Max[i, j]=k then process Pi may request at most k instances of resource type Rj

Allocation : Allocation[i. j]=k, then process Pi is currently allocated K instances of resource type Rj

Need : if Need[i, j]=k then process Pi may need K more instances of resource type Rj Need [i, j]=Max[i, j]-Allocation[i, j]

Safety algorithm

```
1  Initialize work := available and Finish [i]:=false for i=1,2,3 .. n
2  Find an i such that both
       Finish[i]=false b. Needi<= Work
       if no such i exists, goto step 4
3. work :=work+ allocation i; Finish[i]:=true
     goto step 2
4. If finish[i]=true for all i, then the system is in a safe state
```

Resource Request Algorithm

Let Requesti be the request from process Pi for resources.

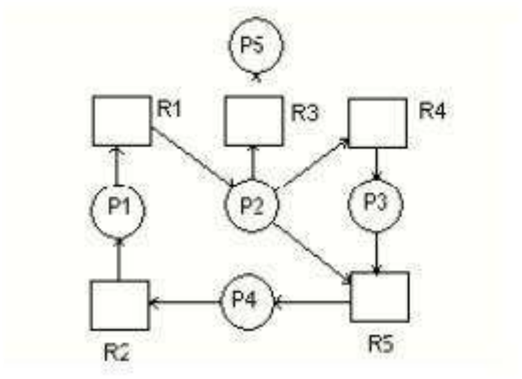
1. If Requesti<= Needi goto step2, otherwise raise an error condition, since the process has exceeded its maximum claim.
2. If Requesti <= Available, goto step3, otherwise Pi must wait, since the resources are not available.
3. Available := Availabe-Requesti;
   Allocationi := Allocationi + Requesti
   Needi := Needi - Requesti;

Now apply the safety algorithm to check whether this new state is safe or not.
If it is safe then the request from process Pi can be granted.

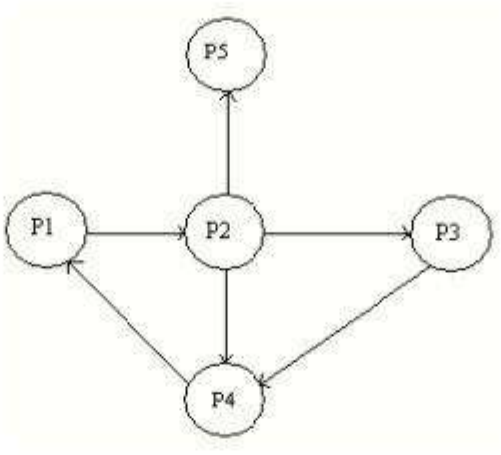
5. Deadlock Detection

(i) Single instance of each resource type

ResourceAllocation Graph



Wait for S



ii) Several Instance of a resource type

Available : Number of available resources of each type

Allocation : number of resources of each type currently allocated toeach process

Request : Current request of each process

If Request [i,j]=k, then process Pi is requesting K more instances of resource type Rj.

```
1. Initialize work := available
        Finish[i]=false, otherwise finish [i]:=true
2. Find an index i such that both
        a. Finish[i]=false
        b. Requesti<=work
        if no such i exists go to step4.
3. Work:=work+allocationi
        Finish[i]:=true goto step2
4. If finish[i]=false
        then process Pi is deadlocked
```

6. Deadlock Recovery

1. Process Termination

- ✓ Abort all deadlocked processes.
- ✓ Abort one deadlocked process at a time until the deadlock cycle is eliminated.

After each process is aborted , a deadlock detection algorithm must be invoked to determine where any process is still dead locked.

2. Resource Preemption

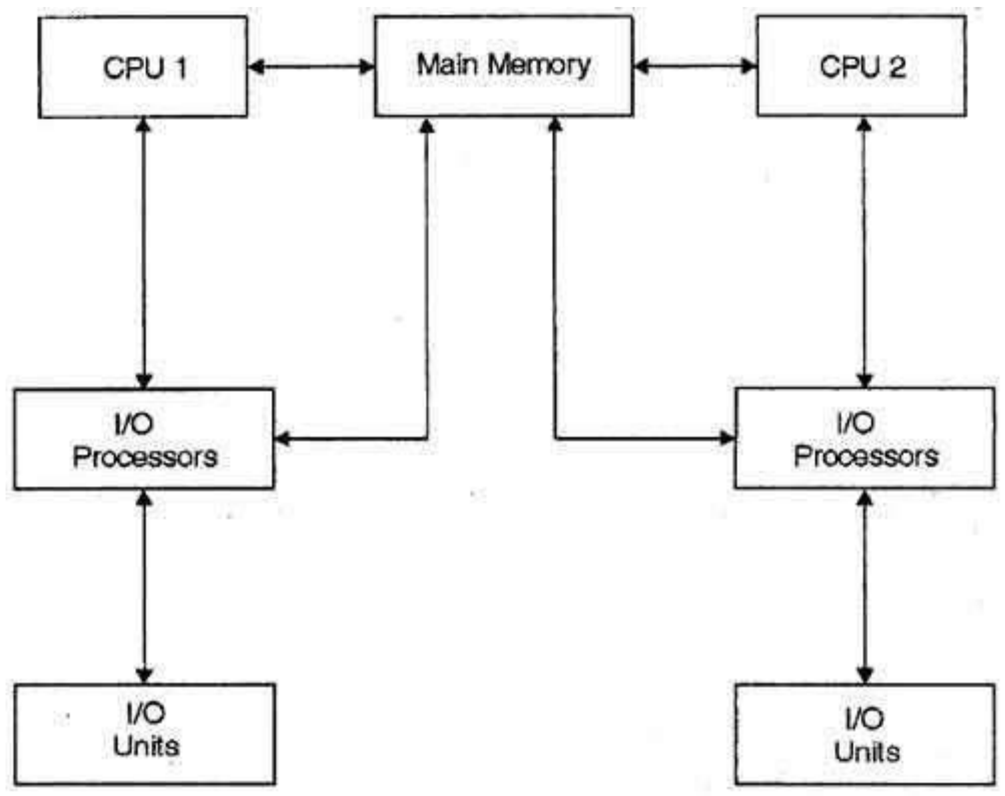
Preemptive some resources from process and give these resources to other processes until the deadlock cycle is broken.

a. **Selecting a victim:** which resources and which process are to be preempted.
b. **Rollback:** if we preempt a resource from a process it cannot continue with its normal execution. It is missing some needed resource. we must rollback the process to some safe state, and restart it from that state.
c. **Starvation :** How can we guarantee that resources will not always be preempted from the same process.

MULTIPROCESSOR OPERATING SYSTEM

Multiprocessor Operating System refers to the use of two or more central processing units (CPU) within a single computer system. These multiple CPUs are in a close communication sharing the computer bus, memory and other peripheral devices. These systems are referred as *tightly coupled systems.*

These types of systems are used when very high speed is required to process a large volume of data. These systems are generally used in environment like satellite control, weather forecasting etc. The basic organization of multiprocessing system is shown in fig.



- ✓ Multiprocessing system is based on the symmetric multiprocessing model, in which each processor runs an identical copy of operating system and these copies communicate with each other. In this system processor is assigned a specific task.
- ✓ A master processor controls the system. This scheme defines a master-slave relationship. These systems can save money in compare to single processor systems because the processors can share peripherals, power supplies and other devices.
- ✓ The main advantage of multiprocessor system is to get more work done in a shorter period of time. Moreover, multiprocessor systems prove more reliable in the situations of failure of one processor. In this situation, the system with multiprocessor will not halt the system; it will only slow it down.
- ✓ In order to employ multiprocessing operating system effectively, the computer system must have the followings:

1. Motherboard Support: A motherboard capable of handling multiple processors. This means additional sockets or slots for the extra chips and a chipset capable of handling the multiprocessing arrangement

2. Processor Support: processors those are capable of being used in a multiprocessing system.

- ✓ The whole task of multiprocessing is managed by the operating system, which allocates different tasks to be performed by the various processors in the system.
- ✓ Applications designed for the use in multiprocessing are said to be threaded, which means that they are broken into smaller routines that can be run independently. This allows the operating system to let these threads run on more than one processor simultaneously, which is multiprocessing that results in improved performance.
- ✓ Multiprocessor system supports the processes to run in parallel. Parallel processing is the ability of the CPU to simultaneously process incoming jobs. This becomes most important in computer system, as the CPU divides and conquers the jobs. Generally the parallel processing is used in the fields like artificial intelligence and expert system, image processing, weather forecasting etc.
- ✓ In a multiprocessor system, the dynamically sharing of resources among the various processors may cause therefore, a potential bottleneck. There are three main sources of contention that can be found in a multiprocessor operating system:
    - **Locking system:** In order to provide safe access to the resources shared among multiple processors, they need to be protected by locking scheme. The purpose of a locking is to serialize accesses to the protected resource by multiple processors. Undisciplined use of locking can severely degrade the performance of system. This form of contention can be reduced by using locking scheme, avoiding long critical sections, replacing locks with lock-free algorithms, or, whenever possible, avoiding sharing altogether.
    - **Shared data:** The continuous accesses to the shared data items by multiple processors (with one or more of them with data write) are serialized by the cache coherence protocol. Even in a moderate-scale system, serialization delays can have significant impact on the system performance. In addition, bursts of cache coherence traffic saturate the memory bus or the interconnection network, which also slows down the entire system. This form of contention can be eliminated by either avoiding sharing or, when this is not possible, by using replication techniques to reduce the rate of write accesses to the shared data.
    - **False sharing:** This form of contention arises when unrelated data items used by different processors are located next to each other in the memory and, therefore, share a single cache line: The effect of false sharing is the same as that of regular sharing bouncing of the cache line among several processors. Fortunately, once it is identified, false sharing can be easily eliminated by setting the memory layout of non-shared data.
    - Apart from eliminating bottlenecks in the system, a multiprocessor operating system developer should provide support for efficiently running user applications on the multiprocessor. Some of the aspects of such support include mechanisms for task placement and migration across processors, physical memory placement insuring most of the memory pages used by an application is located in the local memory, and scalable multiprocessor synchronization primitives.

Multi-Processor Systems

- Much of the discussion in this course has considered the operating system to be running on a time-shared uni-processor ... and this perspective is adequate to fully understand most of those topics. But increasingly many modern computer systems are now multi-processor:
- Multiple general purpose CPUs (as opposed to GPUs) that are capable of running unrelated programs or threads (unlike SIMD array processors) and (to some degree) share memory and I/O devices.

These systems are interesting because they are independent enough to encounter many of the problems associated with distributed computing ... but (because they share memory and I/O devices) do things that push the distributed systems envelope. As people develop applications to exploit these platforms, it is important that they understand the issues they present.

Why Build Multi-Processor Systems

- ✓ We continue to find applications that require ever more computing power. Sometimes these problems can be solved by horizontally scaled systems (e.g. thousands of web servers). But some problems demand, not more computers, but faster computers. Consider a single huge database, that each year, must handle twice as many operations as it served the previous year. Distributed locking, for so many parallel transactions on a single database, could be prohibitively expensive. The (seemingly also prohibitively expensive) alternative would be to buy a bigger computer every year.
- ✓ Long ago it was possible to make computers faster by shrinking the gates, speeding up the clock, and improving the cooling. But eventually we reach a point of dimmisnishing returns where physics (the speed of light, information theory, thermodynamics) makes it ever more difficult to build faster CPUs. Recently, most of our improvements in processing speed have come from:
    - smarter pipe-lining and increasingly parallel and speculative execution
    - putting more cores per chip, more chips per board, and more boards per computer system.
- ✓ But it is reasonable to ask whether or not 16x3B instructions per second is actually equivalent to 48B instructions per second? The answer (see Amdahl's Law) depends on whether or not your application can be divided into 16 or more parallely executable sub-tasks. Fortunately, modern operating systems tend to run large numbers of processes, and expensive computations are increasingly designed to be executable in multiple parallel threads..
- ✓ For these reasons, multi-processor is the dominant architecture for powerful servers and desktops. And, as the dominant architecture, operating systems must do a good job of exploiting them.

Multi-Processor Hardware

The above general defintion covers a wide range of architectures, that actually have very different characteristics. And so it is useful, to overview the most prominent architectures.

