Introduction to Operating Systems

1.Overview Of Operating Systems

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, Windows, OS X, 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

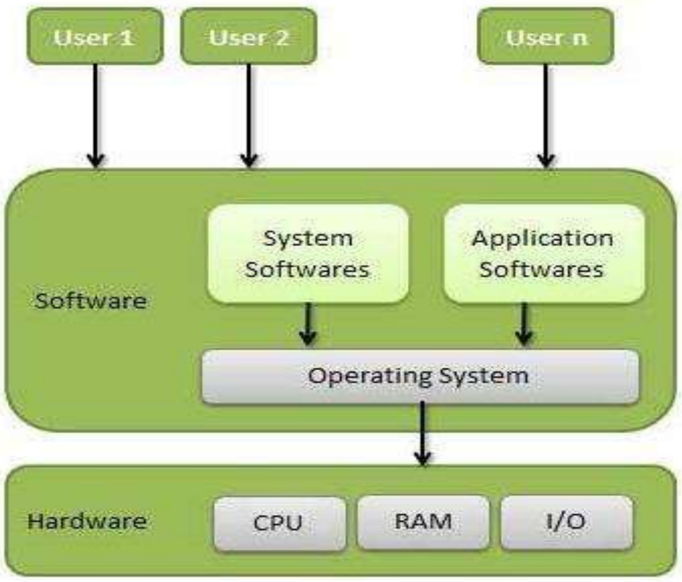


Figure1.1 Components of an operating system

1.1 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.

1.2 Operating System Generations

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

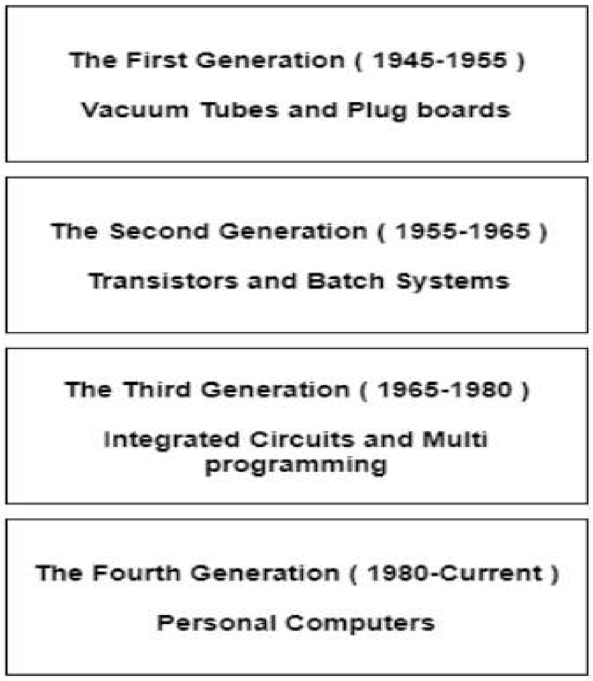

The First Generation (1945-1955)
Vacuum Tubes and Plug boards
The Second Generation (1955-1965)
Transistors and Batch Systems
The Third Generation (1965-1980)
Integrated Circuits and Multi programming
The Fourth Generation (1980-Current)
Personal Computers

Figure 1.2 Operating System Generations

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

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

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

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

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

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

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

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

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

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

The Fourth Generation (1980 - Present): Personal Computers

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

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

1.3 Types of operating systems

Batch operating system

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

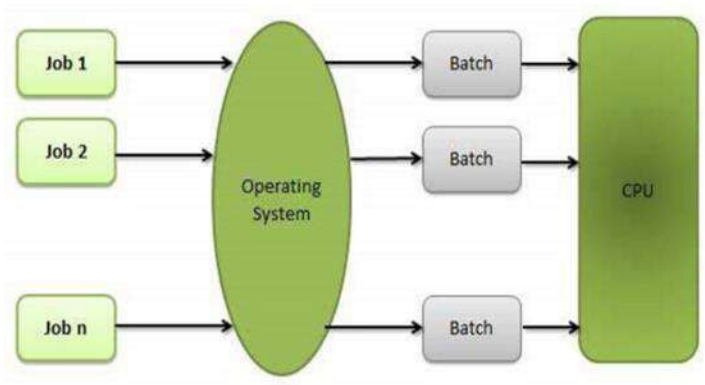


Figure1.3 Batch Processing

The problems with Batch Systems are as follows −

- Lack of interaction between the user and the job.
- CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
- Difficult to provide the desired priority.

Time-sharing operating systems

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if **n** users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows −

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

Disadvantages of Time-sharing operating systems are as follows −

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

Distributed operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.
The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows −

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

Network operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows −

- Centralized servers are highly stable.
- Security is server managed.
- Upgrades to new technologies and hardware can be easily integrated into the system.
- Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows −

- High cost of buying and running a server.
- Dependency on a central location for most operations.
- Regular maintenance and updates are required.

Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

Hard real-time systems

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

Soft real-time systems

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc.

OS structure and strategies

1.4 SERVICES OF OS

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

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

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

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

Program execution

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

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

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

I/O Operation

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

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

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

File system manipulation

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

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

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

Communication

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

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

- Two processes often require data to be transferred between them
- Both the processes can be on one computer or on different computers, but are connected through a computer network.
- Communication may be implemented by two methods, either by Shared Memory or by Message Passing.

Error handling

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

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

Resource Management

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

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

Protection

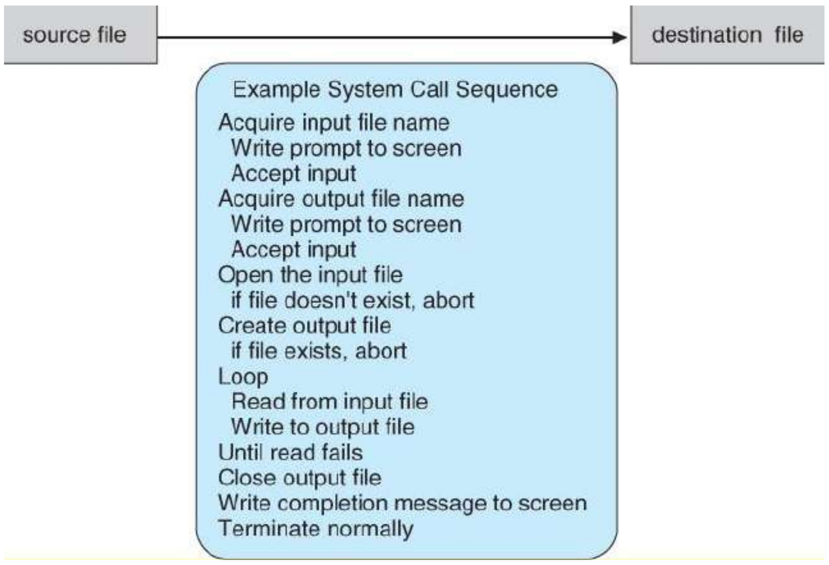
Considering a computer system having multiple users and concurrent execution of multiple processes, the various processes must be protected from each other's activities.

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

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

1.5 System Calls

- System calls provide a means for user or application programs to call upon the services of the operating system.
- Generally written in C or C++, although some are written in assembly for optimal performance.
- Figure 2.4 illustrates the sequence of system calls required to copy a file:



```
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
```

Figure 1.5 - Example of how system calls are used.

- The use of APIs instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the **system call interface**, using a table lookup to access specific numbered system calls, as shown in Figure :

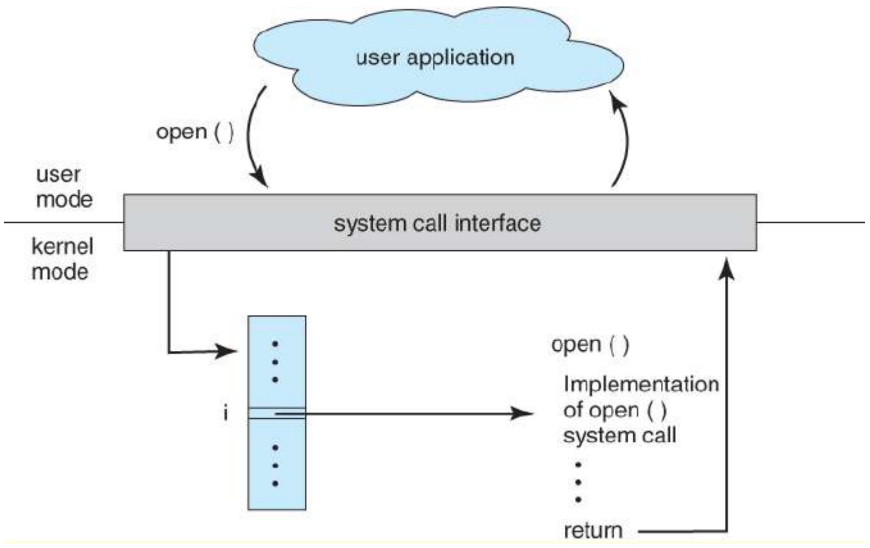


Figure 1.5.1 - The handling of a user application invoking the open( ) system call

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

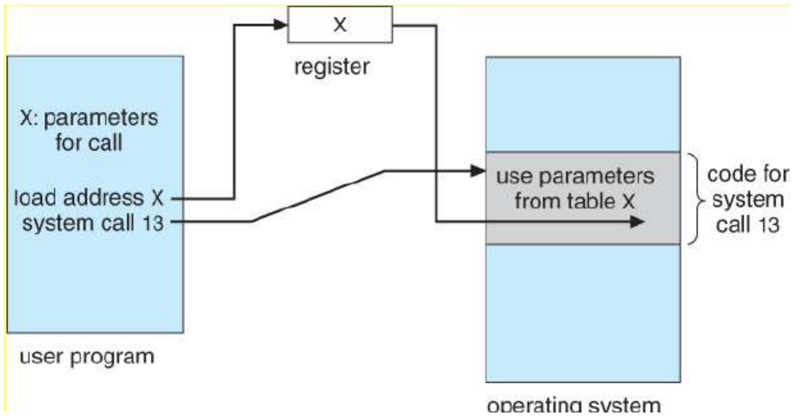


Figure 1.5.2 - Passing of parameters as a table

1.5.1Types of System Calls

Six major categories, as outlined in Figure and the following six subsections:

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

Process Control

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed,and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed

- When processes stop abnormally it may be necessary to provide core dumps and/or other diagnostic or recovery tools.
- Compare DOS (a single-tasking system) with UNIX (a multi-tasking system).
  - When a process is launched in DOS, the command interpreter first unloads as much of itself as it can to free up memory, then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure

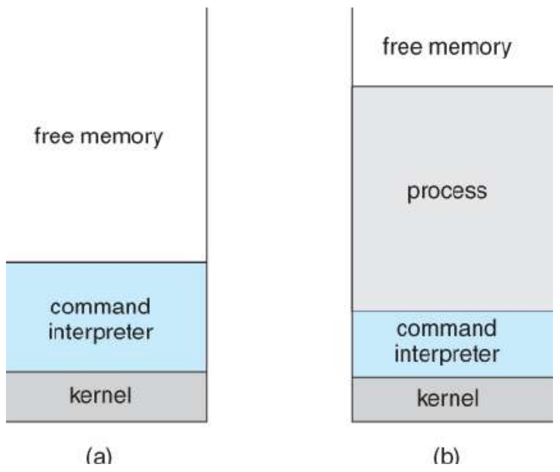


Figure 1.5.3 - MS-DOS execution. (a) At system startup. (b) Running a program.

Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure below.

- The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.
- In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate (clone) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
- The child process then executes an "exec" system call, which replaces its code with that of the desired process.
- The parent (command interpreter) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. (The child is then said to be running "in the background", or "as a background process".)

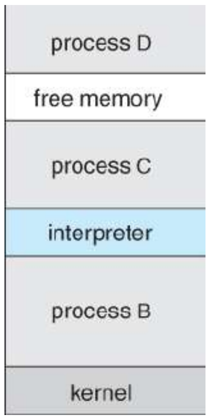


Figure 1.5.4 - FreeBSD running multiple programs

File Management

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

Device Management

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

Information Maintenance

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

Communication

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing** model must support calls to:
    - Identify a remote process and/or host with which to communicate.
    - Establish a connection between the two processes.
    - Open and close the connection as needed.
    - Transmit messages along the connection.

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

1.6 System Programs

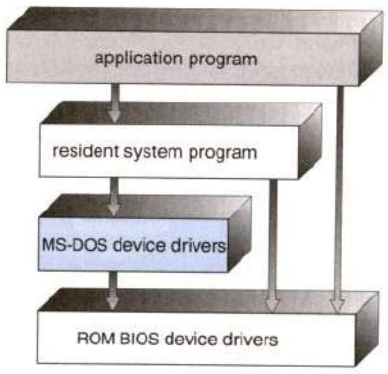
- System programs provide OS functionality through separate applications, which are not part of the kernel or command interpreters. They are also known as system utilities or system applications.
- Most systems also ship with useful applications such as calculators and simple editors, (e.g. Notepad). Some debate arises as to the border between system and non-system applications.
- System programs may be divided into these categories:
    - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
    - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
    - **File modification** - e.g. text editors and other tools which can change file contents.
    - **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages.
    - **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.
    - **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.
    - **Background services** - System daemons are commonly started when the system is booted, and run for as long as the system is running, handling necessary services. Examples include network daemons, print servers, process schedulers, and system error monitoring services.
- Most operating systems today also come complete with a set of **application programs** to provide additional services, such as copying files or checking the time and date.
- Most users' views of the system is determined by their command interpreter and the application programs. Most never make system calls, even through the API, (with the exception of simple (file) I/O in user-written programs.)

1.7 Operating-System Structure

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

1.7.1 Simple Structure

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



In MS-DOS, applications may bypass the operating system.

Figure 1.7.1 MS-DOS Structure

- Operating systems such as MS-DOS and the original UNIX did not have well-defined structures.
- There was no <u>CPU Execution Mode</u> (user and kernel), and so errors in applications could cause the whole system to crash.

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

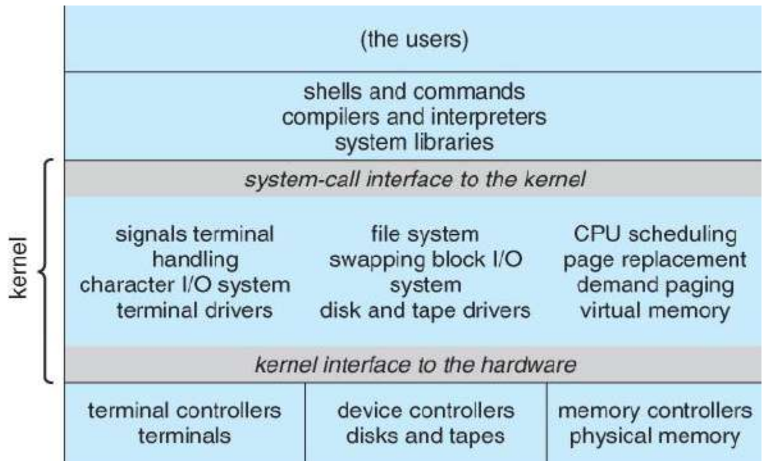

(the users)
shells and commands compilers and interpreters system libraries
system-call interface to the kernel
signals terminal handling character I/O system terminal drivers
file system swapping block I/O system disk and tape drivers
CPU scheduling page replacement demand paging virtual memory
kernel interface to the hardware
terminal controllers terminals
device controllers disks and tapes
memory controllers physical memory
kernel

Figure- Traditional UNIX system structure

1.7.2 Layered Approach

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

- The main *advantage* is simplicity of construction and debugging.
- The main *difficulty* is defining the various layers.
- The main *disadvantage* is that the OS tends to be less efficient than other implementations.

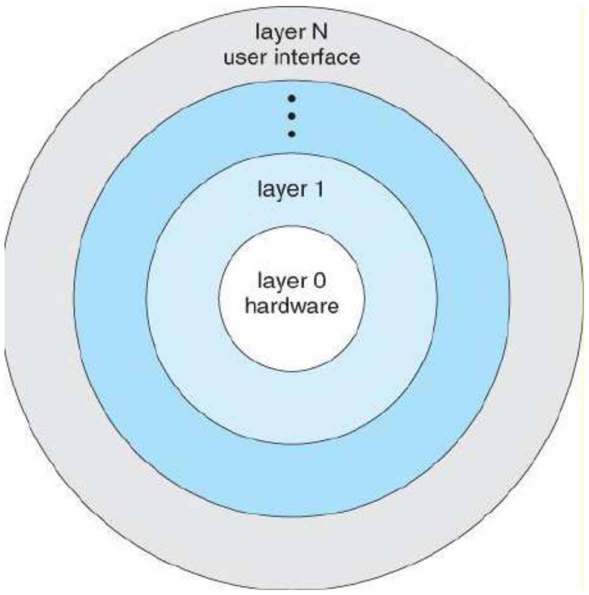


Figure 1.7.2 - A layered operating system

1.7.3 Microkernels

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

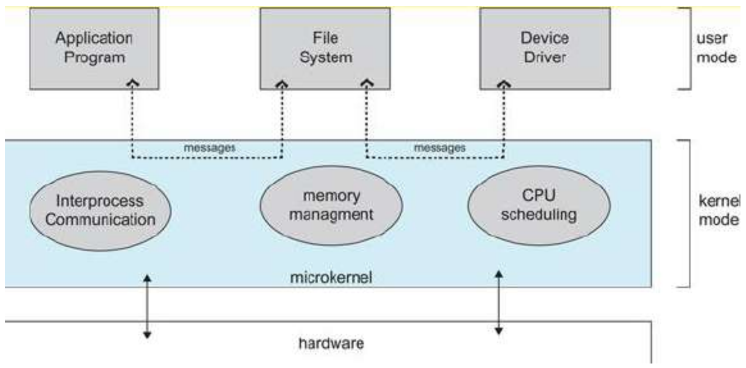


Figure 1.7.3 - Architecture of a typical microkernel

1.7.4 Monolithic Kernel

Apart from microkernel, Monolithic Kernel is another classification of Kernel. Like microkernel this one also manages system resources between application and hardware, but user services and kernel services are implemented under same address space. It increases the size of the kernel, thus increases size of operating system as well.

This kernel provides CPU scheduling, memory management, file management and other operating system functions through system calls. As both services are implemented under same address space, this makes operating system execution faster.

Below is the diagrammatic representation of Monolithic Kernel:

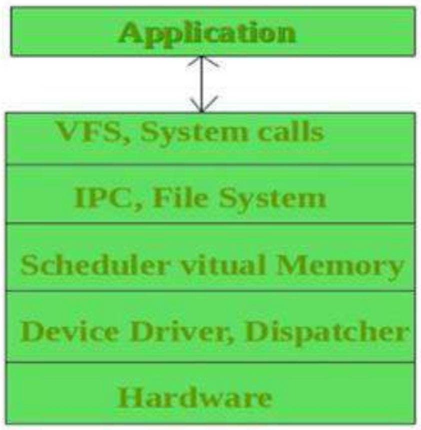


Figure 1.7.4 Monolithic Kernel

If any service fails the entire system crashes, and it is one of the drawbacks of this kernel. The entire operating system needs modification if user adds a new service.

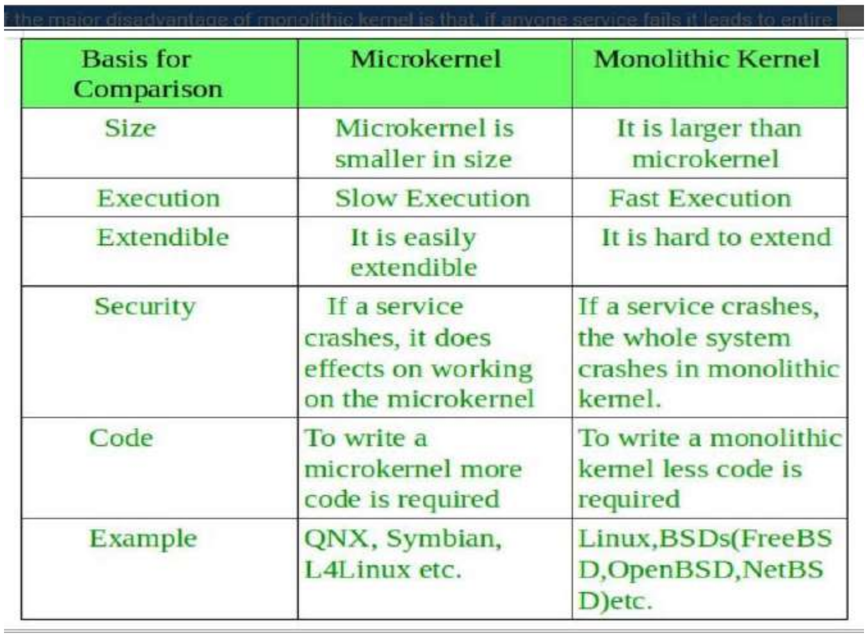
Advantages of Monolithic Kernel –

- One of the major advantage of having monolithic kernel is that it provides CPU scheduling, memory management, file management and other operating system functions through system calls.
- The other one is that it is a single large process running entirely in a single address space.
- It is a single static binary file. Example of some Monolithic Kernel based OSs are: Unix, Linux, Open VMS, XTS-400, z/TPF.