Hyper-Threading

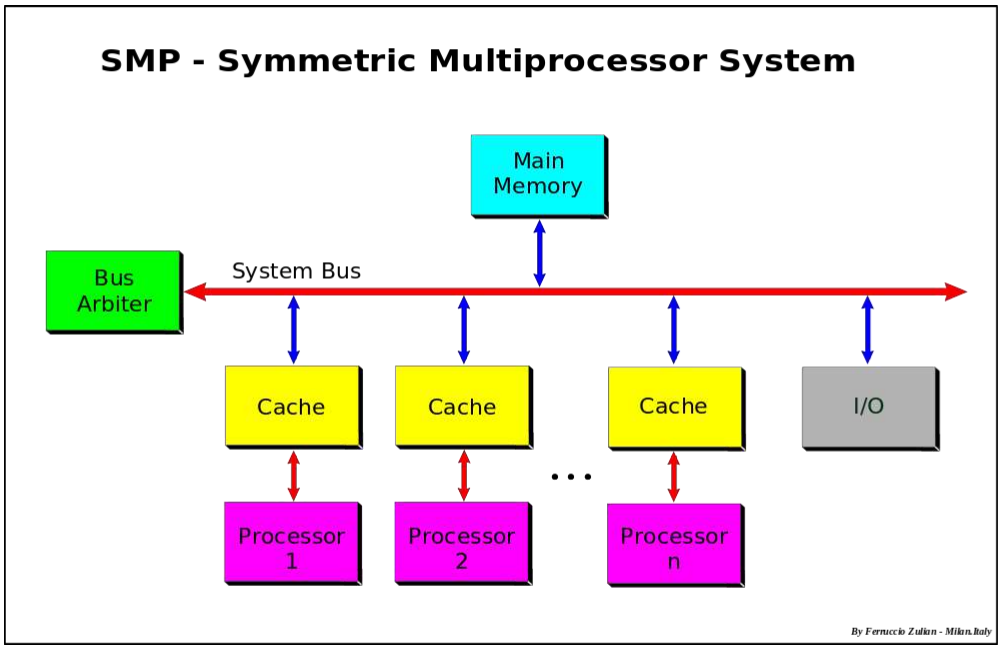
- o CPUs are much faster than memory. A 2.5GHz CPU might be able to execute more than 5 Billion instructions for second. Unfortunately, 80ns memory can only deliver 12 Million fetches or stores per second. This is almost a 1000x mis-match in performance. The CPU has multiple levels of cache to ensure that we seldom have to go to memory, but even so, the CPU spends a great deal of time waiting for memory.
- ✓ The idea of hyper-threading is to give each core two sets of general registers, and the ability to run two independent threads. When one of those threads is blocked (waiting for memory) the other thread can be using the execution engine. Think of this as non-preemptive time-sharing at the micro-code level. It is comon for a pair of hyper-threads to get 1.2-1.8 times the instructions per second that a single thread would have gotten on the same core. It is theoretically possible to

get 2x hyper-threading, but a thread might run out of L1 cache for a long time without blocking, or perhaps both hyper-threads are bocked waiting for memory.

- ✓ From a performance point-of-view, it is important to understand that both hyper-threads are running in the same core, and so sharing the same L1 and L2 cache. Thus hyper-threads that use the same address space will exhibit better locality, and hence run much better than hyper-threads that use different address spaces.

Symmetric Multi-Processors

A Symmetric Multi-Processor has some number of cores, all connected to the same memory and I/O busses. Unlike hyper-threads these cores are completely independent execution engines, and (modulo limitations on memory and bus throughput) N cores should be able to execute N times as many instructions per second as a single core.

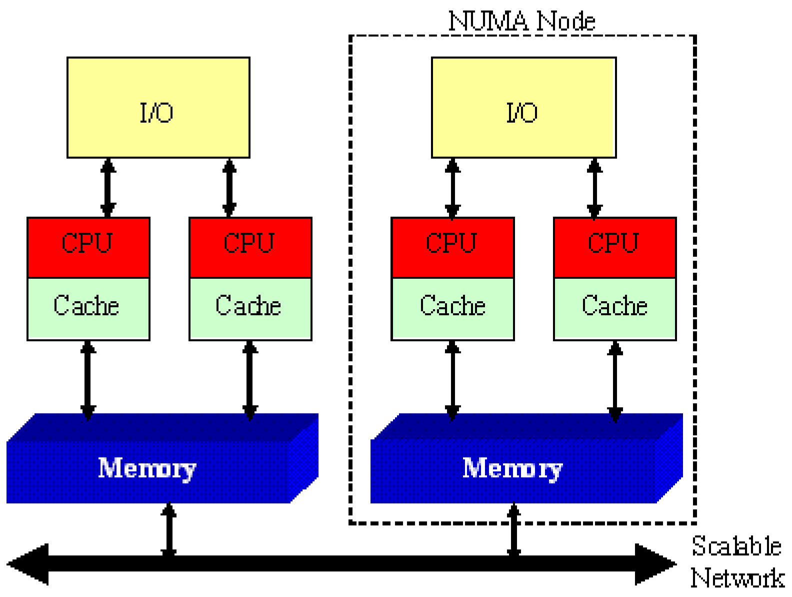


Cache Coherence

- ✓ As mentioned previously, much of processor performance is a result of caching. In most SMP systems, each processor has its own L1/L2 caches. This creates a potential *cache-coherency* problem if (for instance) processor 1 updates a memory location whose contents have been cached by processor 2. Program execution based on stale cache entries would result in incorrect results, and so must be prevented.
- ✓ There are a few general approaches to maintaining cache coherency (ensuring that there are no disagreements about the current contents of any cache line), and most SMP systems with per-processor caching incorporate some Cache Coherency Mechanism to address this issue.

Cache Coherent Non-Uniform Memory Architectures

It is not feasible to create fast memory controllers that can provide concurrent access to large numbers of cores, and eventually memory bandwidth becomes the bottleneck that prevents scaling to larger numbers of CPUs. A Non-Uniform Memory Architecture addresses this problem by giving each node or CPU its own high-speed local memory, and interconnecting all of the memory busses with a slower but more scalable network.



- ✓ Operations to local memory may be several times faster than operations to remote memory, and the maximum throughput of the scalable network may be a small fraction of the per-node local memory bandwidth. Such an architecture might provide nearly linear scaling to much larger numbers of processors, but only if we can ensure that most of the memory references are local.
- ✓ The Operating System might be able to deal with the different memory access speeds by trying allocate memory for each process from the CPU on which that process is running. But there will still be situations where multiple CPUs need to access the same memory.
- ✓ To ensure correct execution, we must maintain coherency between all of the per-node/per-CPU caches. This means that, in addition to servicing remote memory read and write requests, the scalable network that interconnects the nodes must also provide cache coherency. Such architectures are called *Cache Coherent Non-Uniform Memory Architectures* (CC-NUMA), and the implementing networks are called *Scalable Coherent Interconnects*. The best known Scalable Coherent Interconnects are probably Intel's Quick Path Interconnect (QPI), and AMD's HyperTransport.

Power Management

- ✓ The memory and cache interconnections are probably the most interesting part of a multi-processor system, but power management is another very important feature. A multi-core system can consume a huge amount of power ... and most of the time it does not need most of the cores.
- ✓ Many multi-processor systems include mechanisms to slow (or stop) the clocks on unneeded cores, which dramatically reduces system power consumption. This is not a slow process, like a sleep and reboot. A core can be returned to full speed very quickly.

Multi-Processor Operating Systems

- ✓ To exploit a multi-processor system, the operating system must be able to concurrently manage multiple threads/processes on each of the available CPU cores. One of the earliest approaches was to run the operating system on one core, and applications on all of the others. This works reasonably for a small number of cores, but as the number of cores increases, the OS becomes the primary throughput bottleneck. Scaling to larger numbers of cores requires the operating system itself to run on multiple cores. Running efficiently on multiple cores requires the operating system to carefully choose which threads/processes to run on which cores and what resources to allocate to them.

- ✓ When we looked at distributed systems, we saw (e.g. Deutsch's Seven Falacies) that the mere fact that a network is capable of distributing every operation to an arbitrary node does not make doing so a good idea. It will be seen that the same caveat applies to multi-processor systems.

Scheduling

- ✓ If there are threads (or processes) to run, we would like to keep all of the cores busy. If there are not threads (or processes) to run, we would like to put as many cores as possible into low power mode. It is tempting to think that we can just run each thread/process on the next core that becomes available (due to a process blocking or getting a time-slice-end). But some cores may be able to run some threads (or processes) far more efficiently than others.

  - dispatching a thread from the same process as the previous thread (to occupy that core) may be much less expensive because re-loading the page table (and flushing all cached TLB entries) is a very expensive operation.
  - a thread in the same process may run more efficiently because shared code and data may exploit already existing L1/L2 cache entries.
  - threads that are designed to run concurrently (e.g. parallel producer and consumer communicating through shared memory) should be run on distinct cores.

- ✓ Thus, the choice of when to run which thread in which core is not an arbitrary one. The scheduler must consider what process was last running in each core. It may make more sense to leave one core idle, and delay executing some thread until its preferred core becomes available. The operating system will try to make intelligent decisions, but if the developers understand how work can best be allocated among multi-processor cores, they can advise the operating system with operations like *sched_setaffinity(2)* and *pthead_setaffinity_np(3)*.

Synchronization

- ✓ Sharing data between processes is relatively rare in user mode code. But the operating system is full of shared data (process table entries, file descriptors, scheduling and I/O queues, etc). In a uni-processor, the primary causes of race conditions are preemptive scheduling and I/O interrupts. Both of these problems can be managed by disabling (selected) interrupts while executing critical sections ... and many operating systems simply declare that preemptions cannot occur while executing in the operating system.
- ✓ These techniques cease to be effective once the operating system is running on multiple CPUs in a multi-processor system. Disabling interrupts cannot prevent another core from performing operations on a single global object (e.g. I-node). Thus multi-processor operating systems require some other means to ensure the integrity global data structures. Early multi-processor operating systems tried to create a single, global, kernel lock ... to ensure that only one process at a time could be executing in the operating system. But this is essentially equivalent to running the operating system on a single CPU. As the number of cores increases, the (single threaded) operating system becomes the scalability bottleneck.
- ✓ The Solution to this problem is finer grained locking. And as the number of cores increased, the granularity required to achieve high parallelism became ever finer. Depending on the particular shared resource and operations, different synchronizations may have to be achieved with different mechanisms (e.g. compare and swap, spin-locks, interrupt disables, try-locks, or blocking mutexes). Moreover for every resource (and combination of resources) we need a plan to prevent

deadlock. Changing complex code that involves related updates to numerous data structures to implement fine-grained locking is difficult to do (often requiring significant design changes) and relatively brittle (as maintainers make changes that fail to honor all of the complex synchronization rules).

- ✓ If a decision is made to transition the operating system to finer grained locking, it becomes much more difficult for third party developers to build add-ons (e.g. device drivers and file systems) that will work with the finer grained locking schemes in the hosting operating system.

Because of this complexity, there are relatively few operating systems that are able to efficiently scale to large numbers of multi-processor cores. Most operating systems have chosen simplicity and maintainability over scalability.

Device I/O

If an I/O operation is to be initiated, does it matter which CPU initiates it? When an I/O interrupt comes in to a multi-processor system, which processor should be interrupted? There are a few reasons we might want to choose carefully which cores handle which I/O operations:

- as with scheduling, sending all operations for a particular device to a particular core may result in more L1/L2 cache hits and more efficient execution.
- syncrhonization between the synchronous (resulting from system calls) and asynchronous (resulting from interrupts) portions of a device driver if they are all executing in the same CPU.
- each CPU has a limited I/O throughput, and we may want to balance activity among the available cores.
- some CPUs may be bus-wise closer to some I/O devices, so that operations go more quickly initiated from some cores.

Many multi-processor architectures have interrupt controllers that are configurable for which interrupts should be delivered to which processors.

Non-Uniform Memory Architectures

- ✓ CC-NUMA is only viable if we can ensure that the vast majority of all memory references can be satisfied from local memory. Doing this turns out to create a lot of complexity for the operating system.
- ✓ When we were discussing uni-processor memory allocation, we observed that significant savings could be achieved if we shared a single copy of a (read only) load module among all processes that were running that program. This ceases to be true when those processes are running on distinct NUMA nodes. Code and other read-only data should be have a separate copy (in local memory) on each NUMA node. The cost (in terms of wasted memory) is neglibile in comparison performance gains from making all code references local.
- ✓ When a program calls *fork(2)* to create a new process, *exec(2)* to run a new program, or *sbrk(2)* to expand its address space, the required memory should always be allocated from the node-local memory pool. This creates a very strong affinity between processes and NUMA nodes. If it is necessary to migrate a process to a new NUMA node, all of its allocated code and data segments should be copied into local memory on the target node.
- ✓ As noted above, the operating system is full of data structures that are shared among many cores. How can we reduce the number or cost of remote memory references associated with those shared data structures? If the updates are few, sparse and random, there may be little we can do to optimize them ... but their costs will not be high. When more intensive use of shared data structures is required, there are two general appoaches:

1. move the data to the computation
    - lock the data structure.
    - copy it into local memory.
    - update the global pointer to reflect its new location.
    - free the old (remote) copy.
    - perform all subsequent operations on the (now) local copy.
2. move the computation to the data
    - look up the node that owns the resource in question.
    - send a message requesting it to perform the required operations.
    - await a response.

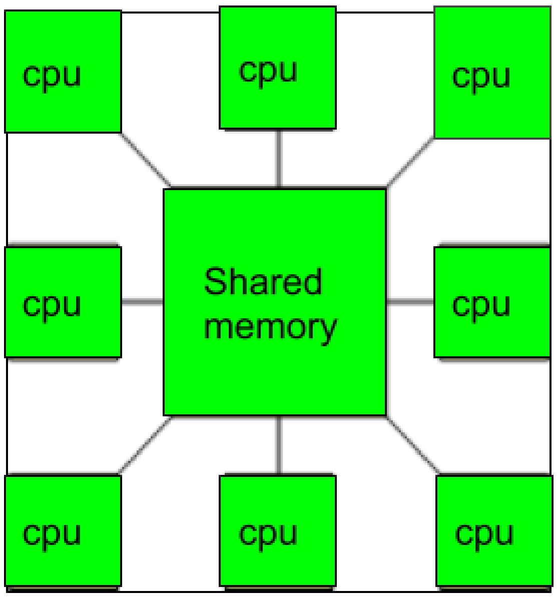
In practice, both of these techniques are used ... the choice determined by the particulars of the resource and its access.

As with fine-grained synchronization of kernel data structures, this turns out to be extremely complicated. Relatively few operating systems have been willing to pay this cost, and so (again) most opt for simplicity and maintainability over performance and scalability.

Multiprocessor and Multicomputer

Multiprocessor:

- ✓ A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost the system's execution speed, with other objectives being fault tolerance and application matching.
- ✓ There are two types of multiprocessors, one is called shared memory multiprocessor and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs shares the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



Applications of Multiprocessor –

1. As a uniprocessor, such as single instruction, single data stream (SISD).
2. As a multiprocessor, such as single instruction, multiple data stream (SIMD), which is usually used for vector processing.
3. Multiple series of instructions in a single perspective, such as multiple instruction, single data stream (MISD), which is used for describing hyper-threading or pipelined processors.

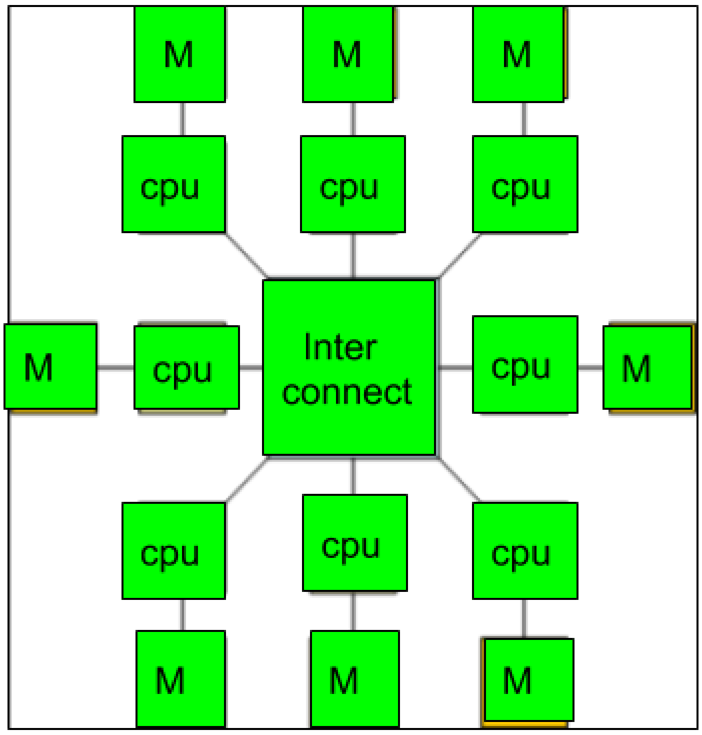
4. Inside a single system for executing multiple, individual series of instructions in multiple perspectives, such as multiple instruction, multiple data stream (MIMD).

Benefits of using a Multiprocessor –

1. Enhanced performance.
2. Multiple applications.
3. Multi-tasking inside an application.
4. High throughput and responsiveness.
5. Hardware sharing among CPUs.

2.Multicomputer:

A multicomputer system is a computer system with multiple processors that are connected together to solve a problem. Each processor has its own memory and it is accessible by that particular processor and those processors can communicate with each other via an interconnection network.



As the multicomputer is capable of messages passing between the processors, it is possible to divide the task between the processors to complete the task. Hence, a multicomputer can be used for distributed computing. It is cost effective and easier to build a multicomputer than a multiprocessor.

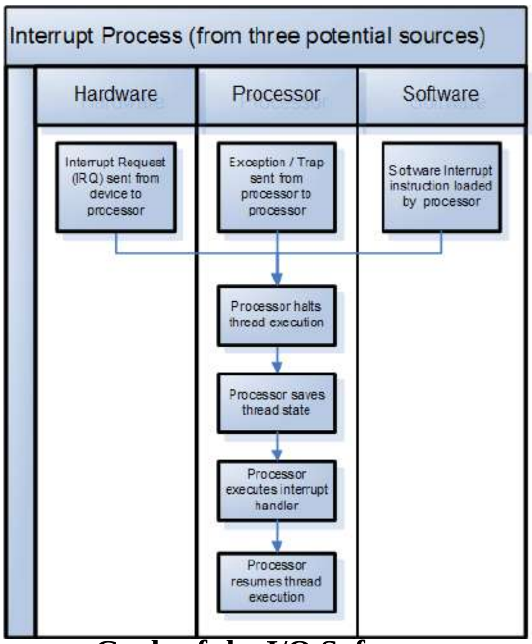
Difference between multiprocessor and Multicomputer:

1. Multiprocessor is a system with two or more central processing units (CPUs) that is capable of performing multiple tasks where as a multicomputer is a system with multiple processors that are attached via an interconnection network to perform a computation task.
2. A multiprocessor system is a single computer that operates with multiple CPUs where as a multicomputer system is a cluster of computers that operate as a singular computer.
3. Construction of multicomputer is easier and cost effective than a multiprocessor.
4. In multiprocessor system, program tends to be easier where as in multicomputer system, program tends to be more difficult.
5. Multiprocessor supports parallel computing, Multicomputer supports distributed computing.

------------------------END OF THE UNIT IV-----------------------

UNIT V

PRINCIPLES OF I/O SOFTWARE


Interrupt Process (from three potential sources)
Hardware
Processor
Software
Interrupt Request (IRQ) sent from device to processor
Exception / Trap sent from processor to processor
Software Interrupt instruction loaded by processor
Processor halts thread execution
Processor saves thread state
Processor executes interrupt handler
Processor resumes thread execution

Goals of the I/O Software

- ✓ A key concept in the design of I/O software is known as **device independence**. It means that I/O devices should be accessible to programs without specifying the device in advance.
- ✓ **Uniform Naming**, simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file-system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device.
- ✓ **Error Handling**: If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.
- ✓ **Synchronous (blocking) and Asynchronous (interrupt-driven) transfers**: Most physical I/O is asynchronous, however, some very high-performance applications need to control all the details of the I/O, so some operating systems make asynchronous I/O available to them.
- ✓ **Buffering**: Often data that come off a device cannot be stored directly in their final destination.
- ✓ **Sharable and Dedicated devices**: Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished. Then another user can have the printer. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems.

Programmed I/O

This is one of the three fundamentally different ways that I/O can be performed. The programmed I/O was the most simple type of I/O technique for the exchanges of data or any types of communication between the processor and the external devices. With programmed I/O, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time. The overall operation of the programmed I/O can be summaries as follow:

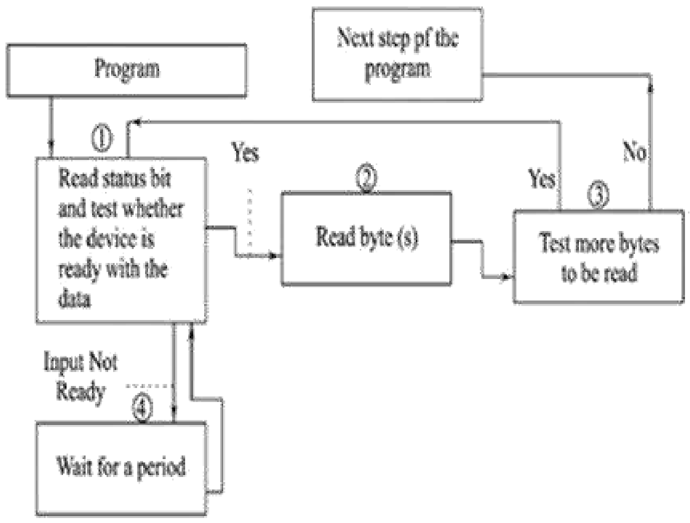
- The processor is executing a program and encounters an instruction relating to I/O operation.

- The processor then executes that instruction by issuing a command to the appropriate I/O module.
- The I/O module will perform the requested action based on the I/O command issued by the processor (READ/WRITE) and set the appropriate bits in the I/O status register.
- The processor will periodically check the status of the I/O module until it find that the operation is complete.

Programmed I/O Mode: Input Data Transfer

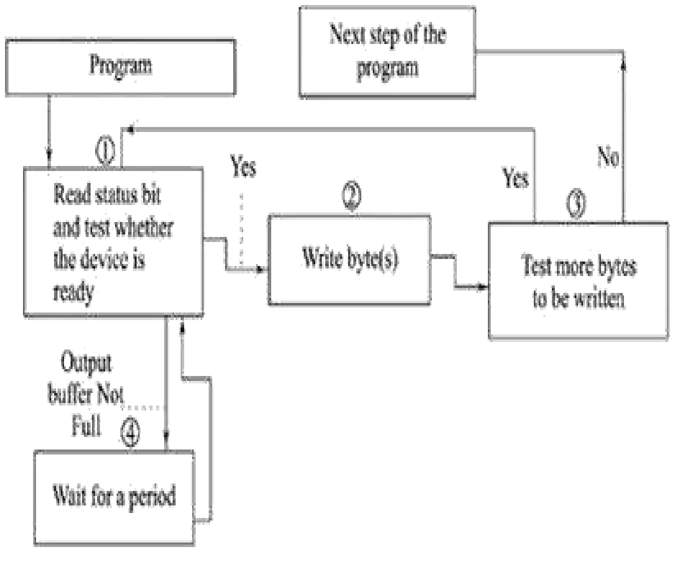
Each input is read after first testing whether the device is ready with the input (a state reflected by a bit in a status register).
The program waits for the ready status by repeatedly testing the status bit and till all targeted bytes are read from the input device.

- The program is in busy (non-waiting) state only after the device gets ready else in wait state.


Program
Next step pf the program
①
Yes
Read status bit and test whether the device is ready with the data
②
Read byte (s)
Yes
③
No
Test more bytes to be read
Input Not Ready
④
Wait for a period

Programmed I/O Mode: Output Data Transfer

- Each output written after first testing whether the device is ready to accept the byte at its output register or output buffer is empty.
- The program waits for the ready status by repeatedly testing the status bit(s) and till all the targeted bytes are written to the device.
- The program in busy (non-waiting) state only after the device gets ready else wait state.



Programmed I/O Commands

- ✓ To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:
  - **Control**
  - **Test**
  - **Read**
  - **Write**

```
copy_from_user(buffer, p, count);              /* p is the kernel bufer */
for (i = 0; i < count; i++) {                  /* loop on every character */
    while (*printer_status_reg != READY) ;     /* loop until ready */
    *printer_data_register = p[i];             /* output one character */
}
return_to_user( );
```

Advantages of Programmed I/O

- ✓ Simple to implement
- ✓ Very little hardware support

Disadvantages of Programmed I/O

- Busy waiting
- Ties up CPU for long period with no useful work

Interrupt-Driven I/O

- ✓ Interrupt driven I/O is an alternative scheme dealing with I/O. Interrupt I/O is a way of controlling input/output activity whereby a peripheral or terminal that needs to make or receive a data transfer sends a signal. This will cause a program interrupt to be set. At a time appropriate to the priority level of the I/O interrupt. Relative to the total interrupt system, the processors enter an interrupt service routine.

Interrupt I/O Inputs

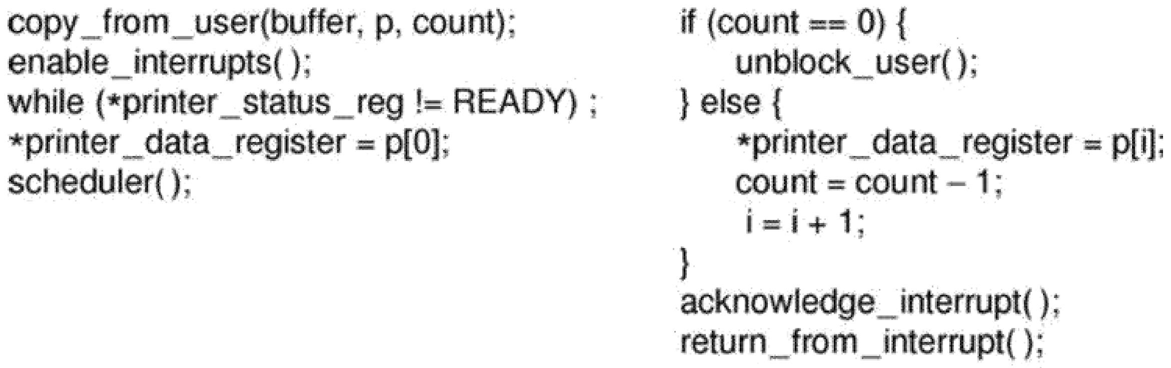
- ✓ For input, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports or memory mapping.