Disadvantages of Monolithic Kernel –

- One of the major disadvantage of monolithic kernel is that, if anyone service fails it leads to entire system failure.
- If user has to add any new service. User needs to modify entire operating system.

Key differences between Monolithic Kernel and Microkernel –



the major disadvantage of monolithic kernel is that, if anyone service fails it leads to entire

Basis for Comparison	Microkernel	Monolithic Kernel
Size	Microkernel is smaller in size	It is larger than microkernel
Execution	Slow Execution	Fast Execution
Extendible	It is easily extendible	It is hard to extend
Security	If a service crashes, it does effects on working on the microkernel	If a service crashes, the whole system crashes in monolithic kernel.
Code	To write a microkernel more code is required	To write a monolithic kernel less code is required
Example	QNX, Symbian, L4Linux etc.	Linux,BSDs(FreeBSD,OpenBSD,NetBSD)etc.

1.7.5 Modules

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

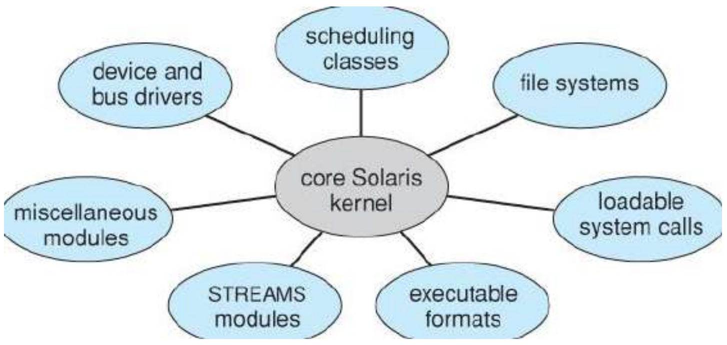


Figure 1.7.5- Solaris loadable modules

1.7.6 Hybrid Systems

- Most OSes today do not strictly adhere to one architecture, but are hybrids of several.

1.7.6.1 Mac OS X

- The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules (kernel extensions) provide the rest of the OS functionality:

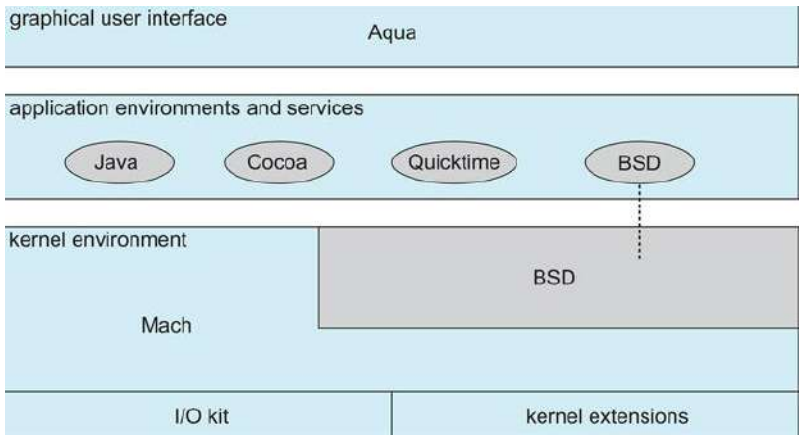


Figure 1.7.6.1 - The Mac OS X structure

1.7.6.2 iOS

- The iOS operating system was developed by Apple for iPhones and iPads. It runs with less memory and computing power needs than Max OS X, and supports touchscreen interface and graphics for small screens:

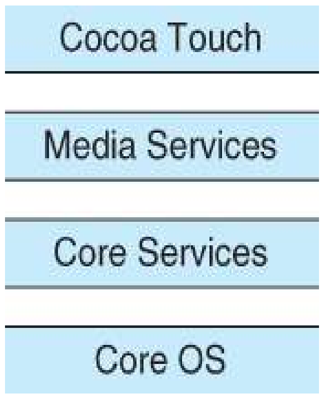


Figure 1.7.6.2 - Architecture of Apple's iOS.

1.7.6.3 Android

- The Android OS was developed for Android smartphones and tablets by the Open Handset Alliance, primarily Google.
- Android is an open-source OS, as opposed to iOS, which has lead to its popularity.
- Android includes versions of Linux and a Java virtual machine both optimized for small platforms.
- Android apps are developed using a special Java-for-Android development environment.

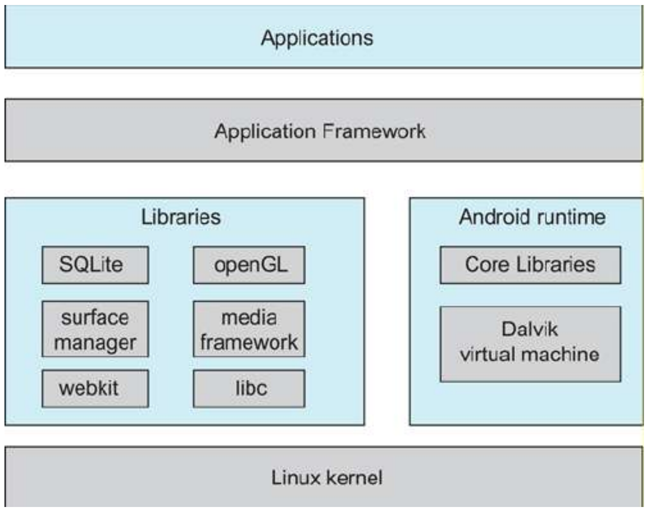


Figure 1.8.6.3 - Architecture of Google's Android

1.8 Virtual Machine

Virtual machines (VMs) allow you to run other operating systems within your current OS. The virtual OS will run as if it's just another program on your computer.

This is ideal for testing out other operating systems, like Windows 10 or alternative Linux operating systems. You can also use virtual machines to run software on operating systems it wasn't designed for. For example, you can run Windows programs on a Mac or run multiple copies of an app on a Mac with a virtual machine.

What Is A Virtual Machine?

A virtual machine is a program that acts as a virtual computer. It runs on your current operating system (the host operating system) and provides virtual hardware to guest operating systems. The guest OS runs in a window on your host OS, just like any other program on your computer.

The virtual machine's emulation engine, called a hypervisor, handles the virtual hardware, including a CPU, memory, hard drive, network interface, and other devices. The virtual hardware devices provided by the hypervisor map to real hardware on your physical machine. For example, a virtual machine's virtual hard disk is stored in a file located on your hard drive.

Virtual machines are divided into two categories based on their use and correspondence to real machine: System virtual machine and Process virtual machine. First category provides a complete system platform that executes complete operating system, second one will run a single program.

You can have several virtual machines installed on your system. You're only limited by the amount of storage you have available for them. Once you've installed several operating systems, you can open your virtual machine program and choose which virtual machine you want to boot. The guest operating system starts up and runs in a window on your host operating system, although you can also run it in full-screen mode.

Virtualization brings you a number of advantages -centralizing network management, reducing dependency on additional hardware and software,etc. But as it is always the case,it has certain shortcoming's too.

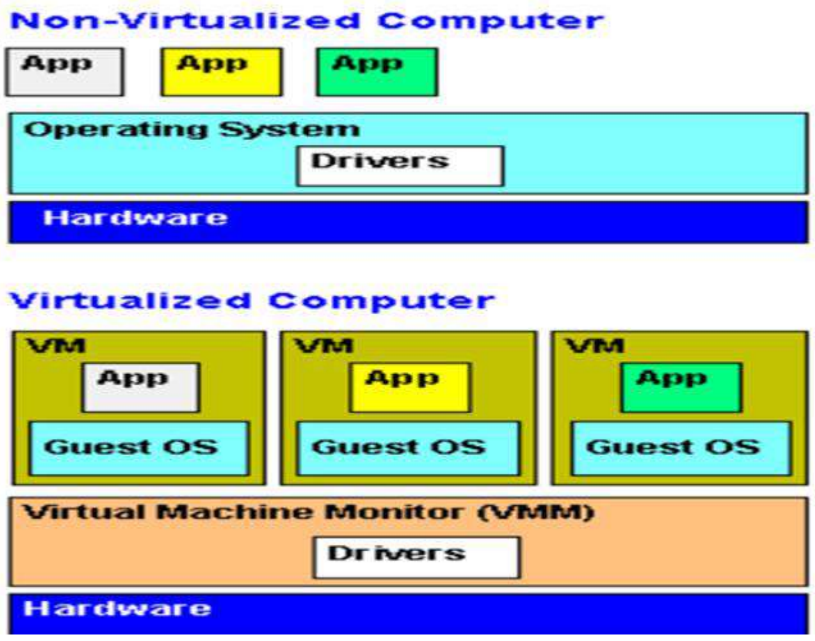


Figure1.8 virtual machine

Virtual machines have a number of popular uses

Test new versions of operating systems: You can try out Windows 10 on your Windows 7 computer if you aren't willing to upgrade yet.

Experiment with other operating systems: Installing various distributions of Linux in a virtual machine lets you experiment with them and learn how they work. And running macOS on Windows 10 in a virtual machine lets you get used to a different operating system you're considering using full-time.

Run software designed for another operating systems: Mac and Linux users can run Windows in a virtual machine to use Windows software on their computers without compatibility headache. Unfortunately, games are a problem. Virtual machine programs introduce overhead and 3D games will not run smoothly in a VM.

Use software requiring an outdated operating system: If you’ve got an important application that only runs on Windows XP, you can install XP in a virtual machine and run the application there. This allows you to use an application that only works with Windows XP without actually installing it on your computer. This is important since Windows XP no longer receives support from Microsoft.

Recommended Virtual Machine Software

VirtualBox is a great, open-source application that runs on Windows, macOS, and Linux. One of the best parts about VirtualBox is that there's no commercial version. This means you get all features for free, including advanced features like snapshots. This allows you to save a virtual machine's state and revert to that state in the future, which is great for testing.

VMware Player is another well-known VM program for Windows and Linux. VMware Player is the free counterpart to VMware Workstation, a commercial application, so you don't get all the advanced features you would with VirtualBox.

However, both VirtualBox and VMware Player are solid programs that offer the basic features free. If one of them doesn't work for you, try the other

Advantages:

1. There are no protection problems because each virtual machine is completely isolated from all other virtual machines.
2. Virtual machine can provide an instruction set architecture that differs from real computers.
3. Easy maintenance, availability and convenient recovery.

Disadvantages:

1. When multiple virtual machines are simultaneously running on a host computer, one virtual machine can be affected by other running virtual machines, depending on the workload.
2. Virtual machines are not as efficient as a real one when accessing the hardware.

2.Process Concepts

2.1.1 Definition

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system. There are two categories of processes in Unix, namely

- **User processes**: They are operated in user mode.
- **Kernel processes**: They are operated in kernel mode.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections – stack, heap, text and data. The following image shows a simplified layout of a process inside main memory –

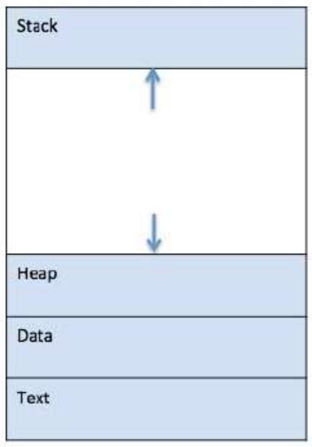


Figure2.1 Process

1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

Program

A computer program is a collection of instructions that performs a specific task when executed by a computer. When we compare a program with a process, we can conclude that a process is a dynamic instance of a computer program.

A part of a computer program that performs a well-defined task is known as an **algorithm**. A collection of computer programs, libraries and related data are referred to as a **software**.

2.1.2 Process Relationships

Parent can run concurrently with child, or wait for completion. Child may share all (fork/join) or part of parent's variables. Death of parent may force death of child. Processes are static (never terminate) or dynamic (can terminate).

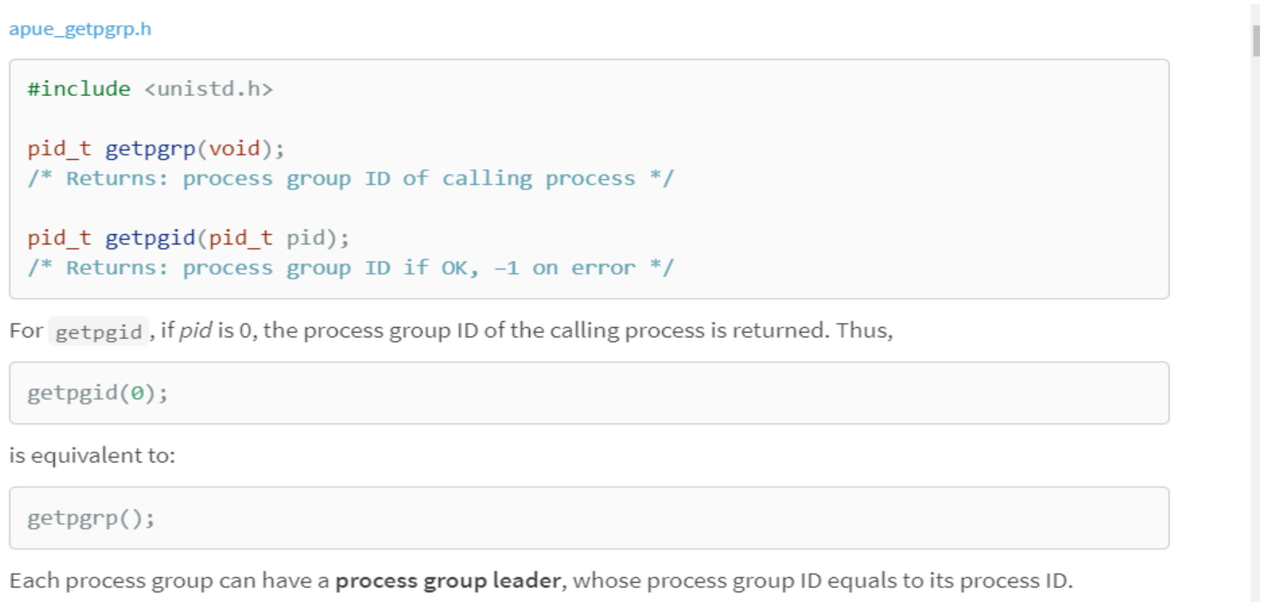
Independent Execution is deterministic and reproducible. Execution can be stopped/ started without affecting other processes. Cooperating Execution depends on other processes or is time dependent. Here the same inputs won't always give the same outputs; the process depends on other external states.

Process Groups

In addition to having a process ID, each process belongs to a **process group**.

- A process group is a collection of one or more processes (usually associated with the same job) that can receive signals from the same terminal.
- Each process group has a unique process group ID. Process group IDs are similar to process IDs: they are positive integers and can be stored in a pid_t data type.

The function getpgrp returns the process group ID of the calling process. The getpgid function took a *pid* argument and returned the process group for that process.


apue_getpgrp.h

```
#include <unistd.h>

pid_t getpgrp(void);
/* Returns: process group ID of calling process */

pid_t getpgid(pid_t pid);
/* Returns: process group ID if OK, −1 on error */
```

For `getpgid`, if *pid* is 0, the process group ID of the calling process is returned. Thus,

```
getpgid(0);
```

is equivalent to:

```
getpgrp();
```

Each process group can have a **process group leader**, whose process group ID equals to its process ID.

2.1.3 Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.

In general, a process can have one of the following five states at a time.

S.No	State & Description
1	**Start** This is the initial state when a process is first started/created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start**state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the

	operating system, it is moved to the terminated state where it waits to be removed from main memory.

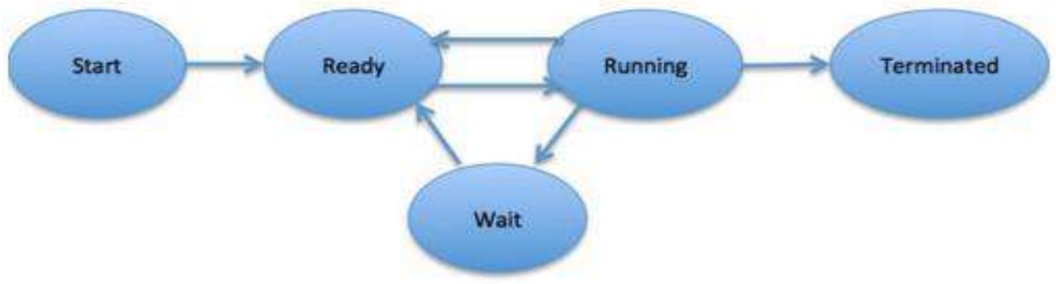


Figure2.2 Process Life Cycle

- The state diagram for a process captures its life-cycle. The states represent the execution status of the process; the transitions represent changes of execution state.

- Each active process has its own execution status, so there is a state diagram for each process. There are relationships between the states of various processes that are maintained by the operating system.

2.1.4 Process State

- Processes may be in one of 5 states, as shown in Figure 3.2 below.
  - **New** - The process is in the stage of being created.
  - **Ready** - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
  - **Running** - The CPU is working on this process's instructions.
  - **Waiting** - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
  - **Terminated -** The process has completed.
- The load average reported by the "w" command indicate the average number of processes in the "Ready" state over the last 1, 5, and 15 minutes, i.e. processes who have everything they need to run but cannot because the CPU is busy doing something else.
- Some systems may have other states besides the ones listed here.

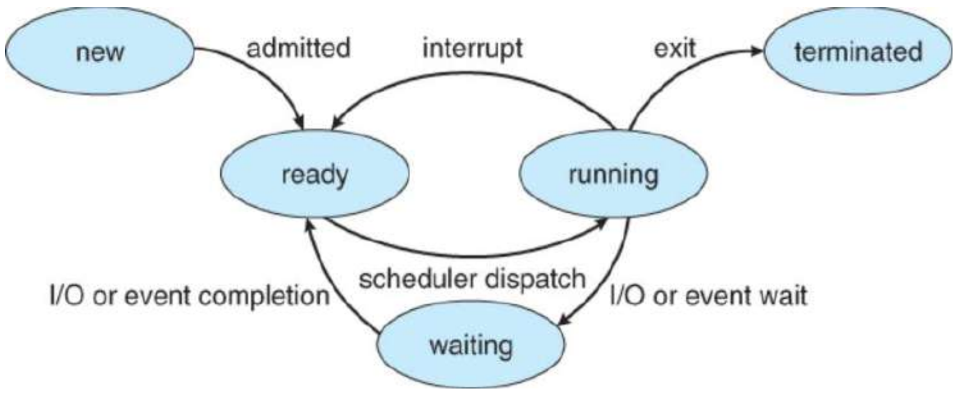


Figure Diagram of process State

2.1.5 Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. A simplified overview of UNIX process states and the transitions between states is shown in the following figure.