Interrupt I/O Outputs

- ✓ For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

Operations in Interrupt I/O

- CPU issues read command.
- I/O module gets data from peripheral whilst CPU does other work.
- I/O module interrupts CPU.
- CPU requests data.
- I/O module transfers data.



```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
     unblock_user( );
} else {
     *printer_data_register = p[i];
     count = count – 1;
      i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Advantages of Interrupt-Driven I/O

- Its faster than Programmed I/O.
- Efficient too.

Disadvantages of Interrupt-Driven I/O

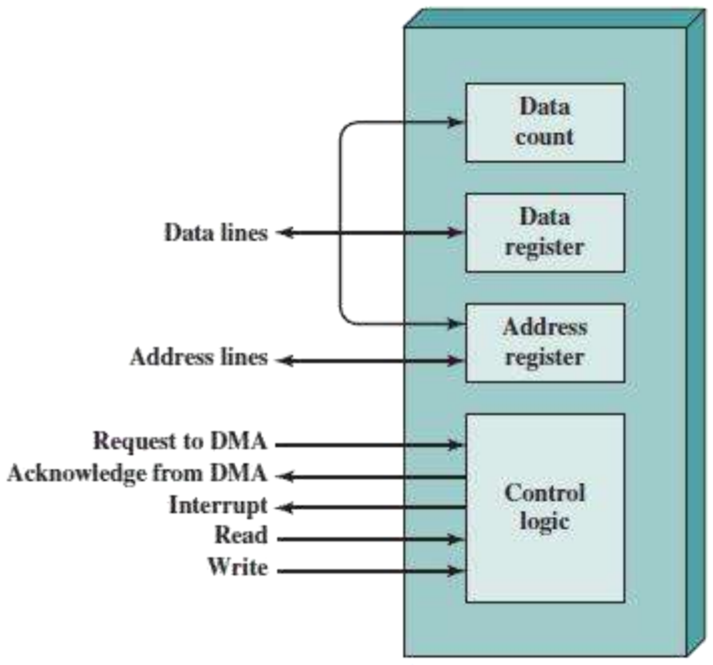
- It can be tricky to write if using a low level language.
- It can be tough to get various pieces to work well together.

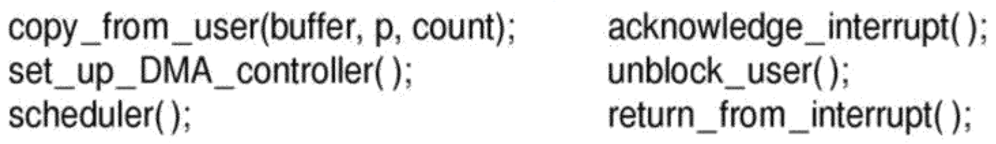
I/O Using DMA

- ✓ Direct Memory Access is a technique for transferring data within main memory and external device without passing it through the CPU.
- ✓ DMA is a way to improve processor activity and I/O transfer rate by taking-over the job of transferring data from processor, and letting the processor to do other tasks.
- ✓ This technique overcomes the drawbacks of other two I/O techniques which are the time consuming process when issuing a command for data transfer and tie-up the processor in data transfer while the data processing is neglected. It is more efficient to use DMA method when large volume of data has to be transferred.
- ✓ For DMA to be implemented, processor has to share its' system bus with the DMA module. Therefore, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily.

Operations of Direct Memory Access

- Read or Write Command
- Control Lines Communication
- Data Lines Communication



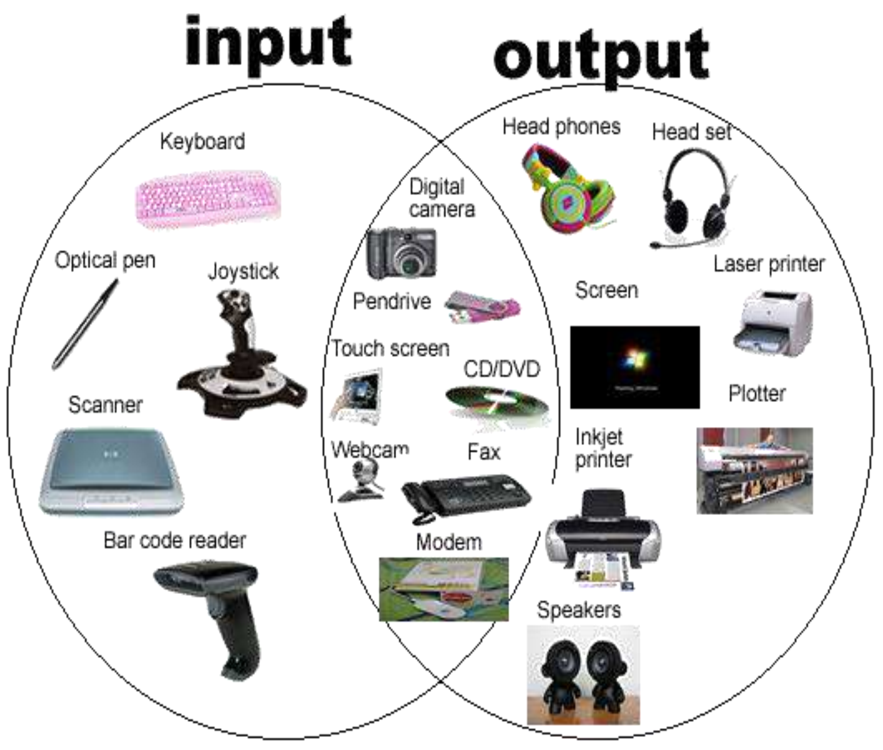


```
copy_from_user(buffer, p, count);        acknowledge_interrupt( );
set_up_DMA_controller( );                unblock_user( );
scheduler( );                            return_from_interrupt( );
```

Advantages of DMA

- Speed: no waiting due to much shorter execution path and no rotation delay.

PRINCIPLES OF I/O HARDWARE



I/O Devices

I/O devices can be roughly divided into two categories: block devices and character devices.

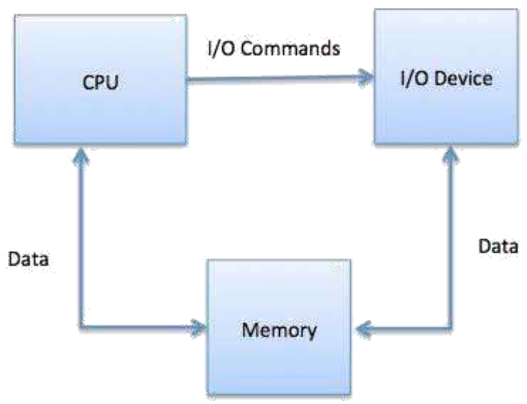
- **Block Devices**
  It stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 to 65,536 bytes.
  Examples are hard disks, Blu-ray discs, and USB sticks.
- **Character Device**
  It delivers or accepts a stream of characters, without regard to any block structure. Not addressable and does not have any seek operation.
  Examples are printers, network interfaces, mice, and most other devices that are not disk-like can be seen as character devices.
- **Doesn't Really Fit**
  Some devices don't fit into this division: For instance, clocks aren't block addressable, nor do they accept character streams. All they do is cause interrupts... at timed intervals. Memory-mapped screens do not fit this division either.

Device Controllers

✓ The electronic component of I/O units is called the device controller or adapter. Operating systems use device drivers to handle all I/O devices. There is a device controller and a device driver for each device to communicate with the operating system. 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, and perform error correction as necessary.

- **Cathode Ray Tube (CRT) Controller**
  Older version of monitors that were bulky, power hungry and fragile!! CRT monitors fire a beam of electrons onto a fluorescent screen. Using magnetic fields, the system is able to bend the beam and draw pixels on the screen.
  The first "laptops" weighed about 12 kilos.
- **LCD Controller**
  This works as a bit serial device at low level. It reads bytes containing the characters to be displayed from memory and generates the signals to modify the polarization of the backlight for the corresponding pixels in order to write them on screen.
  The presence of the controller means that the OS programmer does not need to explicitly program the electrial field of the screen!

Memory-Mapped I/O

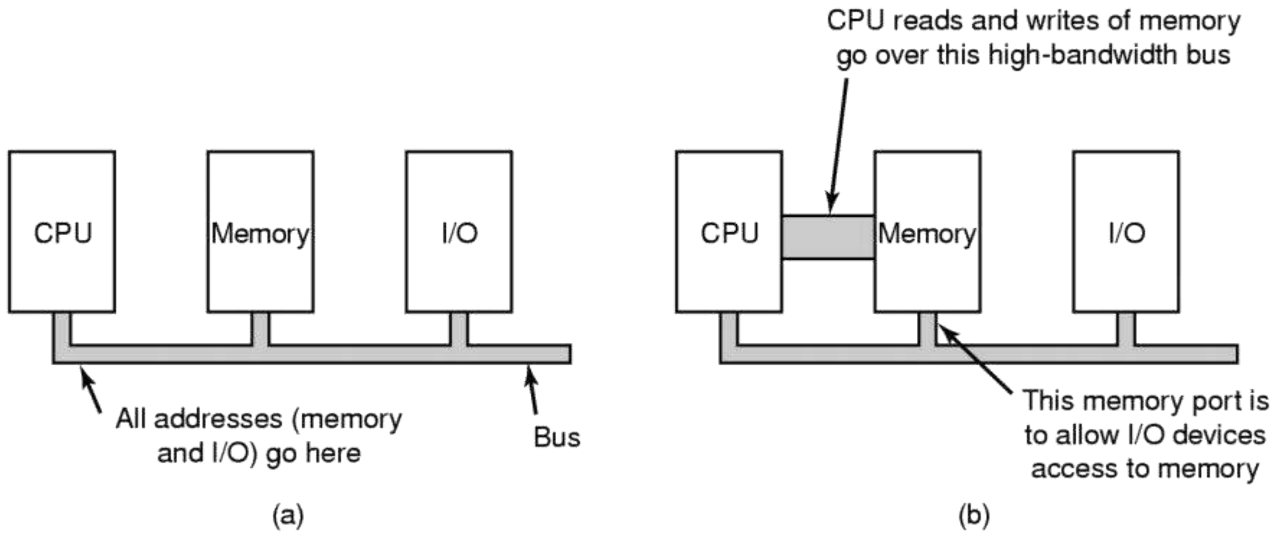


✓ The controller has registers (similar to CPU registers, but for the device) and the OS can write these registers to "give orders" to the device (e.g., "shut down" or "accept data") or read its state (e.g., "are you busy tonight?").

✓ CPU interaction with the control registers and device data buffers either through dedicated port allocation or using device memory to map them all. CPU can communicate with the control registers and the device data buffers in three ways.

- **Seperate I/O and Memory Space**: Each control register is assigned an I/O port number.
  We use special I/O instructions like:
  IN REG, PORT
  OUT PORT, REG
  IN and MOV are quite different instructions!
- **Memory Mapped I/O**: 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.
- **Hybrid**: Memory-mapped data buffers and separate I/O ports for the control registers. (Pentium)
- **Strengths of MM I/O:**

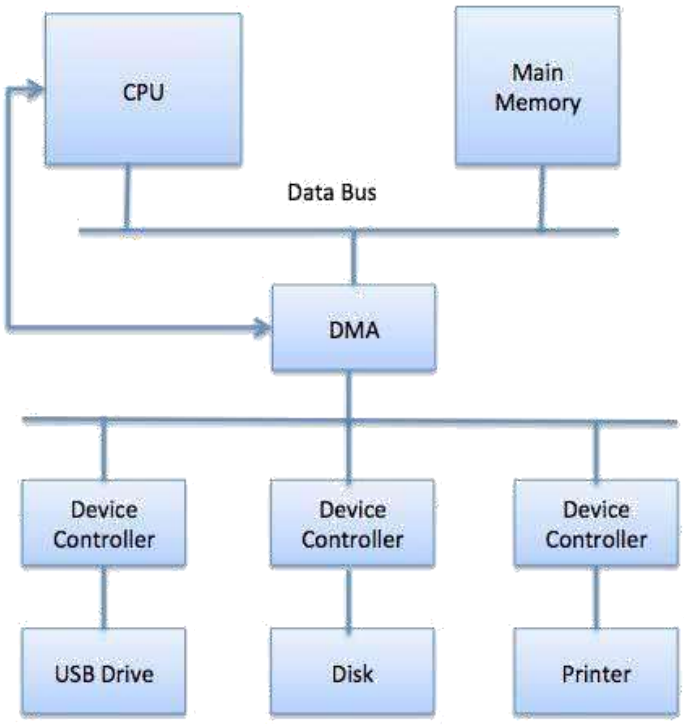
  Special I/O instructions require the OS to
  resort to assembly code: IN and OUT cannot be executed in C or C++.
  Memory-mapped I/O allows C to simply write to memory.
  Control registers are mapped to memory as well.
- **Weaknesses of MM I/O:**
  Memory-caching a device I/O register is disastrous.
  We never detect when the device has changed state!
  To fix this requires selective disabling of caching.
  - All memory modules and all devices must examine each memory reference to see if it is for them.
    If there is a high-speed memory bus (as is typical nowadays) then the I/O devices won't see the memory addresses on the high-speed bus.
    To fix this, we might send all requests to memory and see if they fail, then send them to I/O devices.
    Or, we can "snoop" on memory requests and send appropriate ones to I/O controllers. But they may be slow!
    Or we can assign some range of addresses as "not real" memory. But these would not to be assigned at boot time: no dynamic loading of devices!


CPU
Memory
I/O
All addresses (memory and I/O) go here
Bus
(a)
CPU reads and writes of memory go over this high-bandwidth bus
CPU
Memory
I/O
This memory port is to allow I/O devices access to memory
(b)

(a) Single Bus Architecture (b) Dual Bus Architecture

Direct Memory Access

- ✓ To reduce the overhead of interrupts, DMA hardware bypasses CPU to transfer data directly between I/O device and memory. 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, rather than a byte at a time.



✓

Direct Memory Access Controller

- ✓ DMA controller (DMAC) manages the data transfers and arbitrates access to the system bus. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers.

Working of DMA

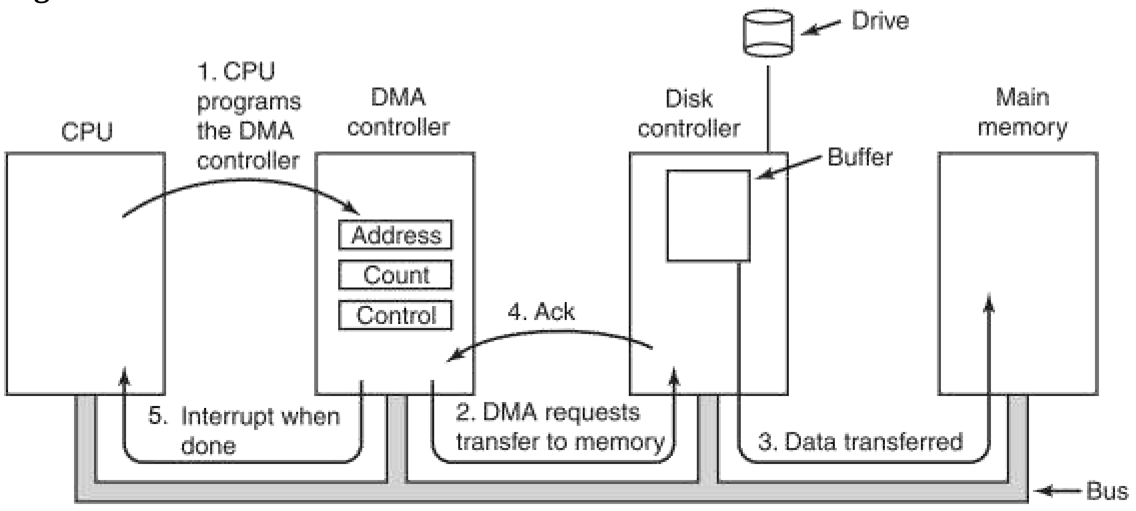


Figure 5-4. Operation of a DMA transfer.

- First the CPU programs the DMA controller by setting its registers so it knows what to transfer where

- Alongside, DMAC issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.
- The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller
- The write to memory is another standard bus cycle
- When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus
- The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete
- DMA controllers vary considerably in their sophistication. The simplest ones handle one transfer at a time, whereas sophisticated DMAC have multiple sets of registers internally, one for each channel. Word transfer may be set up to use a round-robin algorithm, or it may have a priority scheme design to favor some devices over others. Many buses can operate in two modes: **word-at-a-time mode** and **block mode**. Some DMA controllers can also operate in either mode. In word-at-a-time mode, the DMA controller requests the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly. In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called **burst mode**. It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition. The down side to burst mode is that it can block the CPU and other devices for a substantial period if a long burst is being transferred.

DISK SCHEDULING AND MANAGEMENT

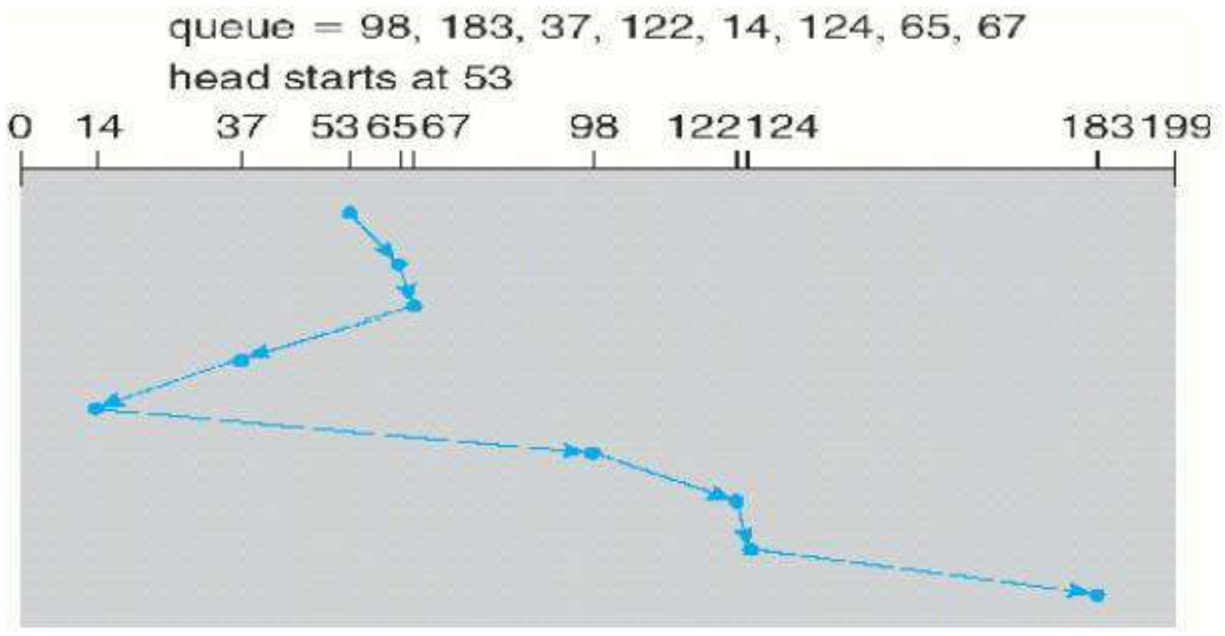
1. Disk scheduling

1.1 FCFS Scheduling

1.2

1.2 SSTF (shortest-seek-time-first)Scheduling

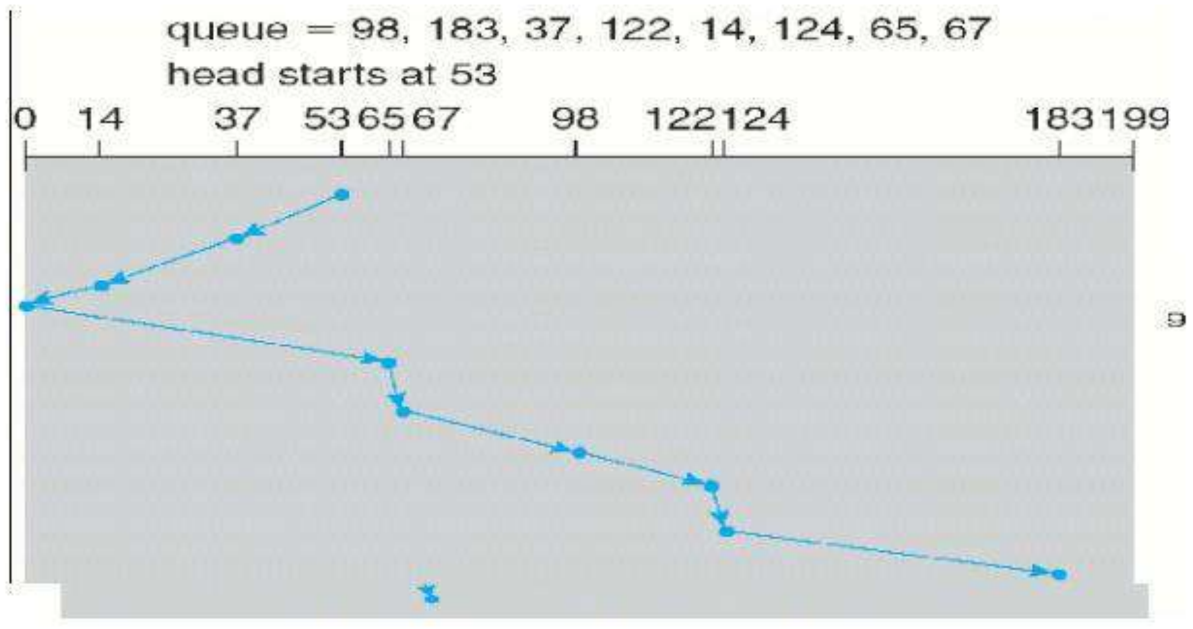
- Service all the requests close to the current head position, before moving the head far away to service other requests. That is selects the request with the minimum seek time from the current



head position**.**

1.3 SCAN Scheduling

- The disk head starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

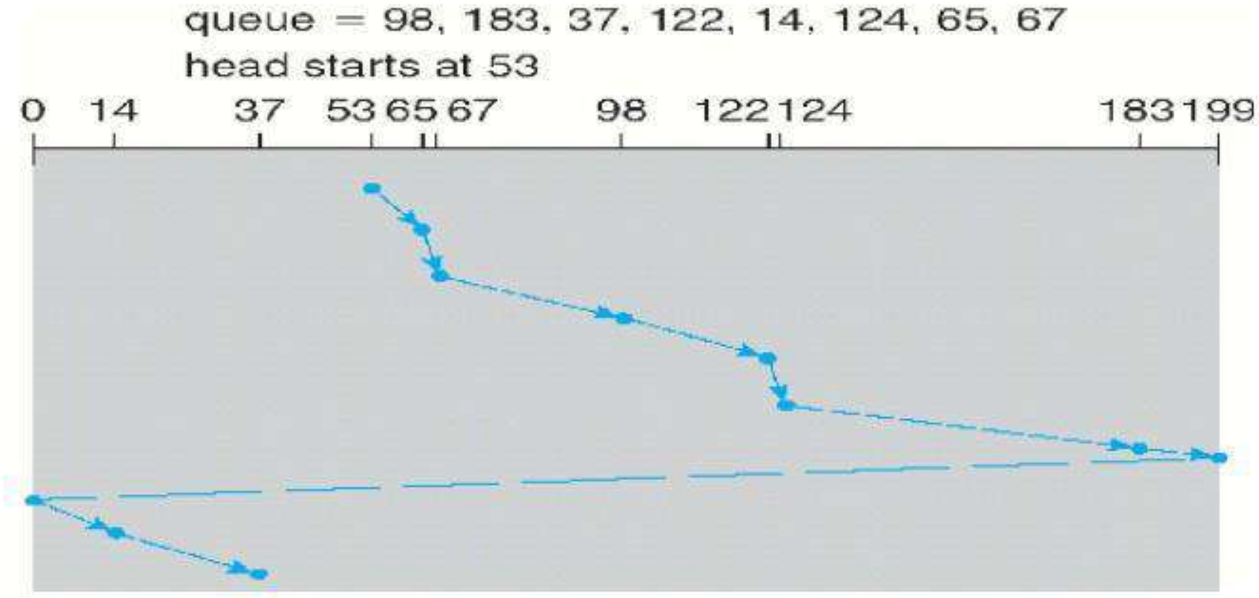


Elevator algorithm:

- ✓ Sometimes the SCAN algorithm is called as the elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up, and then reversing to service requests the other way.

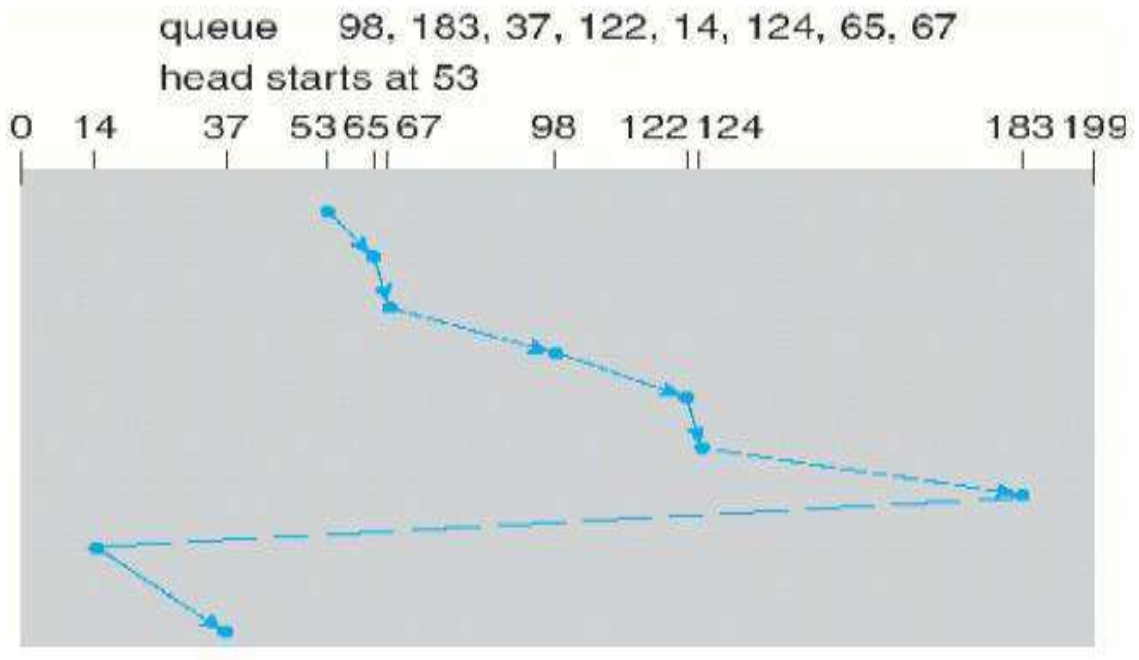