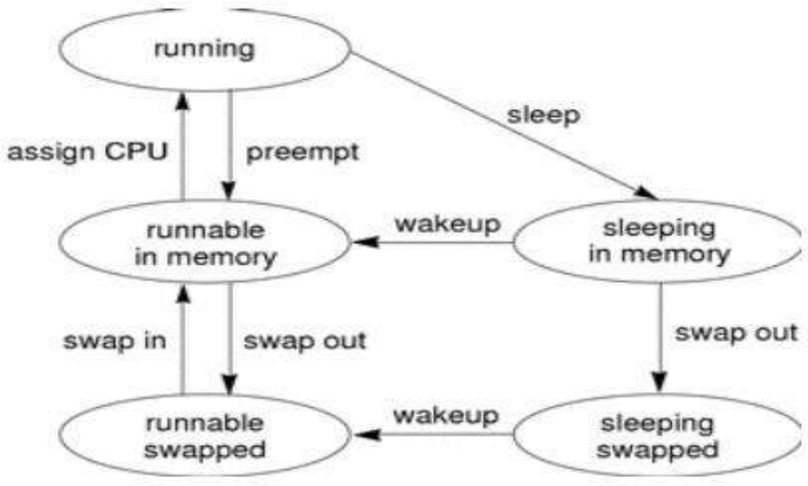


Figure Process State Transition Diagram

An active process is normally in one of the five states in the diagram. The arrows show how the process changes states.

- A process is running if the process is assigned to a CPU. A process is removed from the running state by the scheduler if a process with a higher priority becomes runnable. A process is also pre-empted if a process of equal priority is runnable when the original process consumes its entire time slice.
- A process is runnable in memory if the process is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if the process is in primary memory but is waiting for a specific event before continuing execution. For example, a process sleeps while waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, a wakeup call is sent to the process. If the reason for its sleep is gone, the process becomes runnable.
- When a process' address space has been written to secondary memory, and that process is not waiting for a specific event, the process is runnable and swapped.
- If a process is waiting for a specific event and has had its whole address space written to secondary memory, the process is sleeping and swapped.

If a machine does not have enough primary memory to hold all its active processes, that machine must page or swap some address space to secondary memory.

- When the system is short of primary memory, the system writes individual pages of some processes to secondary memory but leaves those processes runnable. When a running process, accesses those pages, the process sleeps while the pages are read back into primary memory.
- When the system encounters a more serious shortage of primary memory, the system writes all the pages of some processes to secondary memory. The system marks the pages that have been written to secondary memory as swapped. Such processes can only be scheduled when the system scheduler daemon selects these processes to be read back into memory.

Both paging and swapping cause delay when a process is ready to run again. For processes that have strict timing requirements, this delay can be unacceptable.

2.1.6 Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table –

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB –

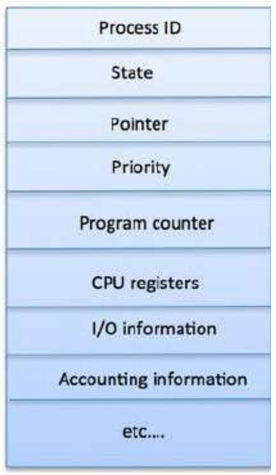


Figure Process Control Block (PCB)

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues −

- **Job queue** − This queue keeps all the processes in the system.
- **Ready queue** − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.
- **Device queues** − The processes which are blocked due to unavailability of an I/O device constitute this queue.

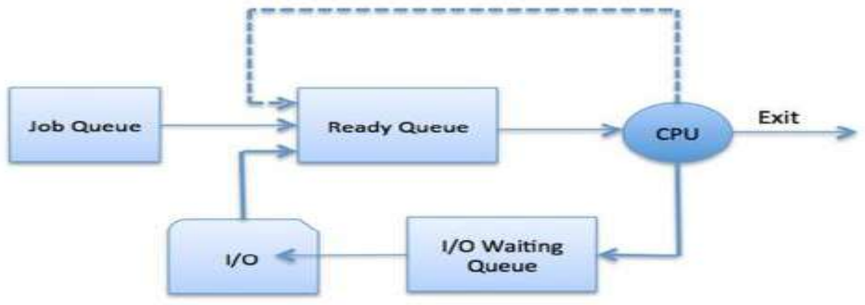


Figure 2.4 Process Scheduling Queues

The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types −

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time	It is also minimal in time sharing system	It is a part of Time sharing systems.

	sharing system		
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

2.1.7 Context Switch

A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

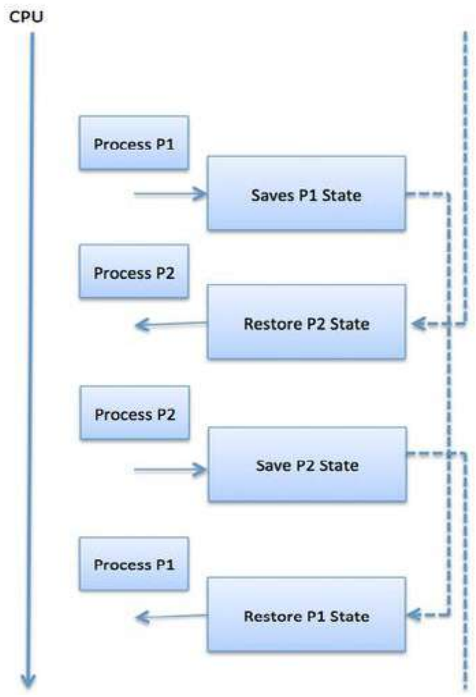


Figure 2.5 Context Switching

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use.

- Program Counter

- Scheduling information
- Base and limit register value
- Currently used register
- Changed State
- I/O State information
- Accounting information

3 Multithreading

3.1 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.

Example: MS Word uses multiple threads: one thread to format the text, another thread to process input etc…

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.

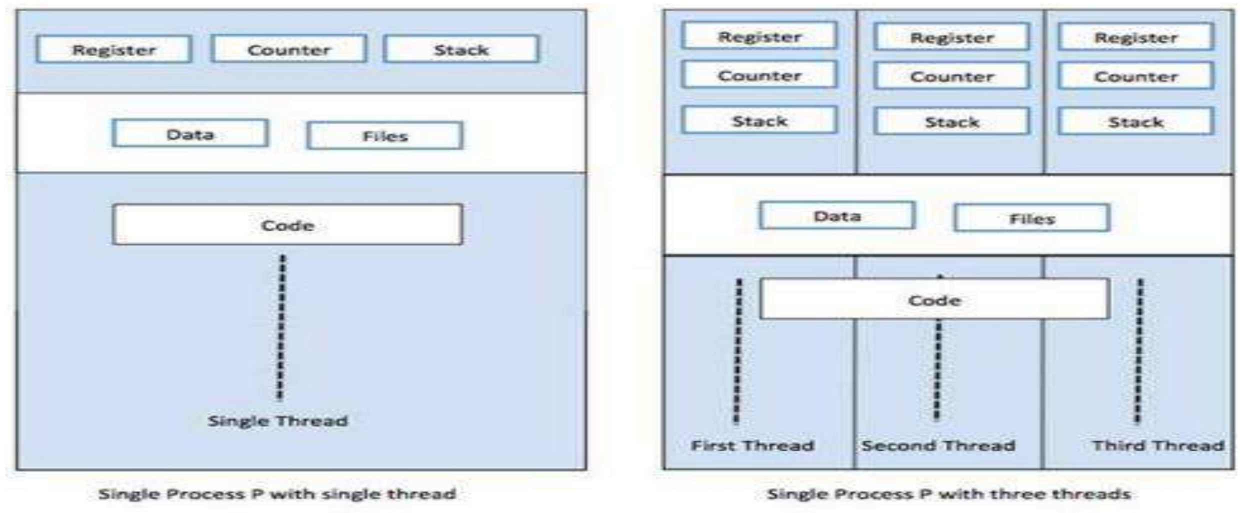


Figure 3.1 (a) Single Thread (b)Multithreading

3.2 Thread State

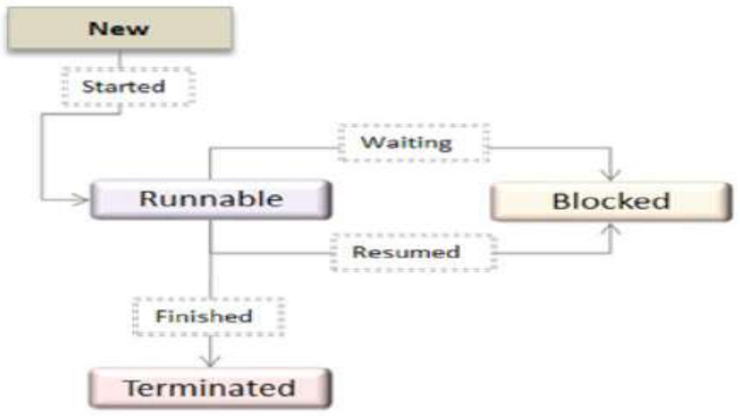


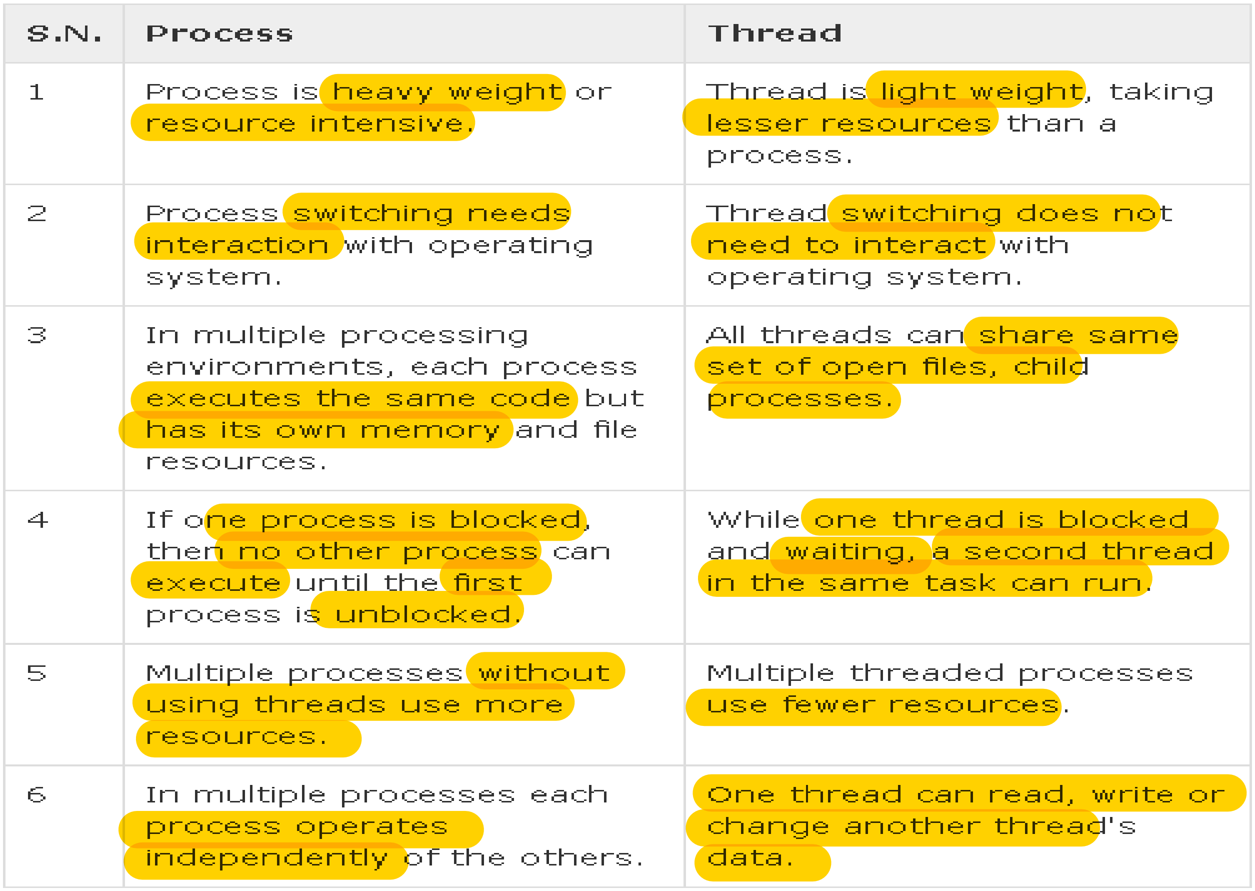
Figure Thread State Diagram

A thread is in the *new* state once it has been created. It doesn't take any CPU resources until it's actually running. Now, the process is taking up CPU resources because it's ready to run. However, it's in a *runnable* state because it could be waiting for another thread to run and so it has to wait for its turn.

A thread which is not allowed to continue remains in a *blocked* state. Let's say that a thread is waiting for input/output (I/O), but it never gets those resources, so it will remain in a blocked state. blocked thread won't use CPU resources. The thread isn't stopped forever. For example, if you allow emergency vehicles to pass, it does not mean that you are forever barred from your final destination. Similarly, those threads (emergency vehicles) that have a higher priority are processed ahead of you. If a thread becomes blocked, another thread moves to the front of the line.

Finally, a thread is *terminated* if it finishes a task successfully or abnormally. At this point, no CPU resources are used.

Difference between Process and Thread



S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread

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

3.3 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.

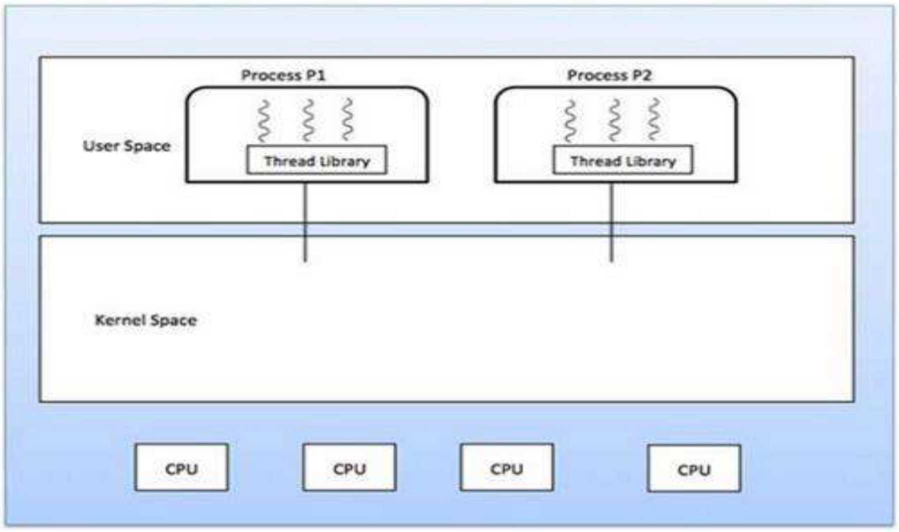


Figure 3.2 User Level Threads

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.

3.4 Concept of multithreads

3.4.1 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.

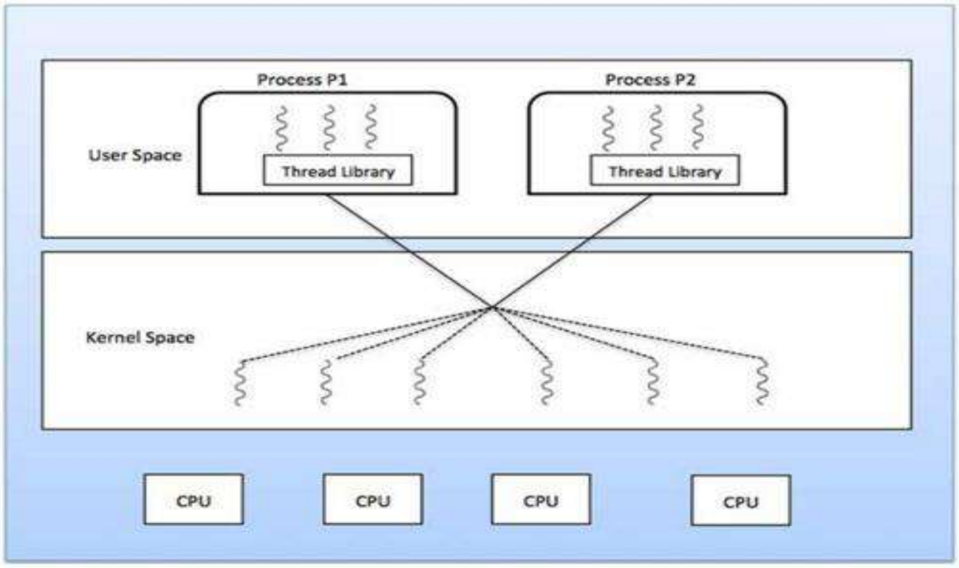


Figure 3.3 Many to Many Model

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.

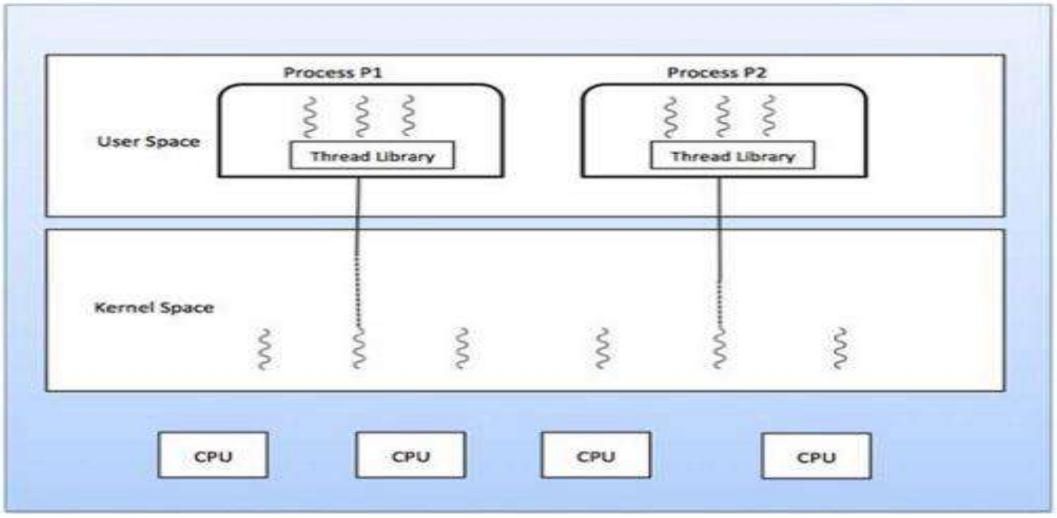


Figure 3.4 Many to One Model

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.

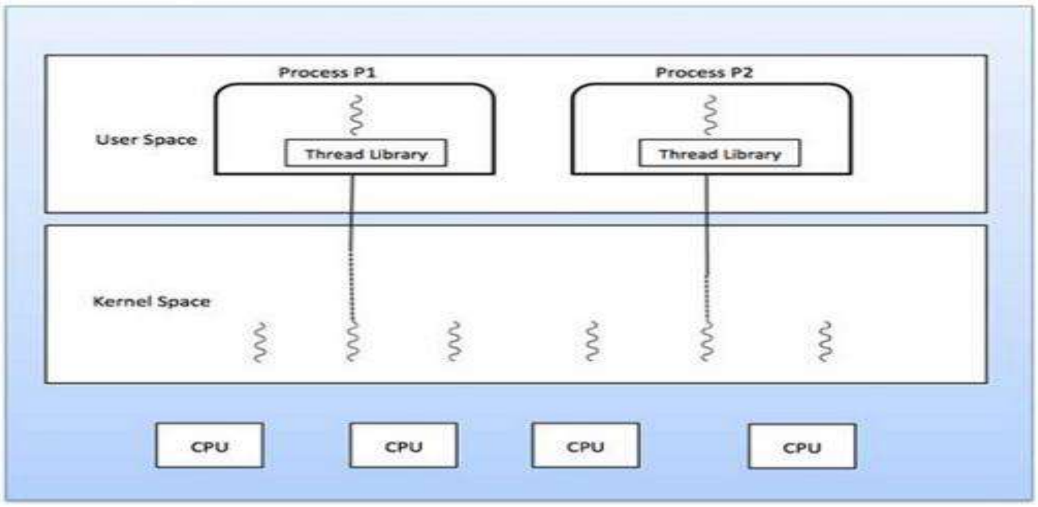


Figure 3.5 One to One Model

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.

3.4 .2 What are Thread Libraries?

Thread libraries provide programmers with API for creation and management of threads.
Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within the user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

Three types of Thread

1. **POSIX Ptheads**, may be provided as either a user or kernel library, as an extension to the POSIX standard.
2. **Win32 threads**, are 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 Ptheads or Win32 threads depending on the system.

3.5 Benefits of Multithreading

1. Responsiveness
2. Resource sharing, hence allowing better utilization of resources.
3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
5. Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

3.6 Multithreading Issues

Below we have mentioned a few issues related to multithreading.

Thread Cancellation

Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

fork() System Call

fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?