1.4 C-SCAN Scheduling

Variant of SCAN designed to provide a more uniform wait time. It moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.



1.5 LOOK Scheduling

Both SCAN and C-SCAN move the disk arm across the full width of the disk. In this, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.



2. Disk Management

Disk Formatting:

Low-level formatting or physical formatting:

- ✓ Before a disk can store data, the sector is divided into various partitions. This process is called low-level formatting or physical formatting. It fills the disk with a special data structure for each sector.
- ✓ The data structure for a sector consists of
  - Header,
  - Data area (usually 512 bytes in size), and
  - Trailer.
- ✓ The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC).
- ✓ This formatting enables the manufacturer to Test the disk and
- ✓ To initialize the mapping from logical block numbers

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps.

1. The first step is Partition the disk into one or more groups of cylinders. Among the partitions, one partition can hold a copy of the OS's executable code, while another holds user files.
2. The second step is logical formatting .The operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space and an initial empty directory.

Boot Block:

- For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial progr am is called bootstrap program & it should be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.

- To do its job, the bootstrap program
- Finds the operating system kernel on disk,
- Loads that kernel into memory, and
- Jumps to an initial address to begin the operating-system execution.

Advantages:

- ROM needs no initialization.
- It is at a fixed location that the processor can start executing when powered up or reset.

- ✓ It cannot be infected by a computer virus. Since, ROM is read only.
- ✓ The full bootstrap program is stored in a partition called the boot blocks, at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.
- ✓ The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code.
- ✓ Bootstrap loader: load the entire operating system from a non-fixed location on disk, and to start the operating system running.

Bad Blocks:

- The disk with defected sector is called as bad block.
- Depending on the disk and controller in use, these blocks are handled in a variety of ways;

Method 1: “Handled manually

- ✓ If blocks go bad during normal operation, a special program must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

Method 2: “sector sparing or forwarding”

- ✓ The controller maintains a list of bad blocks on the disk. Then the controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding.

A typical bad-sector transaction might be as follows:

- The oper ating system tries to read logical block 87.
- The controller calculates the ECC and finds that the sector is bad.
- It reports this finding to the operating system.
- The next time that the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Method 3: “sector slipping”

For an example, suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202,moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

FILE SYSTEM STORAGE

1 File Concept

- A file is a named collection of related information that is recorded on secondary storage.
- From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.
  Examples of files:
- A text file is a sequence of characters organized into lines (and possibly pages).
- A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

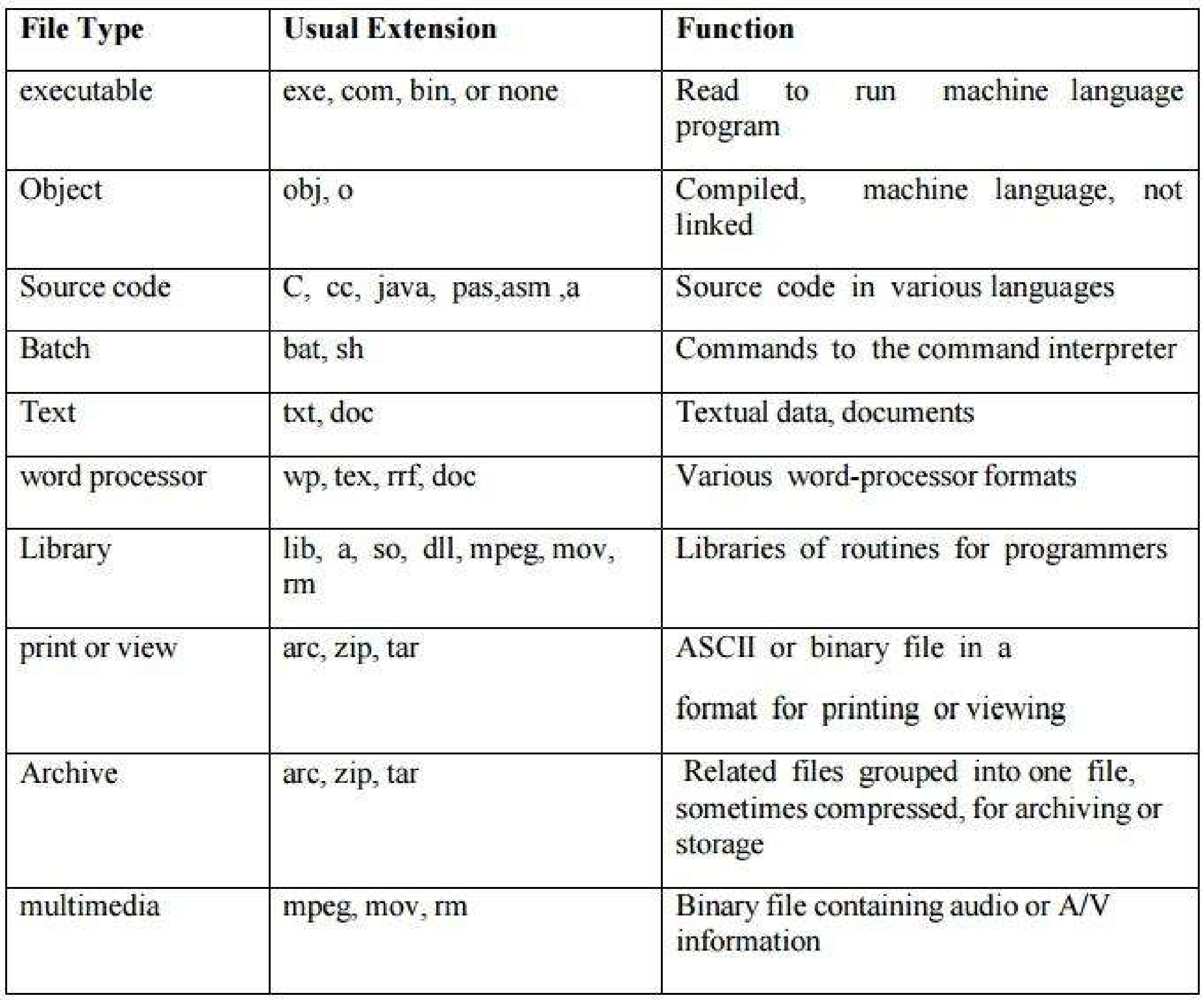
2 File Attributes

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

3 File Operations

- Creating a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file

4 File Types



File Type	Usual Extension	Function
executable	exe, com, bin, or none	Read to run machine language program
Object	obj, o	Compiled, machine language, not linked
Source code	C, cc, java, pas,asm ,a	Source code in various languages
Batch	bat, sh	Commands to the command interpreter
Text	txt, doc	Textual data, documents
word processor	wp, tex, rrf, doc	Various word-processor formats
Library	lib, a, so, dll, mpeg, mov, rm	Libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
Archive	arc, zip, tar	Related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	Binary file containing audio or A/V information

DIRECTORY STRUCTURE:

There are five directory structures. They are

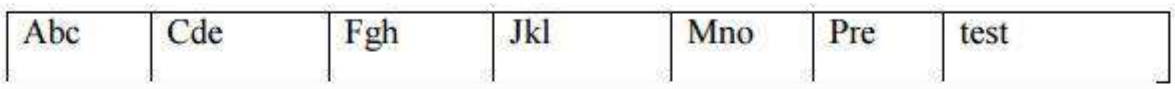
- Single-level directory
- Two-level directory
- Tree-Structured directory
- Acyclic Graph directory
- General Graph directory

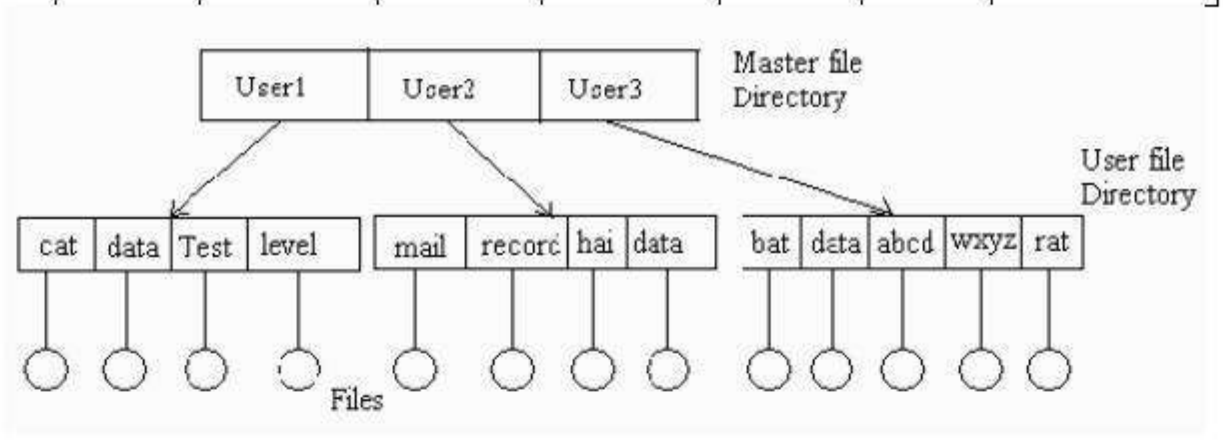
1. Single – Level Directory

- The simplest directory structure is the single- level directory.
- All files are contained in the same directory.

Disadvantage:
When the number of files increases or when the system has more than one user, since all files are in the same directory, they must have unique names.

Directory





2. Two – Level Directory

- ✓ In the two level directory structures, each user has her own user file directory (UFD).
- ✓ When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
- ✓ When a user refers to a particular file, only his own UFD is searched.
- ✓ Thus, different users may have files with the same name.
- ✓ Although the two – level directory structure solves the name-collision problem

Disadvantage:

Users cannot create their own sub-directories.

3. Tree – Structured Directory

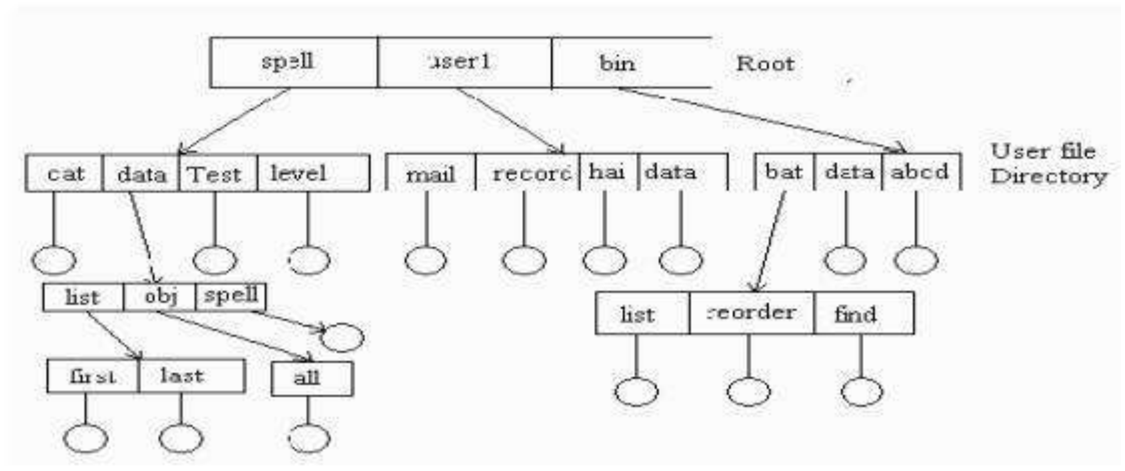
- ✓ A tree is the most common directory structure.
- ✓ The tree has a root directory. Every file in the system has a unique path name.
- ✓ A path name is the path from the root, through all the subdirectories to a specified file.
- ✓ A directory (or sub directory) contains a set of files or sub directories.
  - A directory is simply another file. But it is treated in a special way.
  - All directories have the same internal format.
  - One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
  - Special system calls are used to create and delete directories.
  - Path names can be of two types: absolute path names or relative path names.
  - An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
  - A relative path name defines a path from the current directory.

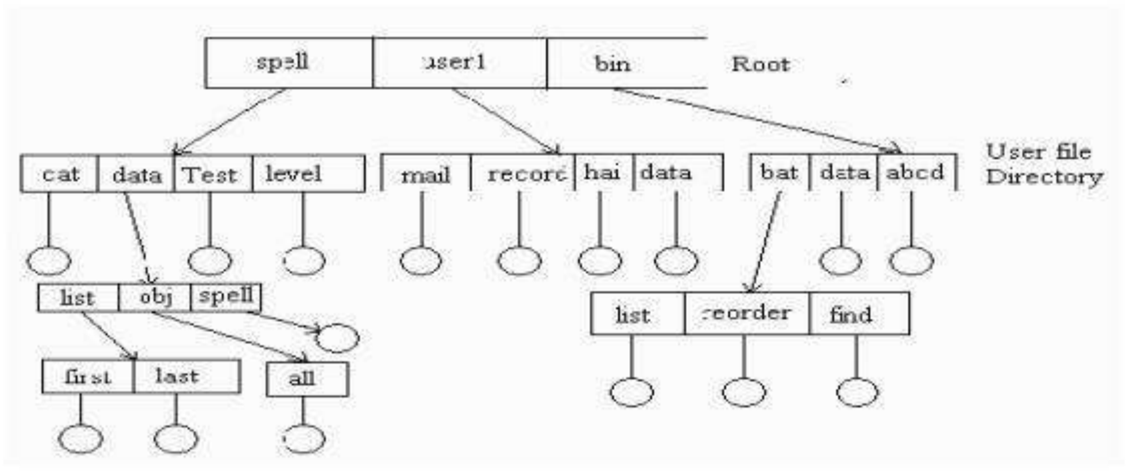
4. Acyclic Graph Directory.

- An acyclic graph is a graph with no cycles.
- To implement shared files and subdirectories this directory structure is used.
- An acyclic – graph directory structure is more flexible than is a simple tree structure, but it is also more complex. In a system where sharing is implemented by symbolic link, this situation is

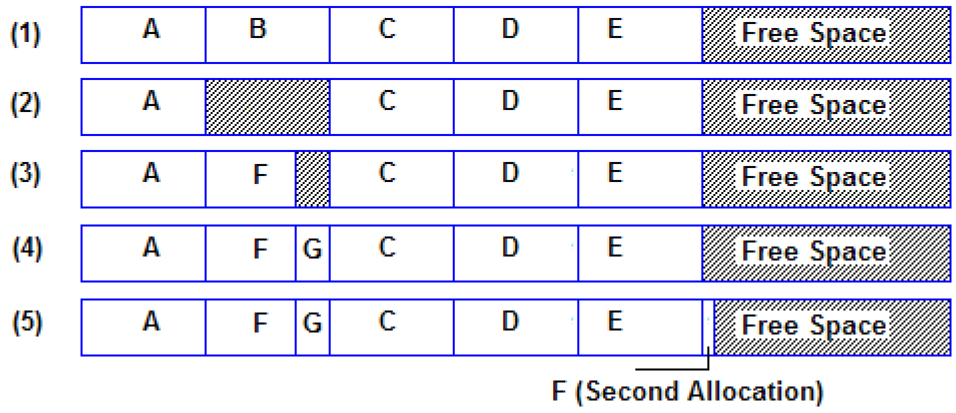
somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed.

- Another approach to deletion is to preserve the file until all references to it are deleted.
- To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.



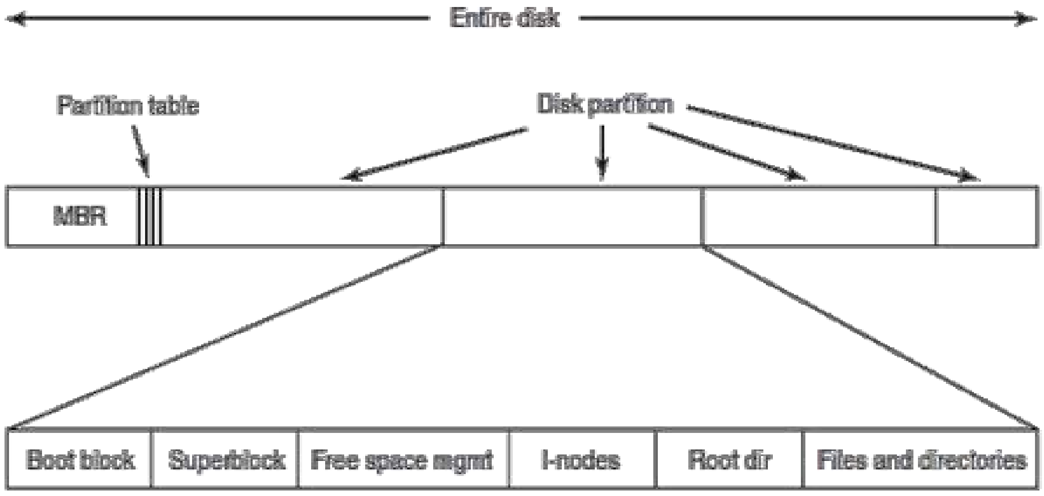


FILE SYSTEM IMPLEMENTATION



File system implementation defines how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably.

File-System Layout