Security Issues

Yes, there can be security issues because of extensive sharing of resources between multiple threads.

There are many other issues that you might face in a multithreaded process, but there are appropriate solutions available for them. Pointing out some issues here was just to study both sides of the coin.

4. Process Scheduling

4.1 What is 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.

CPU Scheduling: Dispatcher

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

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program from where it left last time.

The dispatcher should be as fast as possible, given that it is invoked during every process switch. The time taken by the dispatcher to stop one process and start another process is known as the **Dispatch Latency**. Dispatch Latency can be explained using the below figure.

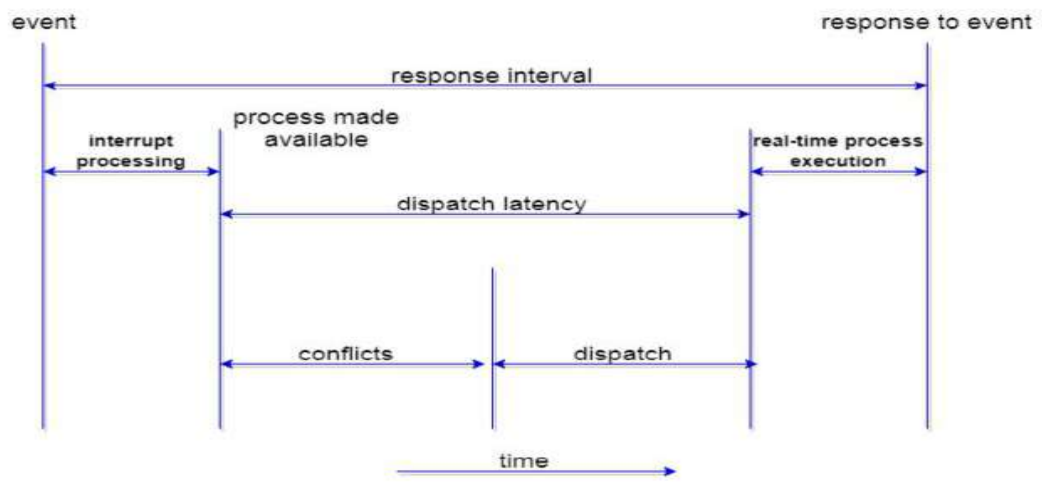


Figure 3.6 Dispatch Latency

Scheduling

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

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

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

When Scheduling takes place only under circumstances 1 and 4, we say the scheduling scheme is **non-preemptive**; otherwise the scheduling scheme is **preemptive**.

Non-Preemptive Scheduling

Under non-preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

This scheduling method is used by the Microsoft Windows 3.1 and by the Apple Macintosh operating systems.

It is the only method that can be used on certain hardware platforms, because It does not require the special hardware(for example: a timer) needed for preemptive scheduling.

Preemptive Scheduling

In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

4.2 CPU Scheduling: Scheduling Criteria

There are many different criterias to check when considering the **"best"** scheduling algorithm, they are:

CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

Turnaround Time=Completion Time - Arrival Time

Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

Waiting Time=Turnaround Time - Burst Time

Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Response Time=First instance of CPU(for any process) – Arrival Time(of that process)

4.3 Objectives of Process Scheduling Algorithm

- Max CPU utilization [Keep CPU as busy as possible]
- Fair allocation of CPU.
- Max throughput [Number of processes that complete their execution per time unit]
- Min turnaround time [Time taken by a process to finish execution]
- Min waiting time [Time a process waits in ready queue]
- Min response time [Time when a process produces first response]

4.4 Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilisation, computer scientists have defined some algorithms, they are:

1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

We will be discussing all the scheduling algorithms, one by one, in detail in the next tutorials.

First Come First Serve Scheduling

In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

- First Come First Serve, is just like FIFO(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first.
- This is used in Batch Systems.
- It's easy to understand and implement programmatically, using a Queue data structure, where a new process enters through the tail of the queue, and the scheduler selects process from the headof the queue.
- A perfect real life example of FCFS scheduling is buying tickets at ticket counter.

Calculating Average Waiting Time

For every scheduling algorithm, Average waiting time is a crucial parameter to judge it's performance.

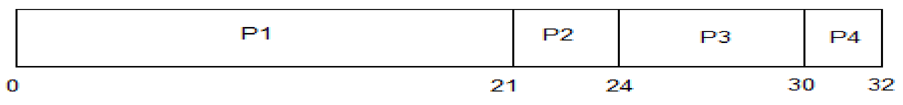
AWT or Average waiting time is the average of the waiting times of the processes in the queue, waiting for the scheduler to pick them for execution.

Lower the Average Waiting Time, better the scheduling algorithm.

Consider the processes P1, P2, P3, P4 given in the below table, arrives for execution in the same order, with Arrival Time 0, and given Burst Time, let's find the average waiting time using the FCFS scheduling algorithm.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

The average waiting time will be 18.75 ms
For the above given proccesses, first **P1** will be provided with the CPU resources,

- Hence, waiting time for **P1** will be 0
- **P1** requires 21 ms for completion, hence waiting time for **P2** will be 21 ms
- Similarly, waiting time for process **P3** will be execution time of **P1** + execution time for **P2**, which will be (21 + 3) ms = 24 ms.
- For process **P4** it will be the sum of execution times of **P1**, **P2** and **P3**.

The **GANTT chart** above perfectly represents the waiting time for each process.

Problems with FCFS Scheduling

Below we have a few shortcomings or problems with the FCFS scheduling algorithm:

1. It is **Non Pre-emptive** algorithm, which means the **process priority** doesn't matter.

If a process with very **least priority** is being executed, more like **daily routine backup** process,which takes more time, and all of a sudden some other high priority process arrives, like **interrupt to avoid system crash**, the high priority process will have to wait, and hence in this case, the system will crash, just because of improper process scheduling.

2. Not optimal Average Waiting Time.
3. Resources utilization in parallel is not possible, which leads to **Convoy Effect**, and hence poor resource(CPU, I/O etc) utilization.

What is Convoy Effect?

Convoy Effect is a situation where many processes, who need to use a **resource for short time are blocked** by one process holding that resource for a long time.

This essentially leads to poor utilization of resources and hence poor performance.

Here we have simple formulae for calculating various times for given processes:

Completion Time: Time taken for the execution to complete, starting from arrival time.

Turn Around Time: Time taken to complete after arrival. In simple words, it is the difference between the Completion time and the Arrival time.

Waiting Time: Total time the process has to wait before it's execution begins. It is the difference between the Turn Around time and the Burst time of the process.

Shortest Job First(SJF) Scheduling

Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first.

- This is the best approach to **minimize waiting time.**
- This is used in Batch Systems.
- It is of two types:
  1. Non Pre-emptive
  2. Pre-emptive
- To successfully implement it, the **burst time/duration time of the processes should be known** to the processor **in advance**, which is practically not feasible all the time.
- This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all)

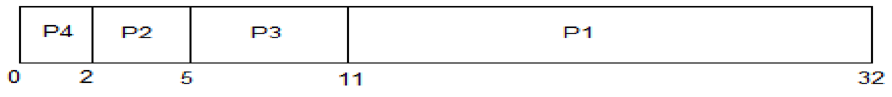
Non Pre-emptive Shortest Job First

Consider the below processes available in the ready queue for execution, with arrival time as 0 for all and given burst times.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

As you can see in the **GANTT chart** above, the process **P4** will be picked up first as it has the shortest burst time, then **P2**, followed by **P3** and at last **P1**.

We scheduled the same set of processes using the First come first serve algorithm in the previous tutorial, and got average waiting time to be 18.75 ms, whereas with SJF, the average waiting time comes out 4.5 ms.

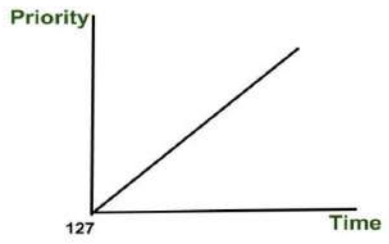
Problem with Non Pre-emptive SJF

If the **arrival time** for processes are different, which means all the processes are not available in the ready queue at time 0, and some jobs arrive after some time, in such situation, sometimes process with short burst time **have to wait for the current process's execution to finish,** because in Non Pre-emptive SJF, on arrival of a process with short duration, the existing job/process's execution is not halted/stopped to execute the short job first.

This leads to the problem of **Starvation**, where a shorter process has to wait for a long time until the current longer process gets executed. This happens if shorter jobs keep coming, but this can be solved using the concept of **aging**.

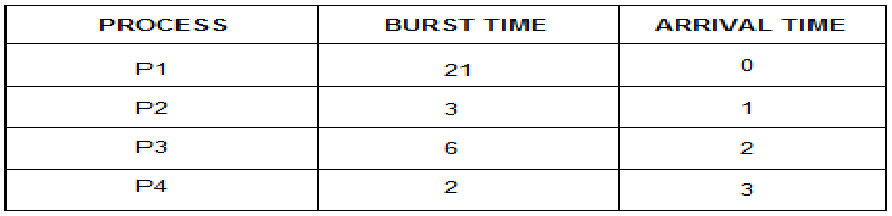
Solution to Starvation : Aging

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.For example, if priority range from 127(low) to 0(high), we could increase the priority of a waiting process by 1 Every 15 minutes. Eventually even a process with an initial priority of 127 would take no more than 32 hours for priority 127 process to age to a priority-0 process.



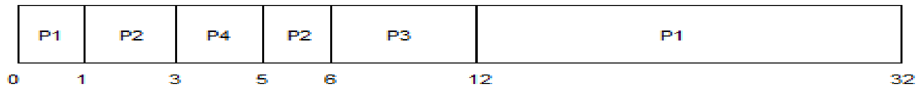
Pre-emptive Shortest Job First

In Preemptive Shortest Job First Scheduling, jobs are put into ready queue as they arrive, but as a process with **short burst time** arrives, the existing process is preempted or removed from execution, and the shorter job is executed first.



PROCESS	BURST TIME	ARRIVAL TIME
P1	21	0
P2	3	1
P3	6	2
P4	2	3

The GANTT chart for Preemptive Shortest Job First Scheduling will be,



The average waiting time will be, ((5-3) + (6-2) + (12-1))/4 = 4.25 ms

The average waiting time for preemptive shortest job first scheduling is less than both, non-preemptive SJF scheduling and FCFS scheduling.

As you can see in the **GANTT chart** above, as **P1** arrives first, hence it's execution starts immediately, but just after 1 ms, process **P2** arrives with a **burst time** of 3 ms which is less than the burst time of**P1**, hence the process **P1**(1 ms done, 20 ms left) is preemptied and process **P2** is executed.

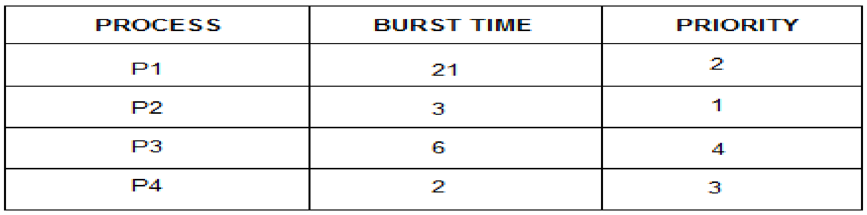
As **P2** is getting executed, after 1 ms, **P3** arrives, but it has a burst time greater than that of **P2**, hence execution of **P2** continues. But after another millisecond, **P4** arrives with a burst time of 2 ms, as a result **P2**(2 ms done, 1 ms left) is preemptied and **P4** is executed.

After the completion of **P4**, process **P2** is picked up and finishes, then **P2** will get executed and at last **P1**.

The Pre-emptive SJF is also known as **Shortest Remaining Time First**, because at any given point of time, the job with the shortest remaining time is executed first.

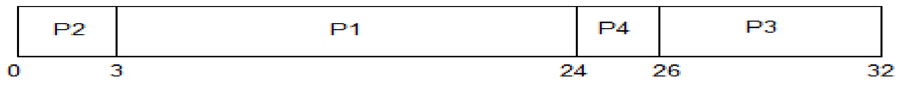
Priority Scheduling

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.



PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

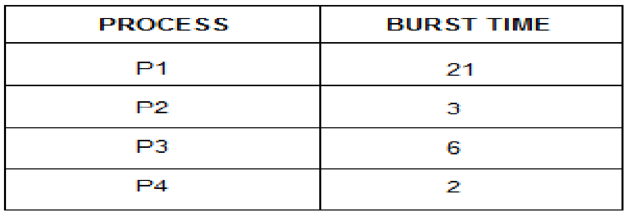
The GANTT chart for following processes based on Priority scheduling will be,



The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

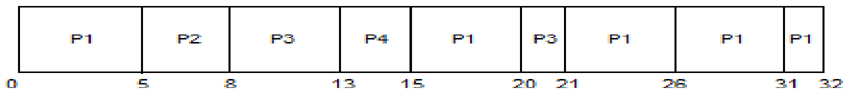
Round Robin Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preempted processes.



PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which **processes are easily classified into different groups.**

For example: A common division is made **between foreground(or interactive) processes** and **background (or batch) processes**. These two types of processes have different response-time requirements, and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A multi-level queue scheduling algorithm **partitions** the ready queue into **several separate queues.** The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.

For example: separate queues might be used for foreground and background processes. The foreground queue might be scheduled by Round Robin algorithm, while the background queue is scheduled by an FCFS algorithm.

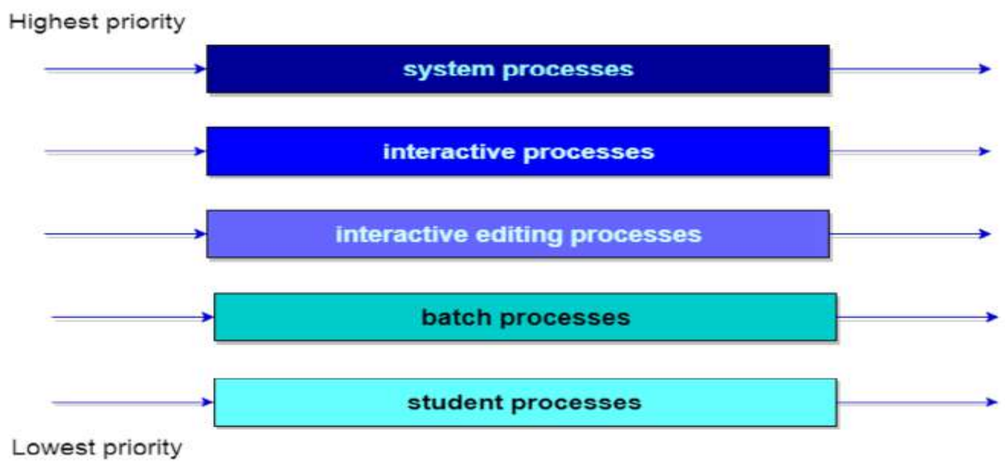
In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. **For example:** The foreground queue may have absolute priority over the background queue.

Let us consider an example of a multilevel queue-scheduling algorithm with five queues:

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes

4. Batch Processes
5. Student Processes

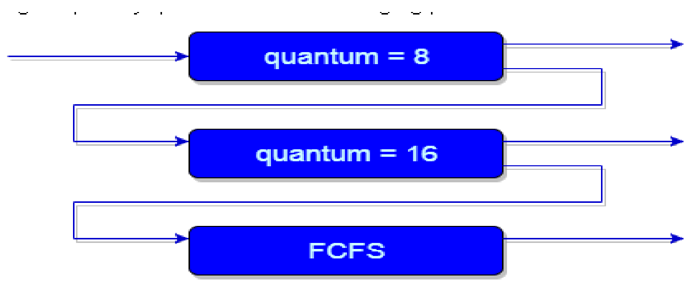
Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process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



Multilevel Feedback Queue Scheduling

In a multilevel queue-scheduling algorithm**, processes are permanently assigned to a queue** on entry to the system. Processes do not move between queues. This setup has the advantage of low scheduling overhead, but the disadvantage of being inflexible.

Multilevel feedback queue scheduling, however, **allows a process to move between queues**. The idea is to separate processes with **different CPU-burst** characteristics. If a process uses too much CPU time, it will be moved to a lower-priority queue. Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.



An example of a multilevel feedback queue can be seen in the below figure.

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

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher-priority queue.
- The method used to determine when to demote a process to a lower-priority queue.
- The method used to determine which queue a process will enter when that process needs service.

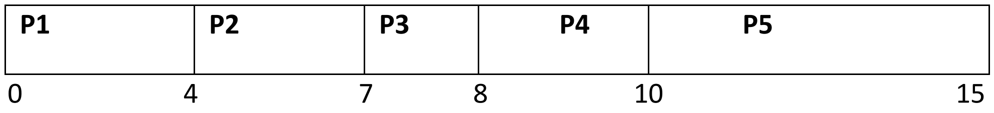
The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it also requires some means of selecting values for all the parameters to define the best scheduler. Although a multilevel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

EXERCISE PROBLEMS SCHEDULING ALGORITHMS

FIRST COME FIRST SERVE(FCFS)

PROCESS	AT	BT	CT	TAT	WT	RST
P1	0	4	*4*	*4*	*0*	*0*
P2	1	3	*7*	*6*	*3*	*3*
P3	2	1	*8*	*6*	*5*	*5*
P4	3	2	*10*	*7*	*5*	*5*
P5	4	5	*15*	*11*	*6*	*6*

GANTT CHART:



Avg. TAT=(4+6+6+7+11)/5=6.8

Avg. WT =(0+3+2+5+6)/5 =3.2

EXPLANATION:

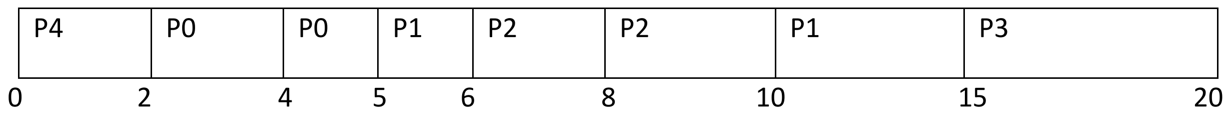
- The Process P1 arrives first and is executes for 4ms.
- When the Process P1 is in execution other process such as P2,P3,P4,P5 arrives.
- As the Scheduling algorithm named as FCFS, means the process which comes first will be executes first irrespective of other processes.
- So, P1 arrives first and it is executed first then P2 ,P3,P4,P5 executes sequentially as they are in the queue(depending on arrival time).

SHORTEST JOB FIRST-

I-PREEMPTIVE:

PROCESS	AT	BT	CT	TAT	WT	RST
P0	2	3	*5*	*3*	*0*	*0*
P1	4	6	*15*	*11*	*5*	*1*
P2	6	4	*10*	*4*	*0*	*0*
P3	8	5	*20*	*12*	*7*	*7*
P4	0	2	*2*	*2*	*0*	*0*

GANTT CHART:



Avg. TAT=(3+11+4+12+2)/5= 6.4

Avg. WT=(0+5+0+7+0)/5=2.4

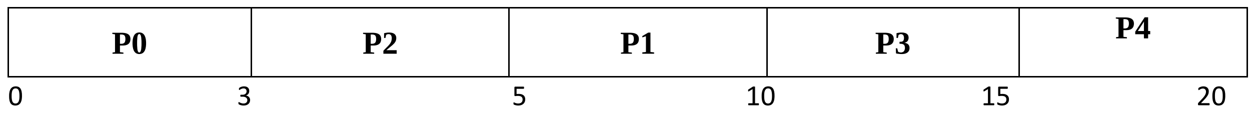
EXPLANATION:

- At 0ms, P4 arrives and runs for 2ms.
- At 2ms ,P0 arrives and runs for 2ms.
- At 4ms , P0 has 1ms left and P1 arrives .since P0 is short compare to P1 , hence P0 executes .
- At 5ms, P1 executes for 1ms.
- At 6ms ,P1 has left with 5ms and P2 arrives at same time . since P2 is shorter compare to P1 as it is having 4ms(BT) so, P2 executes for 2ms.
- At 8ms , P2 has 2ms and P3 arrives . P2 is shorter than P1 and P0, hence P2 executes.
- At 10ms,Both P1 and P3 have same time left so P1 will be executed due to prior arrival then P3 executes.