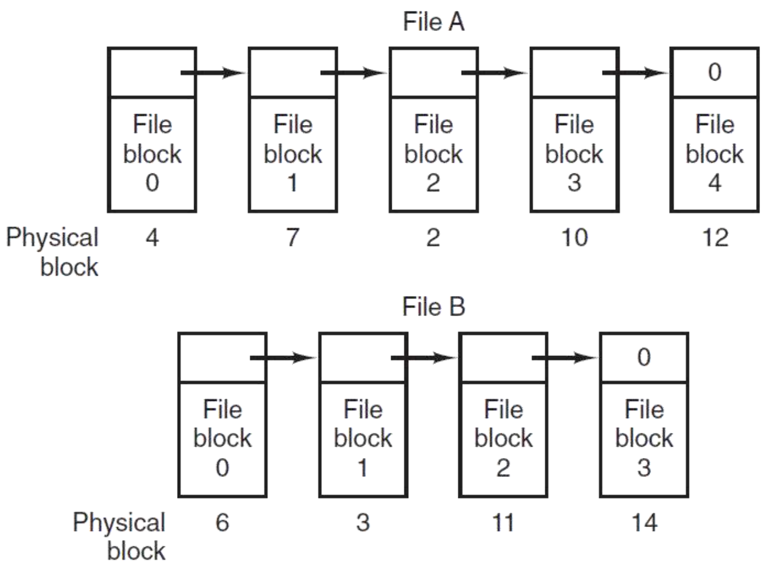


File Systems are stored on disks. The above figure depicts a possible File-System Layout.

- **MBR:** Master Boot Record is used to boot the computer
- **Partition Table:** Partition table is present at the end of MBR. This table gives the starting and ending addresses of each partition.
- **Boot Block:** When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, which is called the boot block, and execute it. The program in the boot block loads the operating system contained in that partition. Every partition contains a boot block at the beginning though it does not contain a bootable operating system.
- **Super Block:** It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched.

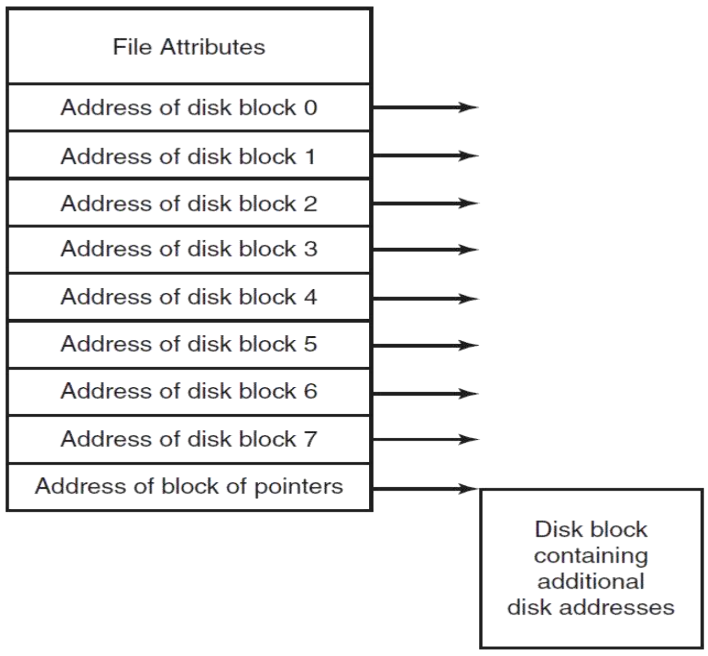
Implementing Files

- **Contiguous Allocation:**
  Each file is stored as a contiguous run of disk blocks.
  Example: On a disk with 1KB blocks, a 50KB file would be allocated 50 consecutive blocks. With 2KB blocks it would be 25 consecutive blocks.
  Each file begins at the start of a new block, so that if file A is occupying 3½ blocks, some space is wasted at the end of the last block.\
- **Advantages:**
  - Simple to implement.
  - The read performance is excellent because the entire file can be read from the disk in a single operation.
- **Drawbacks:**
  Over the course of time the disk becomes **fragmented**.
  - **Linked List Allocation:**
    The second method for storing files is to keep each one as a linked list of disk blocks. The first word of each block is used as a pointer to the next one. The rest of the block is for data. Unlike Contiguous allocation no space is lost in disk fragmentation.



Random access of a file is very slow.

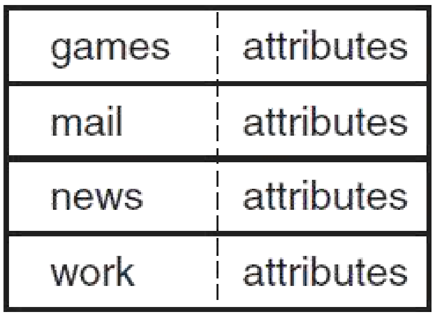
- **Linked-List Allocation Using a Table in Memory:**
  The disadvantage of linked list can be overcome by taking the pointer word from each disk block and putting it in a table in memory. Such a table in main memory is called a FAT (File Allocation Table). Using FAT random access can be made much easier.
  The primary disadvantage of this method is that the entire table must be in memory all the time to make it work.
- **I-nodes:** I-node is a data structure which is used to identify which block belongs to which file. It contains the attributes and disk addresses of the file's blocks. Unlike the in-memory table the i-node need to be in memory only when the corresponding file is open.



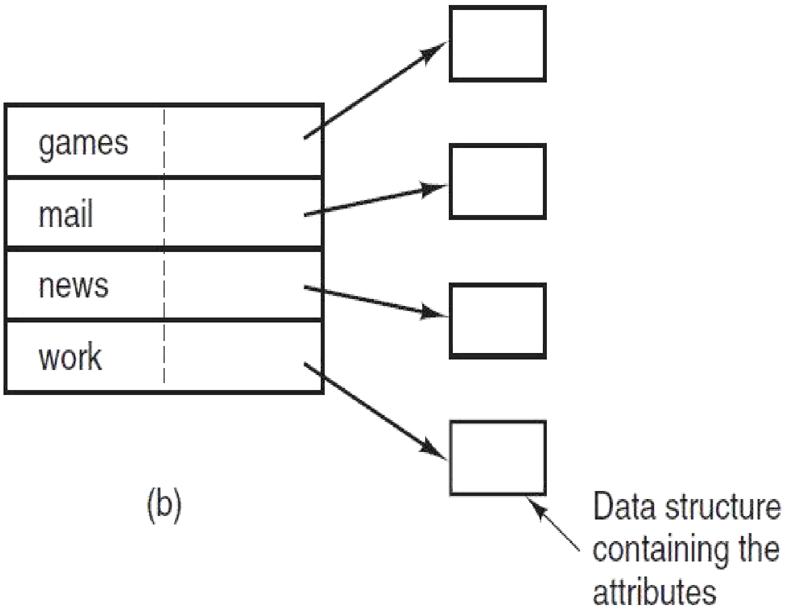
Implementing Directories

The main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data. A directory can be designed in two ways.

- In a simple design a directory consists of a list of fixed-size entries,one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses telling where the disk blocks are.



- For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number.



Shared Files

✓ When several users are working together sharing of files takes place. In such a case it is convenient for a file to appear simultaneously in different directories belonging to different users. Sharing files is convenient but also introduces problems like if the original file or the shared file are appended with new features then it will not be updated in both the copies of the file, it is updated only at its original location. This way the purpose of file sharing is defeated. This problem can be solved by two ways:

- i-node
- Symbolic linking

Log-Structured File Systems

A log-structured filesystem is a file system in which data and metadata are written sequentially to a circular buffer, called a log.

- The idea that drove the LFS design is that as CPU's and RAM's are getting faster and larger in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.
- Using LFS all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.
- Since eventually the log will occupy the entire disk, at which time no new segments can be written to the log. To deal with this problem, LFS has a cleaner thread that spends its time scanning the log circularly to compact it. In this the disk is a big circular buffer, with the writer thread adding new segments to the front and the **cleaner** thread removing old ones from the back.
- While Log-Structured File Systems are a good idea they are not widely used.

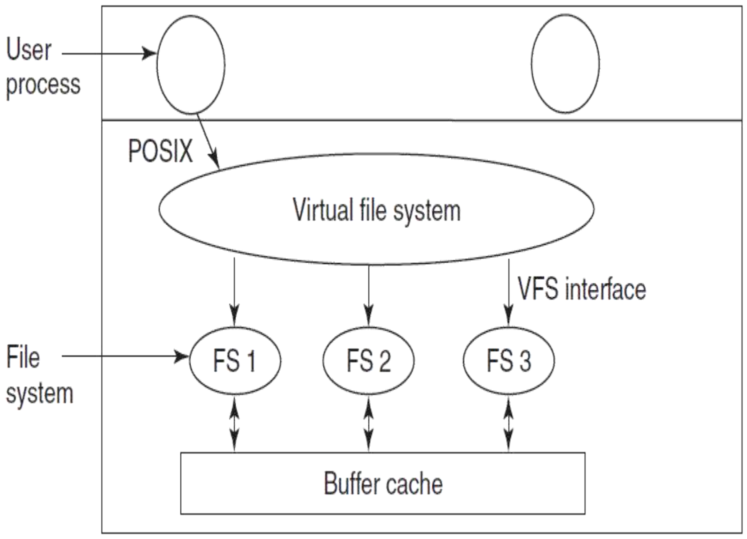
Journaling File Systems

✓ The basic idea here is to keep a log of what the file system is going to do before it does it, so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job. Such file systems, called journaling file systems, are actually in use.
Operations that take place for removing a file:
  - Remove the file from its directory
  - Release the i-node to the pool of free i-nodes
  - Return all the disk blocks to the pool of free disk blocks

✓ The functioning of Journaling file system is such that it first writes a log entry listing the three actions to be completed. The log entry is then written to disk. Only after the log entry has been written, the various operations begin. After the operations are completed successfully, the log entry is erased. If the system now crashes upon recovery the file system can check the log to see if any operations were pending. If so, all of them can be rerun (multiple times in the event of repeated crashes) until the file is correctly removed.
Crash recovery can be made fast and secure when log operations are **idempotent.**

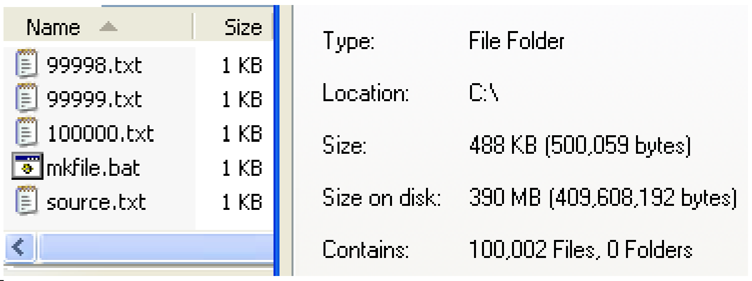
Virtual File Systems

- An operating system can have multiple file systems in it. **Virtual File Systems** are used to integrate multiple file systems into an orderly structure. The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file system to actually manage the data.
Structure of Virtual File Systems in UNIX system:



- The VFS also has a 'lower' interface to the concrete file systems, which is labeled **VFS interface**. This interface consists of several dozen function calls that the VFS can make to each file system to get work done. VFS has two distinct interfaces: the upper one to the user processes and the lower one to the concrete file systems VFS supports remote file systems using the **NFS (Network File System)** protocol.

FILE SYSTEM MANAGEMENT AND OPTIMIZATION



Name	Size
99998.txt	1 KB
99999.txt	1 KB
100000.txt	1 KB
mkfile.bat	1 KB
source.txt	1 KB

Type:	File Folder
Location:	C:\
Size:	488 KB (500,059 bytes)
Size on disk:	390 MB (409,608,192 bytes)
Contains:	100,002 Files, 0 Folders

Disk-Space Management

- Since all the files are normally stored on disk one of the main concerns of file system is management of disk space.

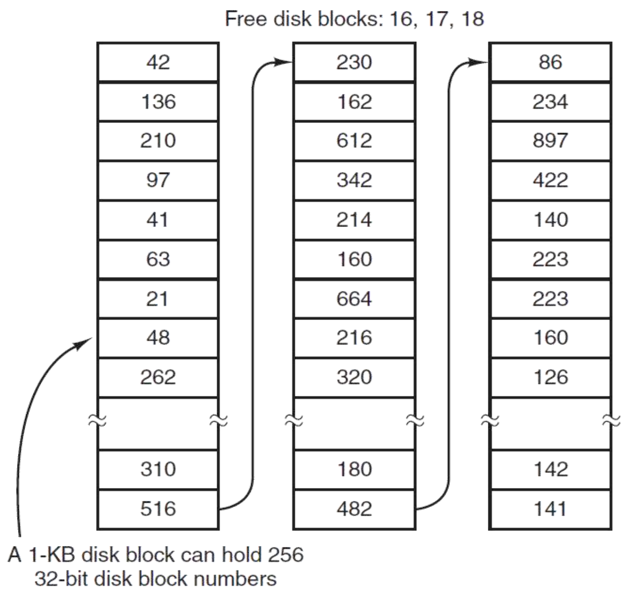
Block Size

- The main question that arises while storing files in a fixed-size blocks is the size of the block. If the block is too large space gets wasted and if the block is too small time gets waste. So, to choose a correct block size some information about the file-size distribution is required. Performance and space-utilization are always in conflict.

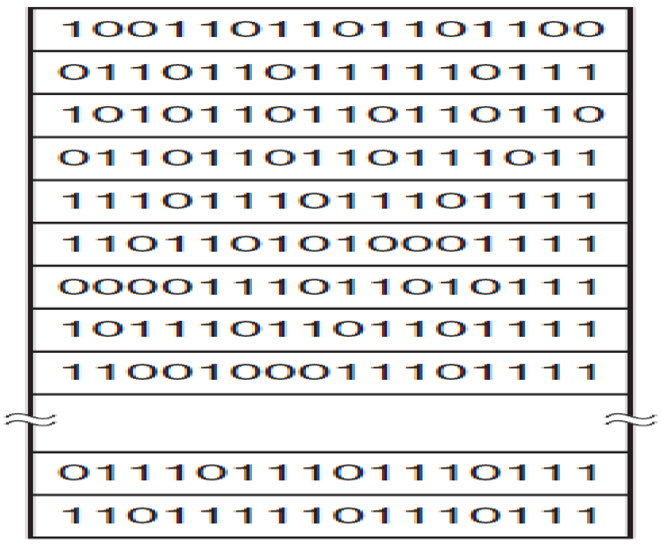
Keeping track of free blocks