II-NON-PREEMPTIVE:

PROCESS	AT	BT	CT	TAT	WT	RST
P0	**0**	**3**	***3***	***3***	***0***	***0***
P1	**1**	**5**	***10***	***9***	***4***	***4***
P2	**3**	**2**	***5***	***2***	***0***	***0***
P3	**9**	**5**	***15***	***6***	***1***	***1***
P4	**12**	**5**	***20***	***8***	***3***	***3***

GANTT CHART:



Avg. TAT =(3+9+2+6+8)/5=5.6

Avg. WT= (0+4+0+1+3)/5 = 1.6

EXPLANATION:

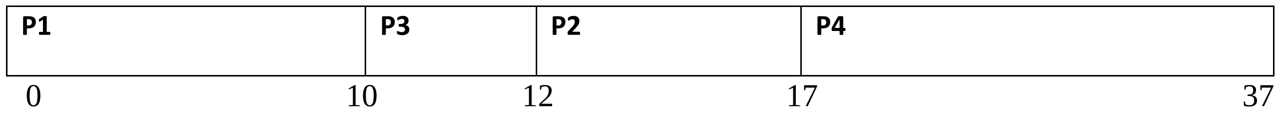
- In case of Non- Preemptive (SJF), we consider the shortest Burst time.
- Initially at 0 ms ,P0 arrives and executes for 3ms .
- During P0 executes P1 and P2 arrives , after P1 execution then P2 has shortest burst time so P2 will be executed.
- The Process P1 is executed next, because all the left over processes has the same burst time i.e, P3,P4 so P1 is executed then P3,P4 executes .

PRIORITY SCHEDULING-

I-NON-PREEMPTIVE:

PROCESS	PRIORITY	AT	BT	CT	TAT	WT	RST
P1	**2**	**0**	**10**	***10***	***10***	***0***	***0***
P2	**1**	**2**	**5**	***17***	***15***	***10***	***10***
P3	**0 (H)**	**3**	**2**	***12***	***9***	***7***	***7***
P4	**3 (L)**	**5**	**20**	***37***	***32***	***12***	***12***

GANTT CHART:



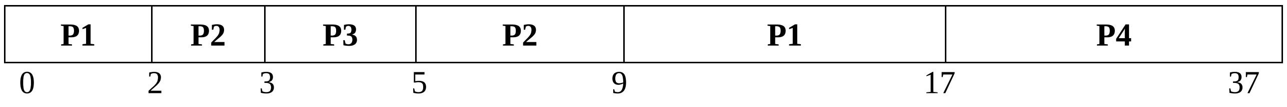
EXPLAINATION:

- Process P1 arrives first and is executed for10 ms
- When P1 is executing, processes P2, P3, P4 arrive
- Among these processes, P3 has the highest priority and hence it is executed. P3 executes for 2 ms
- P2, P4 remain for execution. P2 is executed since it has more priority. P2 executes for 5 ms
- Finally, P4 is executed. It executes for 20 ms

II- PREEMPTIVE:

PROCESS	PRIORITY	AT	BT	CT	TAT	WT	RST
P1	2	0	10	*17*	*17*	*7*	*0*
P2	1	2	5	*9*	*7*	*2*	*0*
P3	0 (H)	3	2	*5*	*2*	*0*	*0*
P4	3 (L)	5	20	*37*	*32*	*12*	*12*

GANTT CHART:



Avg. TAT= (17+7+2+32)/4 = 58/4 = 14.5
Avg. WT= (7+2+0+12)/4 = 21/4 = 5.2

EXPLAINATION:

- Process P1 arrives first and is executed for 2 ms. At 2ms, process P2 arrives. P2 has higher priority than P1. Hence P1 is prempted and P2 starts executing.
- P2 executes for 1 ms. At 3 ms, P3 arrives. P3 has highest priority and hence P2 is prempted and P3 starts its execution.
- P3 executes for 2 ms and gets completed at 5 ms. P4 arrives at 5 ms. Among P1, P2, P4, priority of P2 is highest and hence P2 starts its execution.
- P2 executes for 4 ms (remaining) till 9 ms and gets completed. Now among P1 and P4, P1 starts executing due to higher priority.
- P1 executes for 8 ms (remaining) till 17 ms and gets completed. Now the last process P4 starts executing and P4 executes for 20 ms till 37ms

ROUND ROBIN SCHEDULING-

QUANTUM= 2 MS

PROCESS	AT	BT	CT	TAT	WT	RST
P1	**0**	**4(2)(0)**	***8***	***8***	***4***	***0***
P2	**1**	**5(3)(1)(0)**	***18***	***17***	***12***	***1***
P3	**2**	**2(0)**	***6***	***4***	***2***	***2***
P4	**3**	**1(0)**	***9***	***6***	***5***	***5***
P5	**4**	**6(4)(2)**	***21***	***17***	***11***	***5***
P6	**6**	**3(1)**	***19***	***13***	***10***	***7***

GANTT CHART:

P1	**P2**	**P3**	**P1**	**P4**	**P5**	**P2**	**P6**	**P5**	**P2**	**P6**	**P5**

0 2 4 6 8 9 11 13 15 17 18 19 21

Avg. TAT= (8+17+4+6+17+13)/6 = 65/6 = 10.8
Avg. WT= (4+12+2+5+11+10)/6 = 44/6 = 7.3

EXPLAINATION:

- Process P1 arrives first and is executed for 2 ms. So lime left for completion of P1 is 4-2= 2 ms. In the duration 0 ms to 2 ms, P2 and P3 processes arrive. The ready queue is as follows:

P2	P3	P1

- Now P2 is sent for execution. It executes for 2 ms and time left for completion of P2 is 5-2= 3 ms. In duration 2 ms to 4 ms, P4 and P5 processes arrive. The ready queue is as follows:

P3	P1	P4	P5	P2

- Now P3 is sent for execution. It executes for 2 ms and time left for completion of P2 is 2-2= 0 ms. P3 is completed and is removed from ready queue. In duration 4 ms to 6 ms, P6 process arrives. The ready queue is as follows:

P1	P4	P5	P2	P6

- Now P1 is sent for execution. It executes for 2 ms and time left for completion of P1 is 2-2= 0 ms. P1 is completed and is removed from ready queue. The ready queue is as follows:

P4	P5	P2	P6

- Now P4 is sent for execution. It executes for 1 ms (and gets completed). P4 is removed from ready queue. The ready queue is as follows:

P5	P2	P6

- Now P5 is sent for execution. It executes for 2 ms and time left for completion of P5 is 6-2= 4 ms. The ready queue is as follows:

P2	P6	P5

- Now P2 is sent for execution. It executes for 2 ms and time left for completion of P2 is 3-2= 1 ms. The ready queue is as follows:

P6	P5	P2

- Now P6 is sent for execution. It executes for 2 ms and time left for completion of P6 is 3-2= 1 ms. The ready queue is as follows:

P5	P2	P6

- Now P5 is sent for execution. It executes for 2 ms and time left for completion of P5 is 4-2= 2 ms. The ready queue is as follows:

P2	P6	P5

- Now P2 is sent for execution. It executes for 1 ms (and gets completed). P2 is removed from ready queue. The ready queue is as follows:

P6	P5

- Now P6 is sent for execution. It executes for 1 ms (and gets completed). P6 is removed from ready queue. The ready queue is as follows:

P5

- Now P5 is sent for execution. It executes for 2 ms and time left for completion of P5 is 2-2= 0 ms. The ready queue is now empty indicating the end of execution.

EMPTY

3.Process Synchronization

Process synchronization is the task of synchronizing the execution of processes in such a manner that no two processes have access to the same shared data and resource. In a multi process system when multiple processes are running simultaneously, then they may attempt to gain the access of same shared data and resource at a time. This can lead in inconsistency of shared data. That is the changes made by one process may not be reflected when other process accessed the same shared data. In order to avoid these types of inconsistency of data the processes should be synchronized with each other.

One of the important concepts related to process synchronization is that of the **critical selection problem**. Each process contains a set of code called critical section through which a specific task, such as writing some data to file or changing the value of global variable, is performed. To ensure that only one single process should be entering in critical section at a specific instant of time, the process needs to be coordinated with other by sending requests for entering the critical section. When a process is in its critical section, then no other process is allowed to enter in critical section during the time period when one process is in a critical section.

The following are the different solutions or algorithms used in synchronizing the different processes of any operating system:

1. Peterson's solution

Peterson's solution is one of the famous solutions to critical section problems. This algorithm is created by a computer scientist Peterson. Peterson's solution is solution to the critical section problem involving two processes. Peterson's solution states that when a process is executing in its critical state, then the other process executes the rest of code and vice versa. This insures that only one process is in the critical section at a particular instant of time.

2. Locking solution

Locking solution is another solution to critical problems in which a process acquires a lock before entering its critical section. When a process finishes its executing process in the critical section, then it releases the lock. Then the lock is available for any other process that wants to execute its critical section. The locking mechanism also ensures that only one process is in the critical section at a particular instant of time.

3. Semaphore solution

Semaphore solutions are another algorithm or solution to the critical section problem. It is basically a synchronization tool in which the value of an integer variable called semaphore is retrieved and set using wait and signal operations. Based on the value of the semaphore variable, a process is allowed to enter its critical section.

3.1 Inter-process Communication

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. There are several reasons for providing an environment that allows process cooperation:

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

• **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

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

• **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel. Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of inter-process communication:

(1) shared memory and **(2) message passing.** In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for intercomputer communication. Shared memory allows maximum speed and convenience of communication**,** as it can be done at memory speeds when within a computer.

Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time consuming task of kernel intervention.

Let's discuss an example of communication between processes using shared memory method.

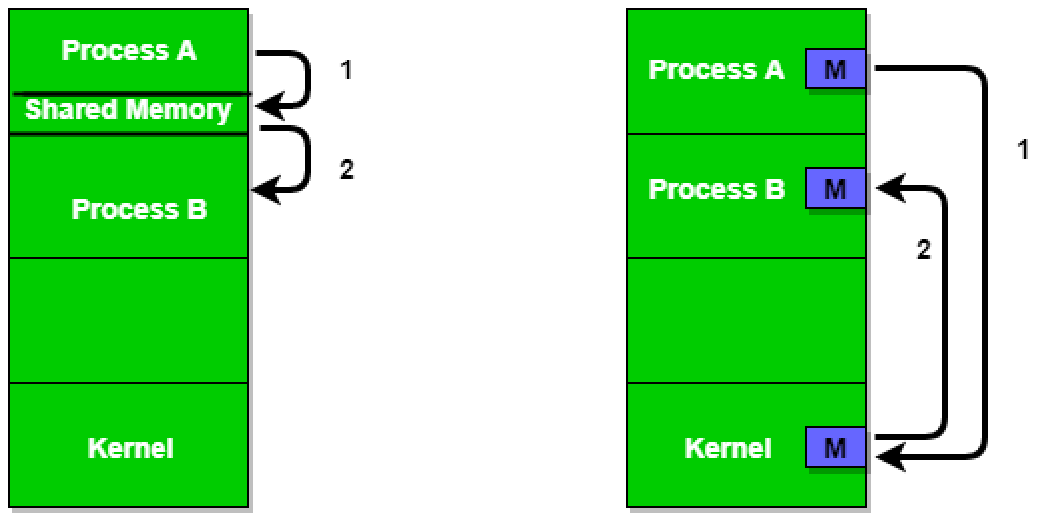


Figure Shared Memory and Message passing

Shared-Memory Systems

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

Message-Passing Systems

The scheme requires that these processes share a region of memory and that the code for accessing and manipulating the shared memory be written explicitly by the application programmer. Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility. actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size**.** If only fixed-sized messages can be sent, the system-level implementation is straightforward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler. This is a common kind of tradeoff seen throughout operating system design.

Naming

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

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

• send(P, message)—Send a message to process P.

• receive (Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

• A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

• A link is associated with exactly two processes.

• Between each pair of processes, there exists exactly one link.

The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.Message passing provides a mechanism to allow processes to communicate and to synchroniz.

Synchronization

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

• **Blocking send-** The sending process is blocked until the message is received by the receiving process or by the mailbox.

• **Nonblocking send-** The sending process sends the message and resumes operation.

• **Blocking receive-** The receiver blocks until a message is available.

• **Nonblocking receive-** The receiver retrieves either a valid message or a null.

Buffering

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

• **Zero capacity-** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

• **Bounded capacity-** The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

• **Unbounded capacity-** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

3.2 Critical Section Problem

A Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

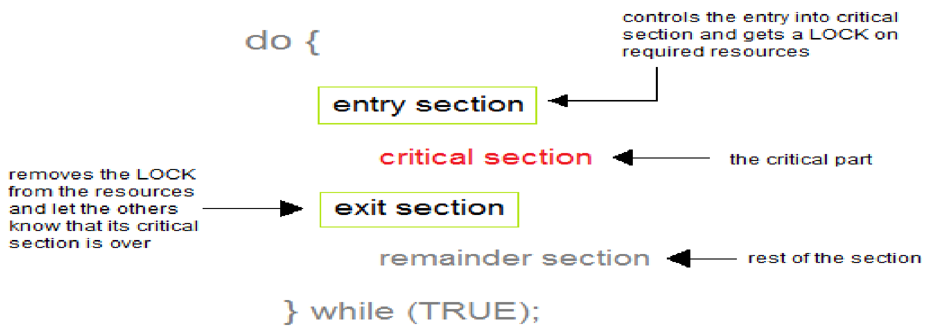

do {
controls the entry into critical section and gets a LOCK on required resources
entry section
critical section
the critical part
removes the LOCK from the resources and let the others know that its critical section is over
exit section
remainder section
rest of the section
} while (TRUE);

Figure general structure of a typical process pi

Solution to Critical Section Problem

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

1. Mutual Exclusion

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. Bounded Waiting

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

3.3 Race Condition

When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute.

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

Example:

In an Operating System, we have a number of processes and these processes require a number of resources. Now, think of a situation where we have two processes and these processes are using the same variable "a". They are reading the variable and then updating the value of the variable and finally writing the data in the memory.

```
SomeProcess()
{
   ...
   read(a) //instruction 1
   a = a + 5 //instruction 2
   write(a) //instruction 3
   ...
}
```

In the above, you can see that a process after doing some operations will have to read the value of "a", then increment the value of "a" by 5 and at last write the value of "a" in the memory. Now, we have two processes P1 and P2 that needs to be executed. Let's take the following two cases and also assume that the value of "a" is 10 initially.

1. In this case, process P1 will be executed fully (i.e. all the three instructions) and after that, the process P2 will be executed. So, the process P1 will first read the value of "a" to be 10 and then increment the value by 5 and make it to 15. Lastly, this value will be updated in the memory. So, the current value of "a" is 15. Now, the process P2 will read the value i.e. 15, increment with 5(15+5 = 20) and finally write it to the memory i.e. the new value of "a" is 20. Here, in this case, the final value of "a" is 20.
2. In this case, let's assume that the process P1 starts executing. So, it reads the value of "a" from the memory and that value is 10(initial value of "a" is taken to be 10). Now, at this time, context switching happens between process P1 and P2. Now, P2 will be in the running state and P1 will be in the waiting state and the context of the P1 process will be saved. As the process P1 didn't change the value of "a", so, P2 will also read the value of "a" to be 10. It will then increment the value of "a" by 5 and make it to 15 and then save it to the memory. After the execution of the process P2, the process P1 will be resumed and the context of the P1 will be read. So, the process P1 is having the value of "a" as 10(because P1 has already executed the instruction 1). It will then increment the value of "a" by 5 and write the final value of "a" in the memory i.e. a = 15. Here, the final value of "a" is 15.

In the above two cases, after the execution of the two processes P1 and P2, the final value of "a" is different i.e. in 1st case it is 20 and in 2nd case, it is 15. What's the reason behind this?

The processes are using the same resource here i.e. the variable "a". In the first approach, the process P1 executes first and then the process P2 starts executing. But in the second case, the process P1 was

stopped after executing one instruction and after that the process P2 starts executing. And here both the processes are dealing on the same resource i.e. variable "a" at the same time.
Here, the order of execution of processes changes the output. All these processes are in a race to say that their output is correct. This is called a race condition.

3.4 Peterson's Solution

- Peterson's Solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.
- Peterson's solution is based on two processes, P0 and P1, which alternate between their critical sections and remainder sections. For convenience of discussion, "this" process is Pi, and the "other" process is Pj. (I.e. j = 1 - i)
- Peterson's solution requires two shared data items:
    - **int turn** - Indicates whose turn it is to enter into the critical section. If turn = = i, then process i is allowed into their critical section.
    - **boolean flag[2]** - Indicates when a process *wants to* enter into their critical section. When process i wants to enter their critical section, it sets flag[i] to true.
- In the following diagram, the entry and exit sections are enclosed in boxes.
    - In the entry section, process i first raises a flag indicating a desire to enter the critical section.
    - Then turn is set to ***j*** to allow the other process to enter their critical section if process j so desires.
    - The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the turn and wants to enter the critical section.
    - Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

```
do {

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure - The structure of process Pi in Peterson's solution.

Peterson’s Solution preserves all three conditions :

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

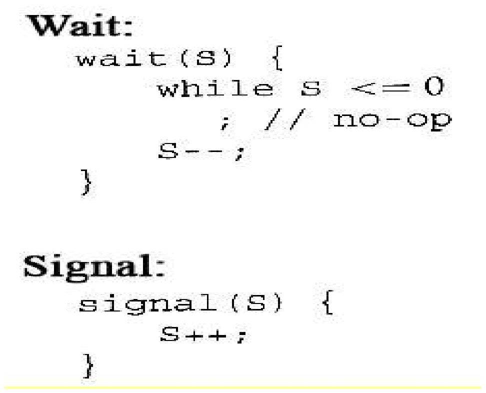
Disadvantages of Peterson’s Solution

- It involves Busy waiting
- It is limited to 2 processes

3.5 Introduction to Semaphores

In 1965, Dijkstra proposed a new and very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called semaphore. So it is basically a synchronizing tool and is accessed only through two low standard atomic operations, wait and signal designated by P(S) and V(S) respectively.

In very simple words, semaphore is a variable which can hold only a non-negative Integer value, shared between all the threads, with operations wait and **s**ignal, which work as follow:



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

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

Figure Semaphore

- **Wait**: Decrements the value of its argument S, as soon as it would become non-negative(greater than or equal to 1).
- **Signal**: Increments the value of its argument S, as there is no more process blocked on the queue.

3.5.1 Semaphore Usage

There are two types of semaphores : Binary Semaphores and Counting Semaphores

- **Binary Semaphores** : They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

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

Figure Mutual-exclusion implementation with semaphores.

- **Counting Semaphores :** They can have any value and are not restricted over a certain domain. They can be used to control access a resource that has a limitation on the number of simultaneous accesses. The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available. Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

- 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.

3.5.2 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);
        }
}
```

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

3.5.3 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 gets 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.

3.5.4 Priority Inversion

- A challenging scheduling problem arises when a high-priority process gets blocked waiting for a resource that is currently held by a low-priority process.
- If the low-priority process gets pre-empted by one or more medium-priority processes, then the high-priority process is essentially made to wait for the medium priority processes to finish before the low-priority process can release the needed resource, causing a priority inversion. If there are enough medium-priority processes, then the high-priority process may be forced to wait for a very long time.
- One solution is a priority-inheritance protocol, in which a low-priority process holding a resource for which a high-priority process is waiting will temporarily inherit the high priority from the waiting process. This prevents the medium-priority processes from preempting the low-priority process until it releases the resource, blocking the priority inversion problem.

3.6 Classical Problems of Synchronization

Semaphore can be used in other synchronization problems besides Mutual Exclusion.

Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.

We will discuss the following three problems:

1. Bounded Buffer (Producer-Consumer) Problem
2. The Readers Writers Problem
3. Dining Philosophers Problem

3.6.1 The Bounded-Buffer Problem

- Bounded buffer problem, which is also called producer consumer problem, is one of the classic problems of synchronization.
- There is a buffer of n slots and each slot is capable of storing one unit of data. There are two processes running, namely, producer and consumer, which are operating on the buffer.

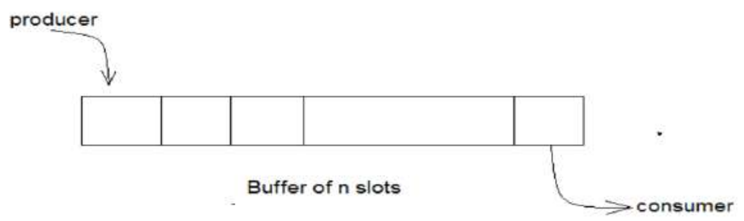


Figure: Bounded Buffer Problem

- A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. these two processes won't produce the expected output if they are being executed concurrently. There needs to be a way to make the producer and consumer work in an independent manner.

- This problem is generalized in terms of the Producer Consumer problem, where a finite buffer pool is used to exchange messages between producer and consumer processes.
  Because the buffer pool has a maximum size, this problem is often called the Bounded buffer problem.

One solution of this problem is to use semaphores. The semaphores which will be used here are:

- m, a **binary semaphore** which is used to acquire and release the lock.
- empty, a **counting semaphore** whose initial value is the number of slots in the buffer, since, initially all slots are empty.
- full, a **counting semaphore** whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
// wait until empty > 0 and then decrement 'empty'
wait(empty);
// acquire lock
wait(mutex);

/* perform the insert operation in a slot */

// release lock
signal(mutex);
// increment 'full'
signal(full);
}
while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
// wait until full > 0 and then decrement 'full'
wait(full);
// acquire the lock
wait(mutex);
/* perform the remove operation in a slot */
// release the lock
signal(mutex);
// increment 'empty'
signal(empty);
}
while(TRUE);
```

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

3.6.2 The Readers-Writers Problem

- In the readers-writers problem there are some processes (termed readers) who only read the shared data, and never change it, and there are other processes (termed writers) who may change the data in addition to or instead of reading it. There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.
- There are several variations to the readers-writers problem, most centered around relative priorities of readers versus writers.
  - The *first* readers-writers problem gives priority to readers. In this problem, if a reader wants access to the data, and there is not already a writer accessing it, then access is granted to the reader. A solution to this problem can lead to starvation of the writers, as there could always be more readers coming along to access the data. (A steady stream of readers will jump ahead of waiting writers as long as there is currently already another reader accessing the data, because the writer is forced to wait until the data is idle, which may never happen if there are enough readers.)
  - The *second* readers-writers problem gives priority to the writers. In this problem, when a writer wants access to the data it jumps to the head of the queue - All waiting readers are blocked, and the writer gets access to the data as soon as it becomes available. In this solution the readers may be starved by a steady stream of writers.
- The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:

- readcount is used by the reader processes, to count the number of readers currently accessing the data.
- mutex is a semaphore used only by the readers for controlled access to readcount.
- rw_mutex is a semaphore used to block and release the writers. The first reader to access the data will set this lock and the last reader to exit will release it; The remaining readers do not touch rw_mutex.
- Note that the first reader to come along will block on rw_mutex if there is currently a writer accessing the data, and that all following readers will only block on mutex for their turn to increment readcount.

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

Figure 5.11 The structure of a writer process.

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

Figure The Structure of Reader process

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

3.6.3 The Dining-Philosophers Problem

The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources amongst a group of processes in a deadlock-free and starvation-free manner:

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

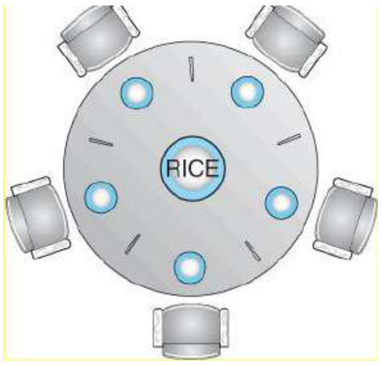
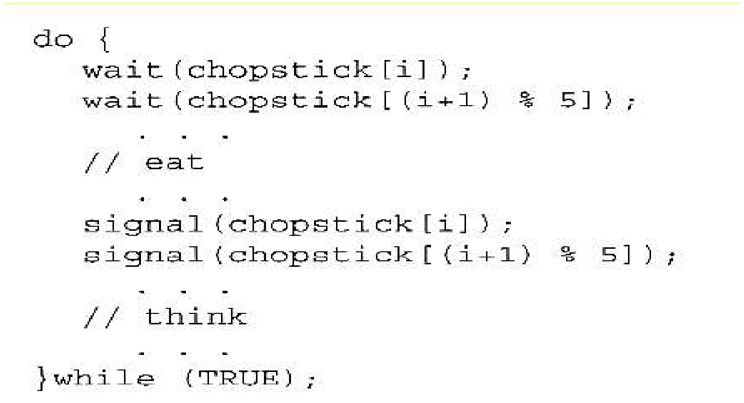


Figure - The situation of the dining philosophers

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



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

Figure - The structure of philosopher i.

- Some potential solutions to the problem include:
    - Only allow four philosophers to dine at the same time. (Limited simultaneous processes.)
    - Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
    - Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first. (Will this solution always work? What if there are an even number of philosophers?)
- Note carefully that a deadlock-free solution to the dining philosophers problem does not necessarily guarantee a starvation-free one. (While some or even most of the philosophers may be able to get on with their normal lives of eating and thinking, there may be one unlucky soul who never seems to be able to get both chopsticks at the same time.

3.7 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*.

3.7.1 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 ( . . . ) {
      . . .
   }
}
```

Figure - Syntax of a monitor.

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

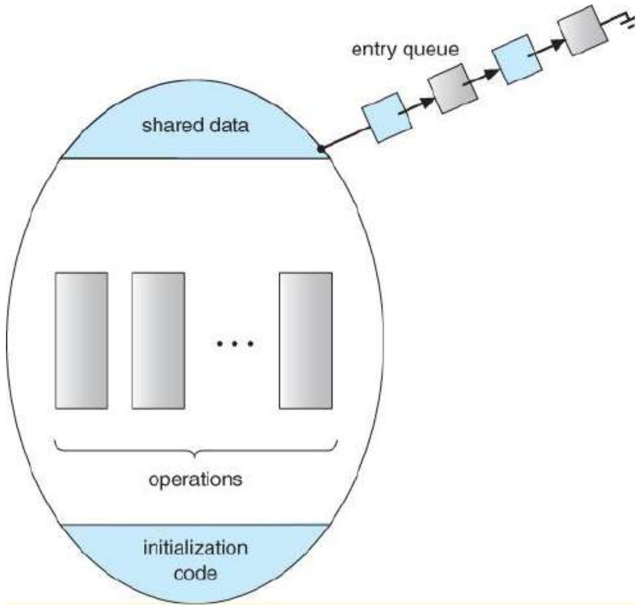


Figure - Schematic view of a monitor

- In order to fully realize the potential of monitors, we need to introduce one additional new data type, known as a ***condition***.
    - A variable of type condition has only two legal operations, ***wait*** and ***signal***. I.e. if X was defined as type condition, 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.
- Figure below illustrates a monitor that includes condition variables within its data space. Note that the condition variables, along with the list of processes currently waiting for the conditions, are in the data space of the monitor - The processes on these lists are not "in" the monitor, in the sense that they are not executing any code in the monitor.

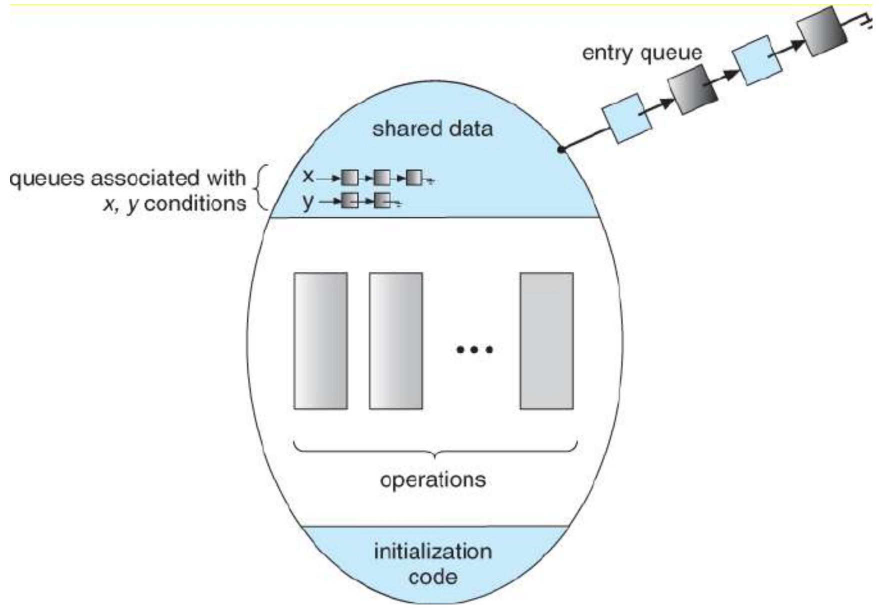


Figure - Monitor with condition variables

- But now there is a potential problem - If process P within the monitor issues a signal that would wake up process Q also within the monitor, then there would be two processes running simultaneously within the monitor, violating the exclusion requirement. Accordingly there are two possible solutions to this dilemma:

Signal and wait - When process P issues the signal to wake up process Q, P then waits, either for Q to leave the monitor or on some other condition.

Signal and continue - When P issues the signal, Q waits, either for P to exit the monitor or for some other condition.

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

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

4.Deadlock Introduction

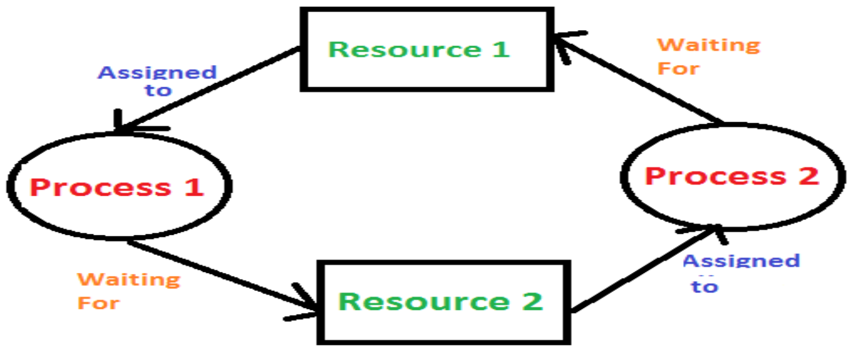
A process in operating systems uses different resources and uses resources in following way.

1) Requests a resource

2) Use the resource

2) Releases the resource

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.
when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



4.1 Necessary Conditions

There are four conditions that are necessary to achieve deadlock:

1. Mutual Exclusion - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.

2. Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

3. No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

4. Circular Wait - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.

4.2 Resource-Allocation Graph

In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:

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

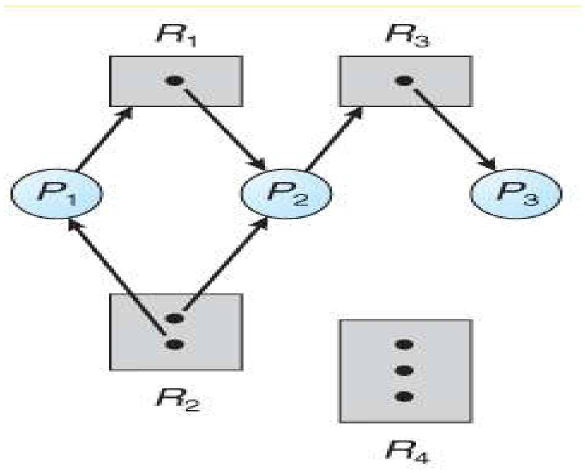


Figure Resource Allocation Graph

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are directed graphs.)
- If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.

- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example

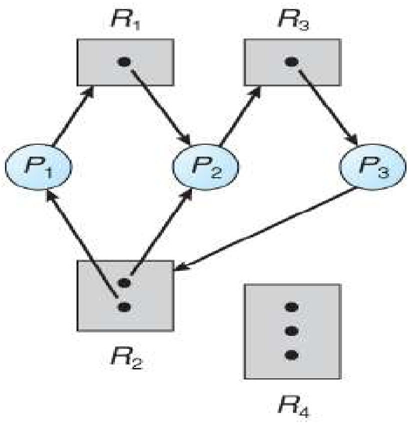


Figure Resource Allocation Graph with a deadlock

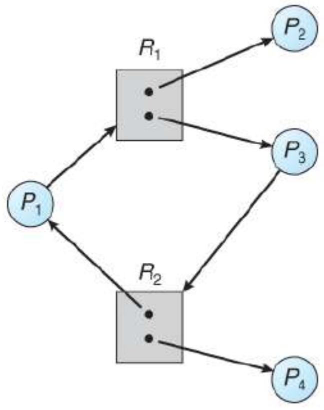


Figure Resource Allocation Graph with a cycle but no deadlock

4.3 Methods for Handling Deadlocks

- There are three ways of handling deadlocks:

  1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.

  2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.

  3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

- In order to avoid deadlocks, the system must have **additional information** about all processes. In particular, the system must know what resources a process will or may request in the future.
- Deadlock detection is fairly straightforward, but deadlock recovery requires either **aborting** processes or **preempting r**esources, neither of which is an attractive alternative.
- If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually **slow down**, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

4.4 Deadlock Prevention

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

4.4.1 Mutual Exclusion

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

4.4.2 Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
    - Require that **all processes request all resources at one time**. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
    - Require that **processes holding resources must release them before requesting new resources**, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
    - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

4.4.3 No Preemption

- Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
    - One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to **re-acquire the old resources along with the new resources** in a single request, similar to the previous discussion.
    - Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are **themselves blocked waiting** for some other resource. If such a process is found, then some of their **resources may get preempted** and added to the list of resources for which the process is waiting.
    - Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

4.4.4 Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources

4.5 Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)
- In some algorithms the **scheduler** only needs to know **the *maximum* number of each resource** that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of **exactly what resources may be needed in what order.**
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.

- A resource allocation *state* is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

4.5.1 Safe State

- A state is *safe* if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a *safe sequence* of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which *MAY* lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

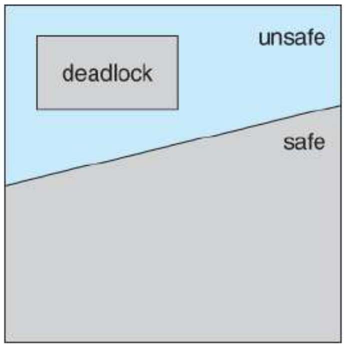


Figure - Safe, unsafe, and deadlocked state spaces.

- For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	**10**	**5**
P1	**4**	**2**
P2	**9**	**2**

- What happens to the above table if process P2 requests and is granted one more tape drive?
- Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

4.5.2 Resource-Allocation Graph Algorithm

- If resource categories have **only single instances of their resources**, then deadlock states can be detected by cycles in the resource-allocation graphs.
- In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.
- In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)
- When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.
- This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.
- Consider for example what happens when process P2 requests resource R2:

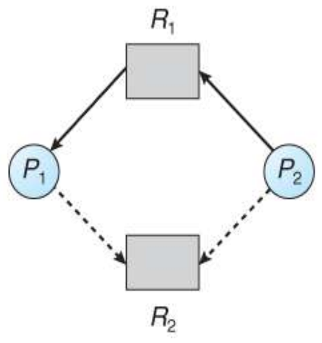


Figure - Resource allocation graph for deadlock avoidance

- The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted

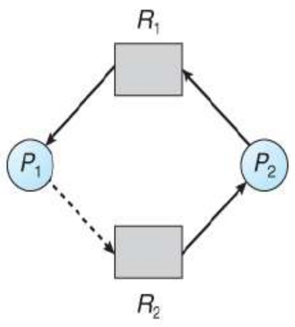


Figure - An unsafe state in a resource allocation graph

4.5.3 Banker's Algorithm

- For resource categories that **contain more than one instance the resource-allocation graph method does not work,** and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)
- When a process starts up, it must **state in advance the maximum allocation of resources** it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a **safe state**. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
    - Available[m] indicates how many resources are currently available of each type.
    - Max[n][m] indicates the maximum demand of each process of each resource.
    - Allocation[n][m] indicates the number of each resource category allocated to each process.
    - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
    - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
    - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

4.5.3.1 Safety Algorithm

- In order to apply the Banker's algorithm, we first need an **algorithm** for determining whether or not a particular **state is safe**.
- This algorithm determines if the current state of a system is safe, according to the following steps:

1. Let Work and Finish be vectors of length m and n respectively.
    - **Work** is a working copy of **the available resources**, which will be modified during the analysis.
    - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
    - Initialize Work to Available, and Finish to false for all elements.
2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

4.5.3.2 Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular **state is safe or not**, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
    1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
    2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
    3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:
        - Available = Available - Request
        - Allocation = Allocation + Request
        - Need = Max – Allocation
        - Max=Initial Allocation+Need

4.5.3.3 An Illustrative Example for Banker's Algorithm

Considering a system with five processes P_0 through P_4 and three resources types A, B, C. Resource type **A has 10 instances, B has 5 instances and type C has 7 instances.**

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Need Matrix=Max-Allocation

	Need		
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Step-1 Work=Available

Finish=False

Step-2 Need[i]<=work

Available= 3 3 2

Work=Available=3 3 2

Need[i]<=work

For p0--→7 4 3<=3 3 2[False]

For p1--→1 2 2<=3 3 2[True]

So Work= Work+Allocation

3 3 2+2 0 0=5 3 2

Process **p1** is completed

For p2---→6 0 0<=532[False]

For p3---→0 1 1<=5 3 2[True]

So Work= Work+Allocation

=5 3 2+2 1 1=7 4 3

Process **p3** is completed

For p4----→4 3 1<=7 4 3[True]

So Work= Work+Allocation

=7 4 3+0 0 2=7 4 5

Process **p4** is completed

For p0---→7 4 3<=7 4 5[True]

So Work= Work+Allocation

=7 4 5+0 1 0=7 5 5

Process **p0** is completed

For p2--→6 0 0<=7 5 5[True]

So Work= Work+Allocation

=7 5 5+3 0 2=**10 5 7**

Process **p2** is completed

Safe State & Safe Sequence is p1,p3,p4,p0,p2

4.6 Deadlock Detection

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

4.6.1 Single Instance of Each Resource Type

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a *wait-for graph*.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.

- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

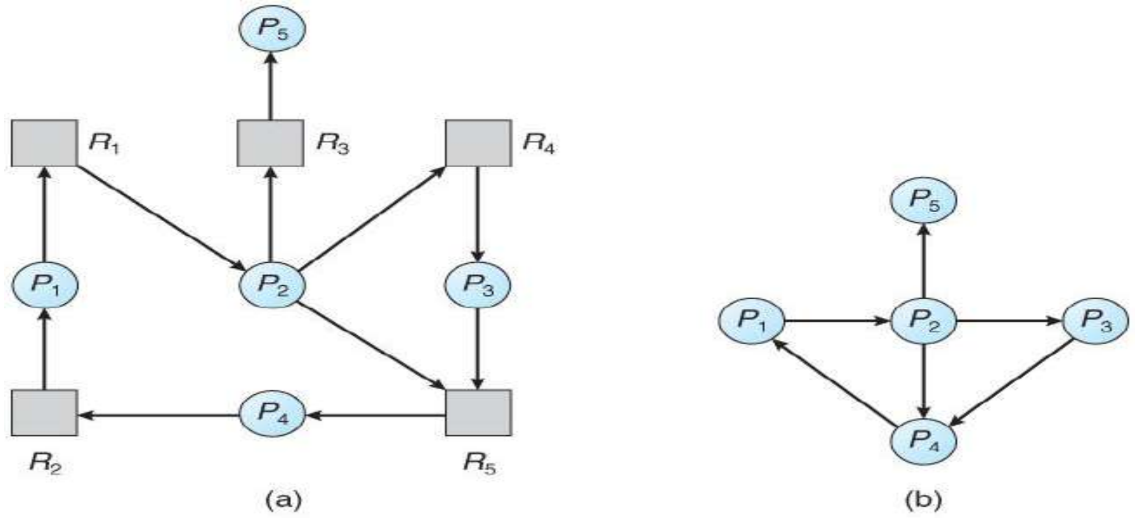


Figure - (a) Resource allocation graph. **(b) Corresponding wait-for graph**

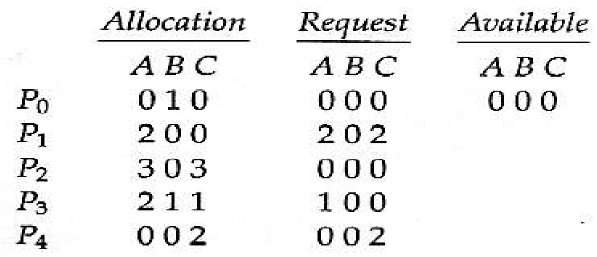
- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

4.6.2 Several Instances of a Resource Type

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:

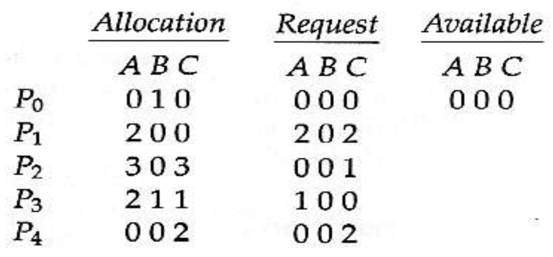
- In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
    - Steps 2 and 3 are unchanged
- In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

- (Note: An alternative method was presented above, in which Finish held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes.)
- Consider, for example, the following state, and determine if it is currently deadlocked:



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

- Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?