After a block size has been finalized the next issue that needs to be catered is how to keep track of the free blocks. In order to keep track there are two methods that are widely used:

- Using a linked list: Using a linked list of disk blocks with each block holding as many free disk block numbers as will fit.



- Bitmap: A disk with n blocks has a bitmap with n bits. Free blocks are represented using 1's and allocated blocks as 0's as seen below in the figure.



Disk quotas

- Multiuser operating systems often provide a mechanism for enforcing disk quotas. A system administrator assigns each user a maximum allotment of files and blocks and the operating system makes sure that the users do not exceed their quotas. Quotas are kept track of on a per-user basis in a quota table.

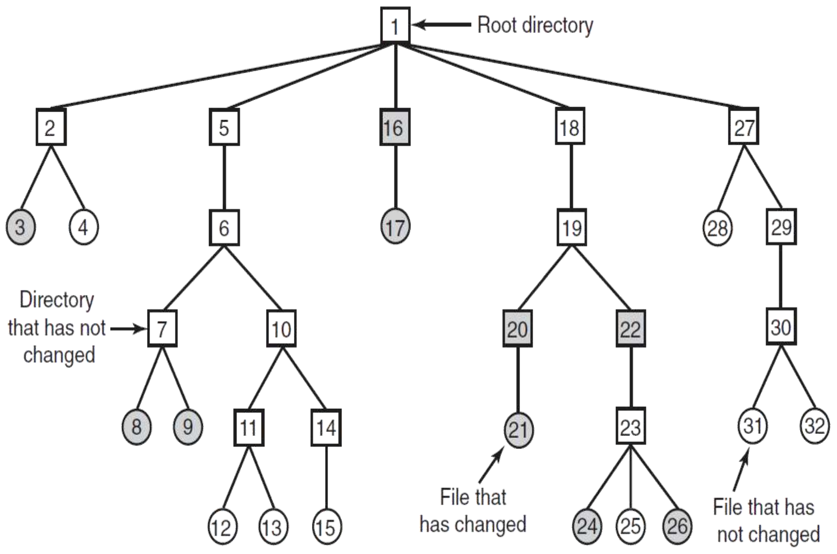
File-system Backups

- If a computer's file system is irrevocably lost, whether due to hardware or software restoring all the information will be difficult, time consuming and in many cases impossible. So it is adviced to always have file-system backups. Backing up files is time consuming and as well occupies large amount of space, so doing it efficiently and convenietly is important. Below are few points to be considered before **creating backups for files.**
- Is it requied to backup the entire file system or only a part of it.
- Backing up files that haven't been changed from previous backup leads to **incremental dumps**. So it's better to take a backup of only those files which have changed from the time of previous backup. But recovery gets complicated in such cases.

- Since there is immense amount of data, it is generally desired to compress the data before taking a backup for the same.
- It is difficult to perform a backup on an active file-system since the backup may be inconsistent.
- Making backups introduces many security issues

There are two ways for dumping a disk to the backup disk:

- **Physical dump:** In this way dump starts at block 0 of the disk, writes all the disk blocks onto thee output disk in order and stops after copying the last one. **Advantages:** Simplicity and great speed. **Disadvantages:** inability to skip selected directories, make incremental dumps, and restore individual files upon request
- **Logical dump:** In this way the dump starts at one or more specified directories and recursively dump all files and directories found that have been changed since some given base date. This is the most commonly used way.



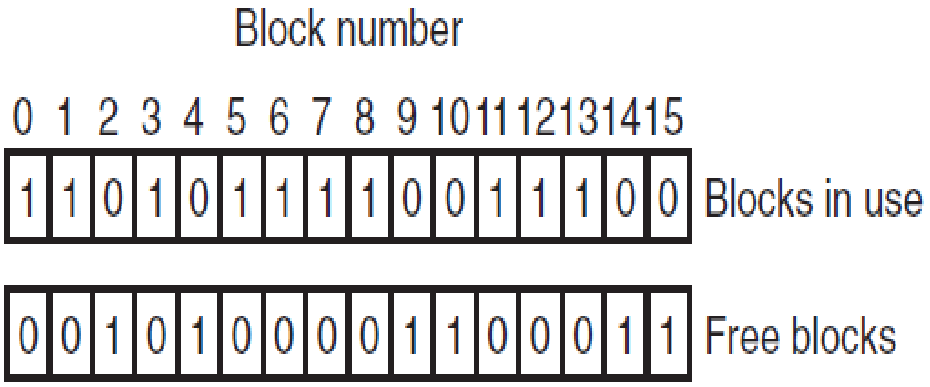
- The above figure depicts a popular algorithm used in many UNIX systems wherein squares depict directories and circles depict files. This algorith dumps all the files and directories that have been modified and also the ones on the path to a modified file or directory. The dump algorithm maintains a bitmap indexed by i-node number with several bits per i-node. Bits will be set and cleared in this map as the algorithm proceeds. Although logical dumping is straightforward, there are few issues associated with it.

- Since the free block list is not a file, it is not dumped and hence it must be reconstructed from scratch after all the dumps have been restored
- If a file is linked to two or more directories, it is important that the file is restored only one time and that all the directories that are supposed to point to it do so
- UNIX files may contain holes
- Special files, named pipes and all other files that are not real should never be dumped.

File-system Consistency

- To deal with inconsistent file systems, most computers have a utility program that checks file-system consistency. For example, UNIX has fsck and Windows has sfc. This utility can be run whenever the system is booted. The utility programs perform two kind consistency checks.

- Blocks: To check block consistency the program builds two tables, each one containing a counter for each block, initially set to 0. If the file system is consistent, each block will have a 1 either in the first table or in the second table as you can see in the figure below.


Block number
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0 Blocks in use
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1 Free blocks

▪ In case if both the tables have 0 in it that may be because the block is missing and hence will be reported as a missing block. The two other situations are if a block is seen more than once in free list and same data block is present in two or more files.

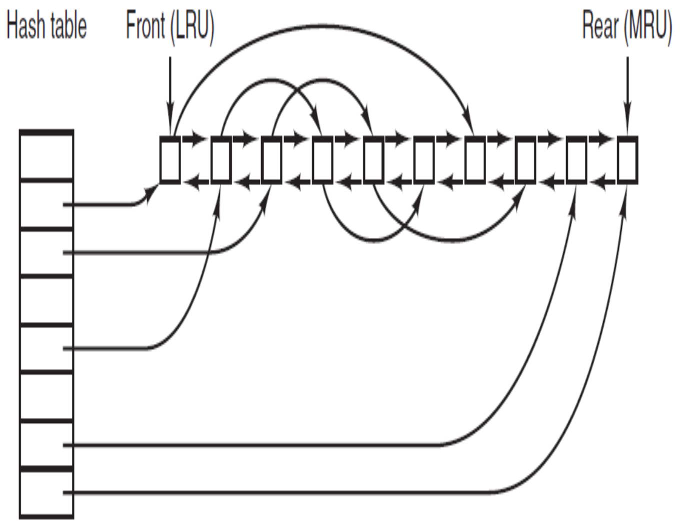
- In addition to checking to see that each block is properly accounted for, the file-system checker also checks the directory system. It too uses a table of counters but per file-size rather than per block. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree

File-system Performance

- Since the access to disk is much slower than access to memory, many file systems have been designed with various optimizations to improve performance as described below.

Caching

- The most common technique used to reduce disk access time is the block cache or buffer cache. Cache can be defined as a collection of items of the same type stored in a hidden or inaccessible place. The most common algorithm for cache works in such a way that if a disk access is initiated, the cache is checked first to see if the disk block is present. If yes then the read request can be satisfied without a disk access else the disk block is copied to cache first and then the read request is processed.



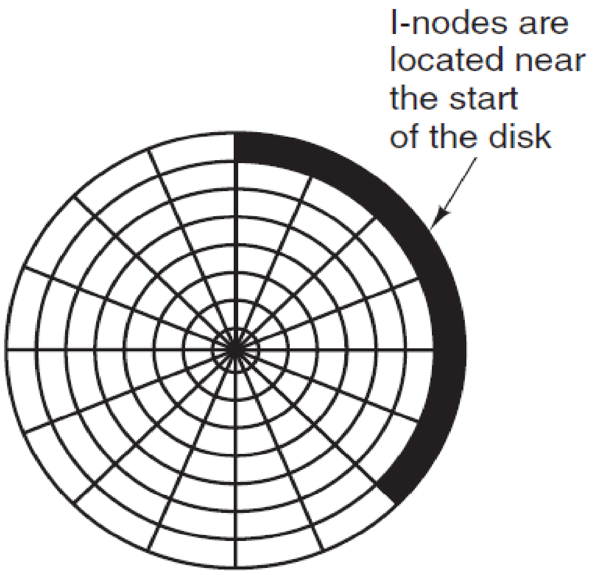
The above figure depicts how to quickly determine if a block is present in a cache or not. For doing so a hash table can be implemented and look up the result in a hash table.

Block Read Ahead

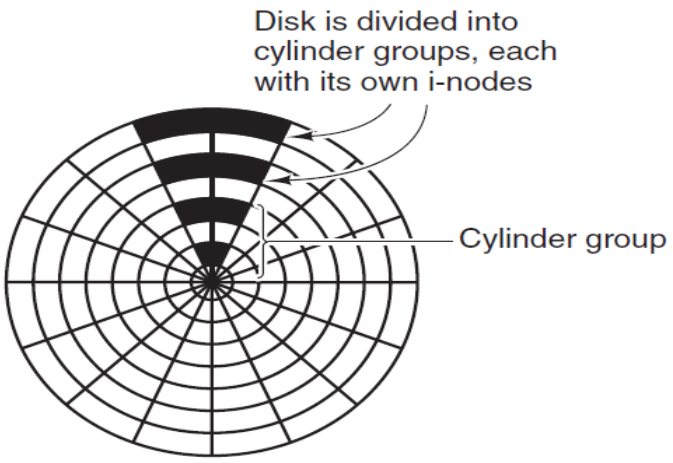
- Another technique to improve file-system performance is to try to get blocks into the cache before they are needed to increase the hit rate. This works only when files are read sequentially. When a file system is asked for block 'k' in the file it does that and then also checks before hand if 'k+1' is available if not it schedules a read for the block k+1 thinking that it might be of use later.

Reducing disk arm motion

- Another way to increase file-system performance is by reducing the disk-arm motion by putting blocks that are likely to be accessed in sequence close to each other,preferably in the same cylinder.



- In the above figure all the i-nodes are near the start of the disk, so the average distance between an inode and its blocks will be half the number of cylinders, requiring long seeks. But to increase the performance the placement of i-nodes can be modified as below.



Defragmenting Disks

- Due to continuous creation and removal of files the disks get badly fragmented with files and holes all over the place. As a consequence, when a new file is created, the blocks used for it may be spread all over the disk, giving poor performance. The performance can be restored by moving files around to make them contiguous and to put all (or at least most) of the free space in one or more large contiguous regions on the disk

………………… END OF THE UNIT V …………………….