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

4.7 Recovery From Deadlock

There are three basic approaches to recovery from deadlock:

1. Inform the system operator, and allow him/her to take manual intervention.
2. Terminate one or more processes involved in the deadlock
3. Preempt resources.

4.7.1 Process Termination

- Two basic approaches, both of which recover resources allocated to terminated processes:
    - Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.

  - Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

- In the latter case there are many factors that can go into deciding which processes to terminate next:

  1. Process priorities.
  2. How long the process has been running, and how close it is to finishing.
  3. How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
  4. How many more resources does the process need to complete.
  5. How many processes will need to be terminated
  6. Whether the process is interactive or batch.
  7. (Whether or not the process has made non-restorable changes to any resource.)

4.7.2 Resource Preemption

When preempting resources to relieve deadlock, there are three important issues to be addressed:

1. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
3. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

4. MEMORY MANAGEMENT

4.1 Background

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

4.1.1 Basic Hardware

- It should be noted that from the memory chips point of view, all memory accesses are equivalent. The memory hardware doesn't know what a particular part of memory is being used for, nor does it care. This is almost true of the OS as well, although not entirely.
- The CPU can only access its registers and main memory. It cannot, for example, make direct access to the hard drive, so any data stored there must first be transferred into the main memory chips before the CPU can work with it. (Device drivers communicate with their hardware via interrupts and "memory" accesses, sending short instructions for example to transfer data from the hard drive to a specified location in main memory. The disk controller monitors the bus for such instructions, transfers the data, and then notifies the CPU that the data is there with another interrupt, but the CPU never gets direct access to the disk.)
- Memory accesses to registers are very fast, generally one clock tick, and a CPU may be able to execute more than one machine instruction per clock tick.
- Memory accesses to main memory are comparatively slow, and may take a number of clock ticks to complete. This would require intolerable waiting by the

CPU if it were not for an intermediary fast memory ***cache*** built into most modern CPUs. The basic idea of the cache is to transfer chunks of memory at a time from the main memory to the cache, and then to access individual memory locations one at a time from the cache.

- User processes must be restricted so that they only access memory locations that "belong" to that particular process. This is usually implemented using a base register and a limit register for each process, as shown in Figures 8.1 and 8.2 below. ***Every*** memory access made by a user process is checked against these two registers, and if a memory access is attempted outside the valid range, then a fatal error is generated. The OS obviously has access to all existing memory locations, as this is necessary to swap users' code and data in and out of memory. It should also be obvious that changing the contents of the base and limit registers is a privileged activity, allowed only to the OS kernel.

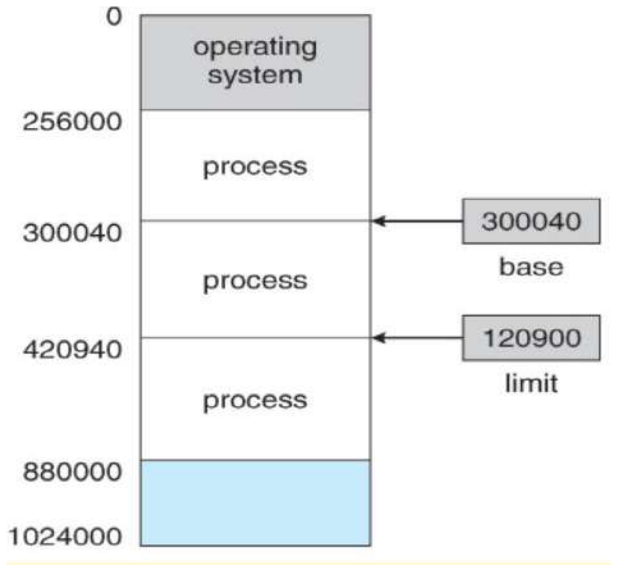


Figure - A base and a limit register define a logical address space

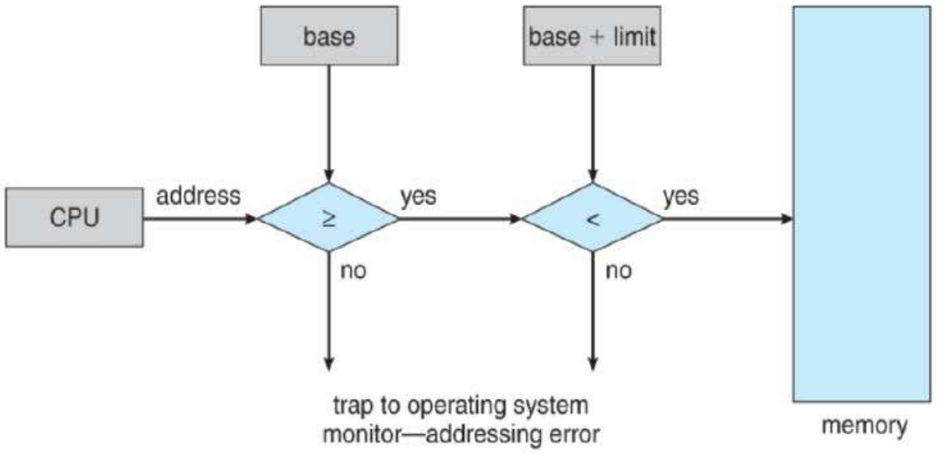


Figure - Hardware address protection with base and limit registers

4.1.2 Address Binding:

Programs are stored on the secondary storage disks as binary executable files. When the programs are to be executed they are brought in to the main memory and placed within a process. The collection of processes on the disk waiting to enter the main memory forms the input queue. One of the processes which are to be executed is fetched from the queue and placed in the main memory. During the execution it fetches instruction and data from main memory. After the process terminates it returns back the memory space. During execution the process will go through different steps and in each step the address is represented in different ways. In source program the address is symbolic. The compiler converts the symbolic address to re-locatable address. The loader will convert this re-locatable address to absolute address.

Binding of instructions and data can be done at any step along the way:

Compile time:-If we know whether the process resides in memory then absolute code can be generated. If the static address changes then it is necessary to re-compile the code from the beginning.

Load time:-If the compiler doesn't know whether the process resides in memory then it generates the re-locatable code. In this the binding is delayed until the load time.

Execution time:-If the process is moved during its execution from one memory segment to another then the binding is delayed until run time. Special hardware is used for this. Most of the general purpose operating system uses this method.

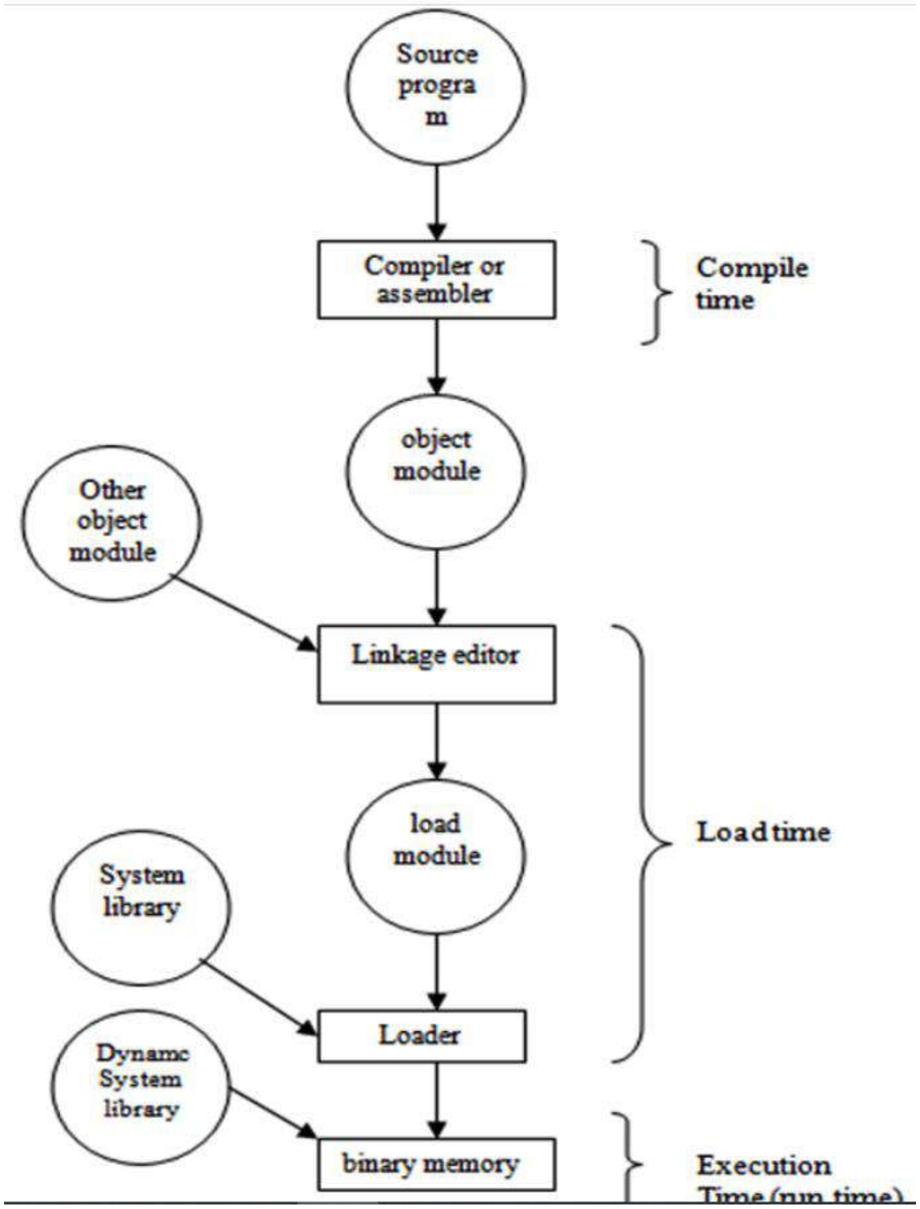


Figure - Multistep processing of a user program

4.1.3 Logical versus physical address

The address generated by the CPU is called logical address or virtual address. The address seen by the memory unit i.e., the one loaded in to the memory register is called the physical address. Compile time and load time address binding methods generate some logical and physical address. The execution time addressing binding generate different logical and physical address. Set of logical address space generated by the programs is the logical address space. Set of physical address corresponding to these logical addresses is the physical address space. The mapping of virtual address to physical address during run time is done by the hardware device called memory management unit (MMU). The base register is also called re-location register. Value of the re-location register is added to every address generated by the user process at the time it is sent to memory.

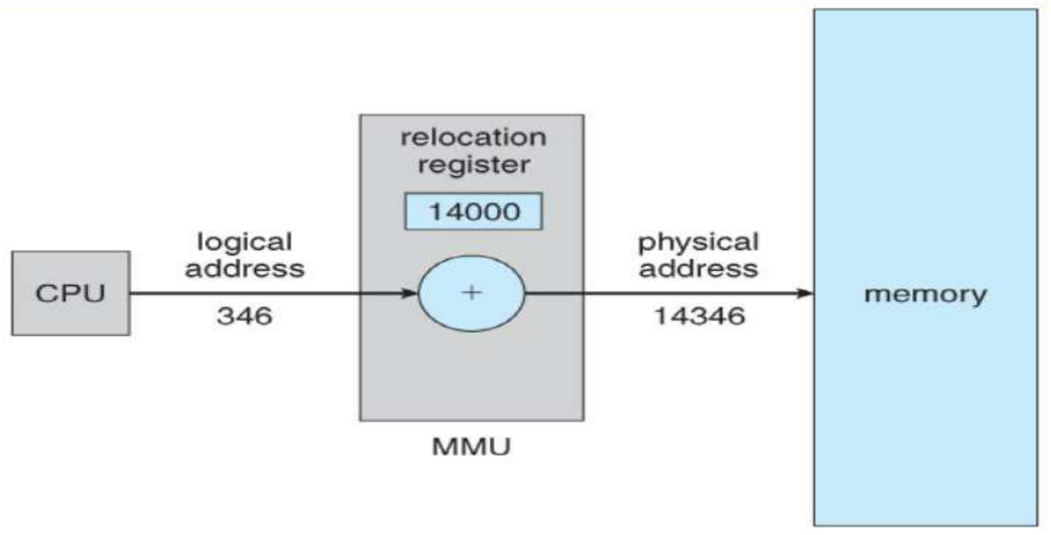


Figure: Dynamic re-location using a re-location registers

The above figure shows that dynamic re-location which implies mapping from virtual addresses space to physical address space and is performed by the hardware at run time. Re-location is performed by the hardware and is invisible to the user dynamic relocation makes it possible to move a partially executed process from one area of memory to another without affecting

BASIS FOR COMPARISON	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	It is the virtual address generated by CPU	The physical address is a location in a memory unit.
Address Space	Set of all logical addresses generated by CPU in reference to a program is referred as Logical Address Space.	Set of all physical address mapped to the correspond logical addresses is referr as Physical Address.
Visibility	The user can view the logical address of a program.	The user can never view physical address of progra
Access	The user uses the logical address to access the physical address.	The user can not directly access physical address.
Generation	The Logical Address is generated by the CPU	Physical Address is Computed by MMU

4.1.4 Dynamic Loading

For a process to be executed it should be loaded in to the physical memory. The size of the process is limited to the size of the physical memory. Dynamic loading is used to obtain better memory utilization. In dynamic loading the routine or procedure will not be loaded until it is called. Whenever a routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is not loaded it cause the loader to load the desired program in to the memory and updates the programs address table to indicate the change and control is passed to newly called routine.

Advantage: Gives better memory utilization. Unused routine is never loaded. Do not need special operating system support. This method is useful when large amount of codes are needed to andle in frequently occurring cases.

4.1.5 Dynamic linking and Shared libraries

Some operating system supports only the static linking. In dynamic linking only the main program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded and the link is established at the time of references. This linking is postponed until the execution time.

With dynamic linking a "stub" is used in the image of each library referenced routine. A "stub" is a piece of code which is used to indicate how to locate the appropriate memory resident library routine or how to load library if the routine is not already present. When "stub" is executed it checks whether the routine is present is memory or not. If not it loads the routine in to the memory. This feature can be used to update libraries i.e., library is replaced by a new version and all the programs can make use of this library. More than one version of the library can be loaded in memory at a time and each program uses its version of the library. Only the programs that are compiled with the new version are affected by the changes incorporated in it. Other programs linked before new version is installed will continue using older libraries this type of system is called "shared library".

4.2 Swapping

Swapping is a technique of temporarily removing inactive programs from the memory of the system. A process can be swapped temporarily out of the memory to a backing store and then brought back in to the memory for continuing the execution. This process is called swapping.

Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum expires then the process that has just finished is swapped out and a new process swaps in to the memory for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high priority process arrives then a low priority will be swapped out and high priority is allowed for execution. This process is also called as Roll out and Roll in.

Normally the process which is swapped out will be swapped back to the same memory space that is occupied previously. This depends upon address binding.

If the binding is done at load time, then the process is moved to same memory location. If the binding is done at run time, then the process is moved to different memory location. This is because the physical address is computed during run time.

Swapping requires backing store and it should be large enough to accommodate the copies of all memory images. The system maintains a ready queue consisting of all the processes whose memory images are on the backing store or in memory that are ready to run. Swapping is constant by other factors: To swap a process, it should be completely idle. A process may be waiting for an i/o operation. If the i/o is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped.

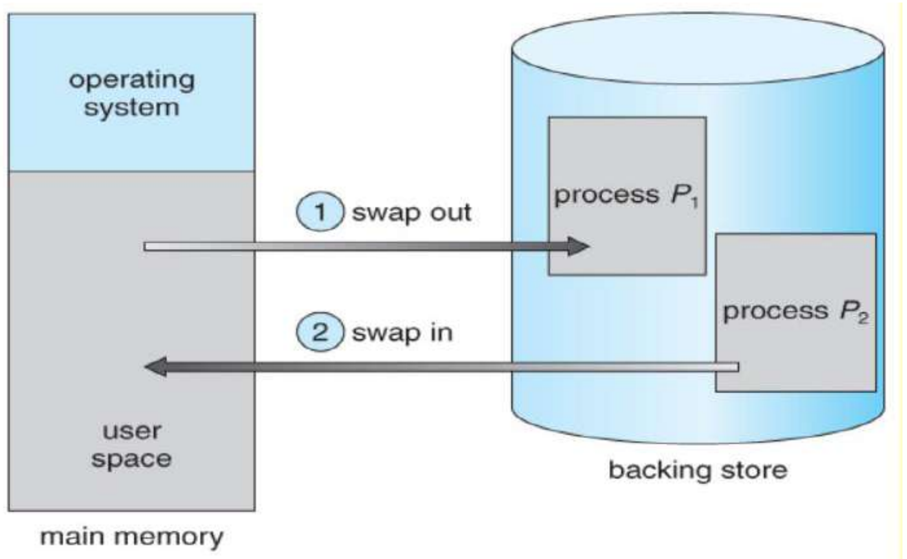


Figure - Swapping of two processes using a disk as a backing store

4.3 Contiguous memory allocation

One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed.

4.3.1 Memory Allocation

One of the simplest method for memory allocation is to divide memory in to several fixed partition. Each partition contains exactly one process. The degree of multi-programming depends on the number of partitions. In multiple partition method, when a partition is free, process is selected from the input queue and is loaded in to free partition of memory. When process terminates, the memory partition becomes available for another process. Batch OS uses the fixed size partition scheme.

The OS keeps a table indicating which part of the memory is free and is occupied. When the process enters the system it will be loaded in to the input queue. The OS keeps track of the memory requirement of each process and the amount of memory available and determines which process to allocate the memory. When a process requests, the OS searches for large hole for this process, hole is a large block of free memory available. If the hole is too large it is split in to two. One part is allocated to the requesting process and other is returned to the set of holes. The set of holes are searched to determine which hole is best to allocate. There are three strategies to select a free hole:

- **First bit**:-Allocates first hole that is big enough. This algorithm scans memory from the beginning and selects the first available block that is large enough to hold the process.
- **Best bit**:-It chooses the hole i.e., closest in size to the request. It allocates the smallest hole i.e., big enough to hold the process.
- **Worst fit**:-It allocates the largest hole to the process request. It searches for the largest hole in the entire list.

Operating Systems

First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is generally faster. Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit reduces the rate of production of smallest holes. All these algorithms suffer from fragmentation.

4.3.2 Memory Protection:

Memory protection means protecting the OS from user process and protecting process from one another. Memory protection is provided by using a re-location register, with a limit register. Re-location register contains the values of smallest physical address and limit register contains range of logical addresses.

(Re-location = 100040 and limit = 74600). The logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in re-location register. When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit register with correct values as a part of context switch. Since every address generated by the CPU is checked against these register, we can protect the OS and other users programs and data from being modified.

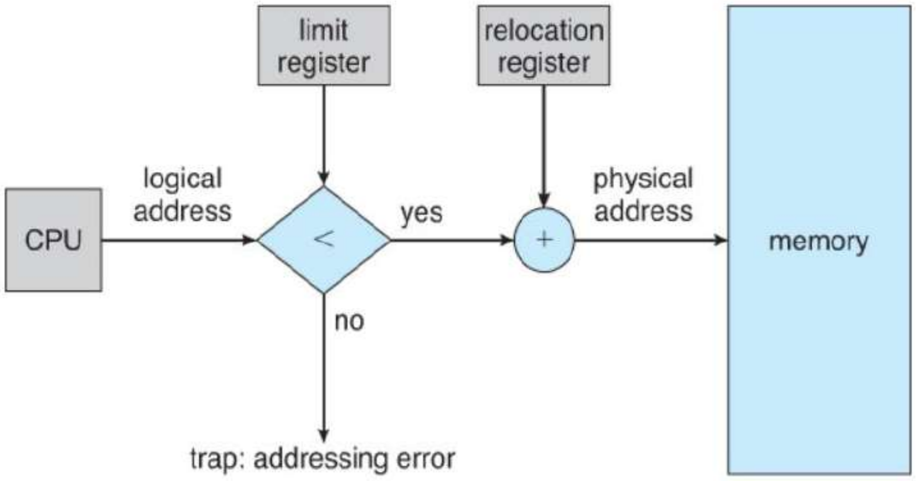


Figure - Hardware support for relocation and limit registers

4.3.3 Fragmentation

Memory fragmentation can be of two types: Internal Fragmentation External Fragmentation

Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data loaded is smaller than the partition.

Eg:-If there is a block of 50kb and if the process requests 40kb and if the block is allocated to the process then there will be 10kb of memory left.

External Fragmentation exists when there is enough memory space exists to satisfy the request, but it not contiguous i.e., storage is fragmented in to large number of small holes.

External Fragmentation may be either minor or a major problem.

One solution for over-coming external fragmentation is **compaction.** The goal is to move all the free memory together to form a large block. Compaction is not possible always. If the relocation is static and is done at load time then compaction is not possible. Compaction is possible if the re-location is dynamic and done at execution time.

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

4.4 Segmentation

Most users do not think memory as a linear array of bytes rather the users thinks memory as a collection of variable sized segments which are dedicated to a particular use such as code, data, stack, heap etc. A logical address is a collection of segments. Each segment has a name and length. The address specifies both the segment name and the offset within the segments. The users specify address by using two quantities: a segment name and an offset. For simplicity the segments are numbered and referred by a segment number. So the logical address consists of <segment number, offset>.

4.4.1 Hardware support

We must define an implementation to map 2D user defined address in to 1D physical address.

This mapping is affected by a segment table. Each entry in the segment table has a segment base and segment limit. The segment base contains the starting physical address where the segment resides and limit specifies the length of the segment.

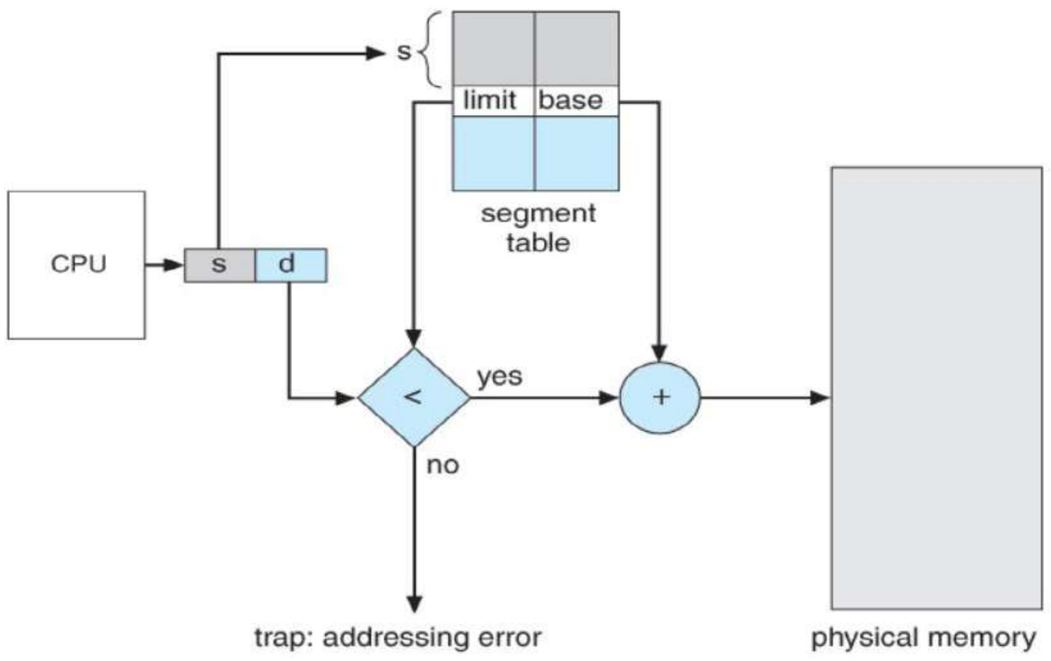
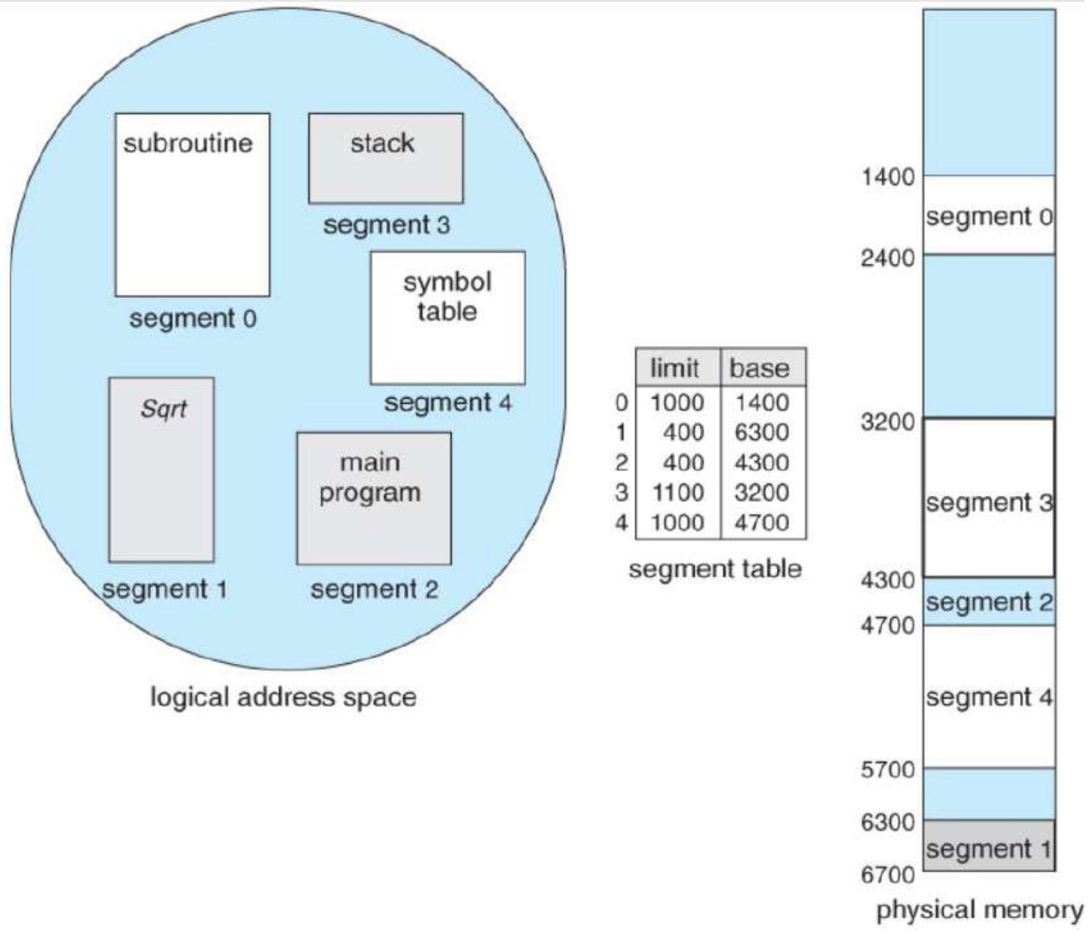


Figure - Segmentation hardware

The use of segment table is shown in the above figure: Logical address consists of two parts: segment number‘s’ and an offset ‘d’ to that segment. The segment number is used as an index to segment table. The offset ’d’ must be in between 0 and limit, if not an error is reported to OS. If legal the offset is added to the base to generate the actual physical address. The segment table is an array of base limit register pairs.



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

Figure - Example of segmentation

4.4.2 Protection and Sharing

A particular advantage of segmentation is the association of protection with the segments. The memory mapping hardware will check the protection bits associated with each segment table entry to prevent illegal access to memory like attempts to write in to read-only segment. Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it.

Segments are shared when the entries in the segment tables of two different processes points to same physical location. Sharing occurs at the segment table. Any information can be shared at the segment level. Several segments can be shared so a program consisting of several segments can be shared. We can also share parts of a program.

Advantages: Eliminates fragmentation. x Provides virtual growth. Allows dynamic segment growth.

Assist dynamic linking. Segmentation is visible

4.4.3 Differences between segmentation and paging

Segmentation:

• Program is divided in to variable sized segments. x User is responsible for dividing the program in to segments.

• Segmentation is slower than paging.

• Visible to user.

• Eliminates internal fragmentation.

• Suffers from external fragmentation.

• Process or user segment number, offset to calculate absolute address.

Paging:

• Programs are divided in to fixed size pages.

• Division is performed by the OS.

• Paging is faster than segmentation.

• Invisible to user.

• Suffers from internal fragmentation.

• No external fragmentation.

• Process or user page number, offset to calculate absolute address

4.5 Paging

Paging is a memory management scheme that permits the physical address space of a process to be non-contiguous. Support for paging is handled by hardware. It is used to avoid external fragmentation. Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing store. When some code or date residing in main memory need to be swapped out, space must be found on backing store.

4.5.1 Basic Method

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

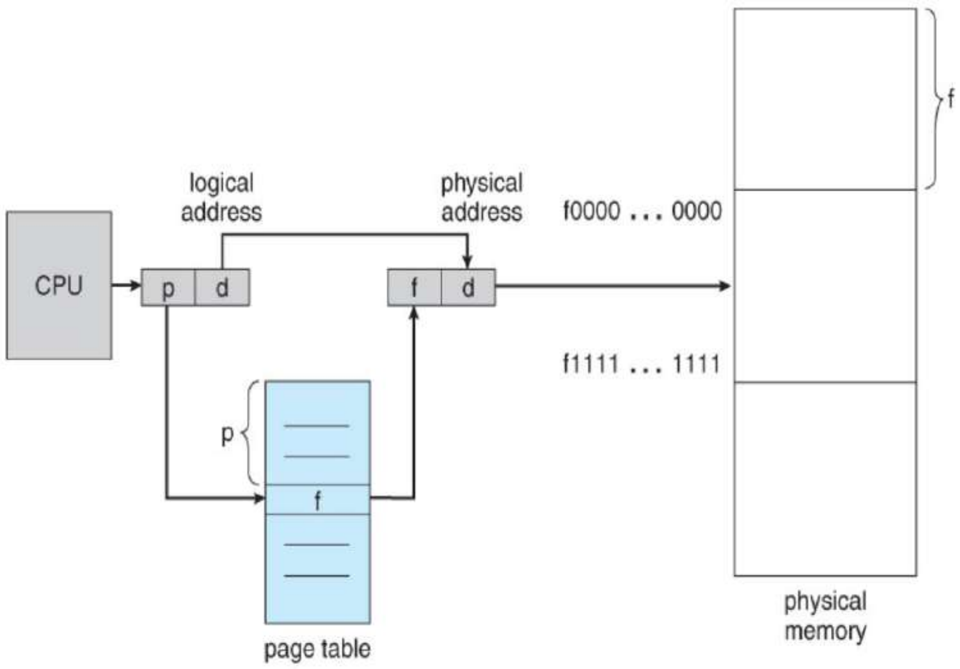


Figure - Paging hardware

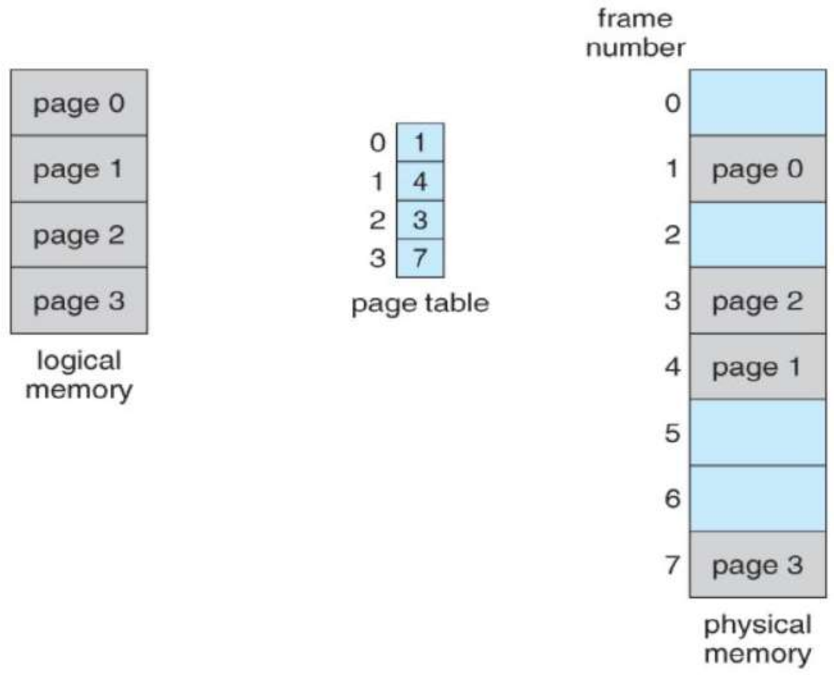
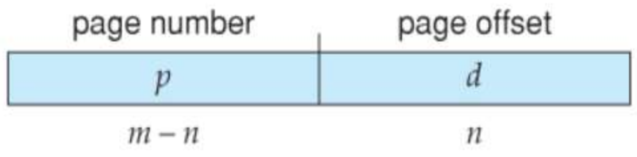


Figure - Paging model of logical and physical memory

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)
- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.
- Note also that the number of bits in the page number and the number of bits in the frame number do not have to be identical. The former determines the address range of the logical address space, and the latter relates to the physical address space.



- (DOS used to use an addressing scheme with 16 bit frame numbers and 16-bit offsets, on hardware that only supported 24-bit hardware addresses. The result was a resolution of starting frame addresses finer than the size of a single frame, and multiple frame-offset combinations that mapped to the same physical hardware address.)
- Consider the following micro example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

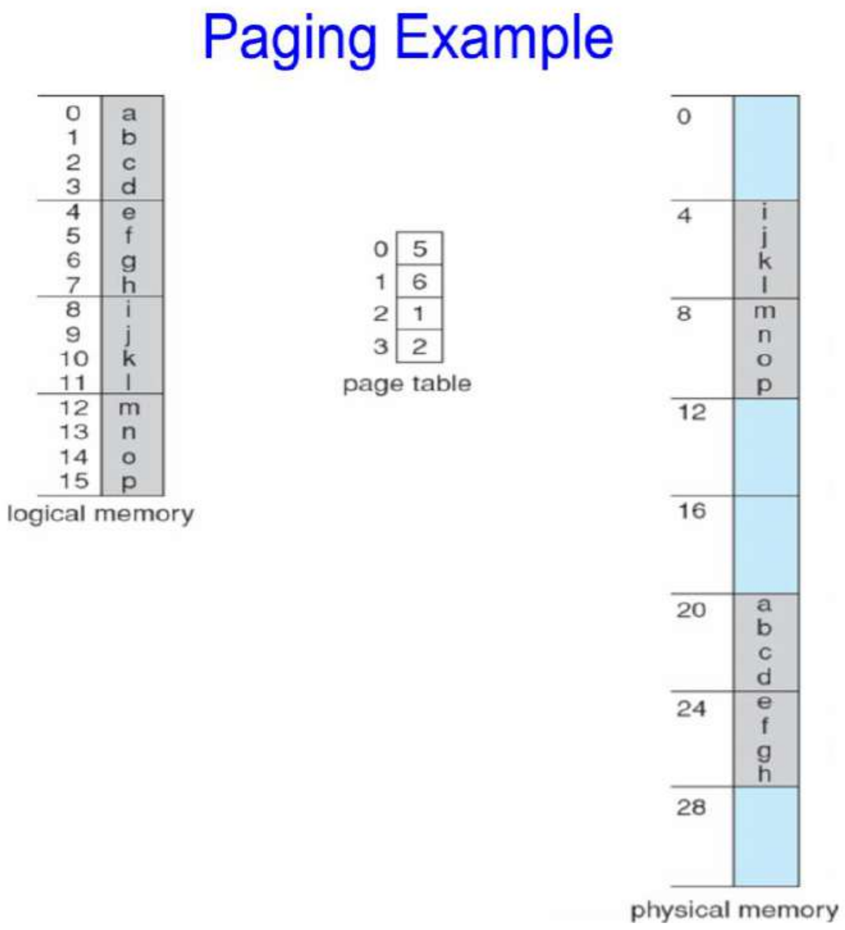


Figure - Paging example for a 32-byte memory with 4-byte pages

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is no external fragmentation with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process. (Possibly more, if processes keep their code and data in separate pages.)
- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.

- Page table entries (frame numbers) are typically 32 bit numbers, allowing access to 2^32 physical page frames. If those frames are 4 KB in size each, that translates to 16 TB of addressable physical memory. (32 + 12 = 44 bits of physical address space.)
- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

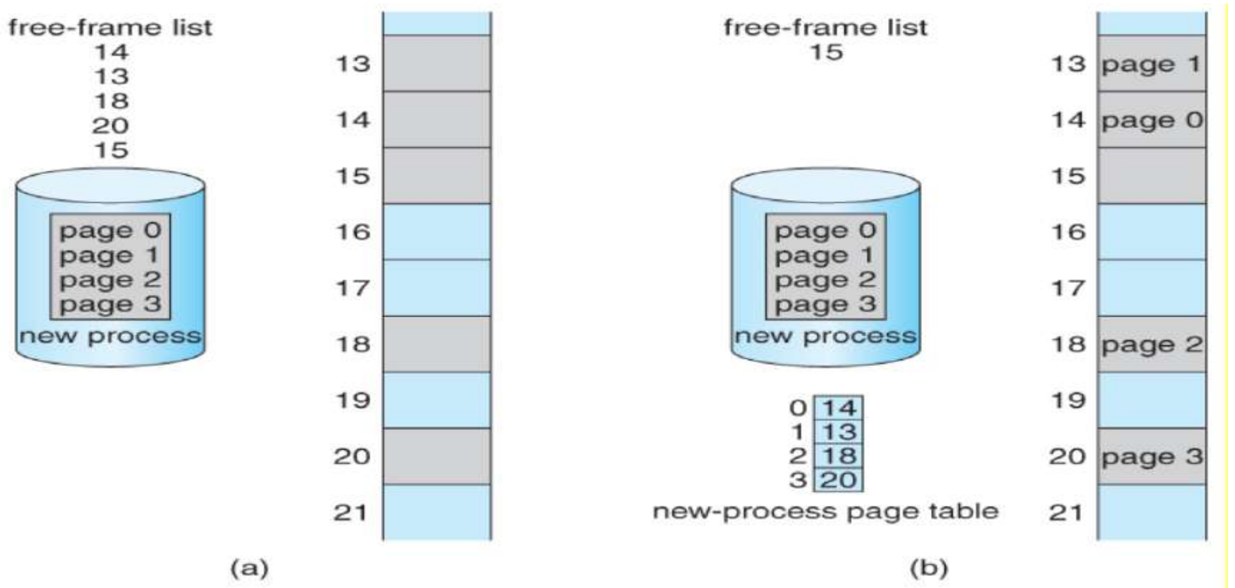


Figure - Free frames (a) before allocation and (b) after allocation

4.5.2 Hardware Support for Paging

The hardware implementation of the page table can be done in several ways:

The simplest method is that the page table is implemented as a set of dedicated registers. These registers must be built with very high speed logic for making paging address translation. Every accessed memory must go through paging map. The use of registers for page table is satisfactory if the page table is small.

If the page table is large then the use of registers is not visible. So the page table is kept in the main memory and a page table base register [PTBR] points to the page table. Changing the page table requires only one register which reduces the context switching type. The problem with this approach is the time required to access memory location. To access a location [i] first we have to index the page table using PTBR offset. It gives the frame number which is combined with the page offset to produce the actual address. Thus we need two memory accesses for a byte.

The only solution is to use special, fast, lookup hardware cache called translation look aside buffer [TLB] or associative register. LB is built with associative register with high speed memory. Each register contains two paths a key and a value.

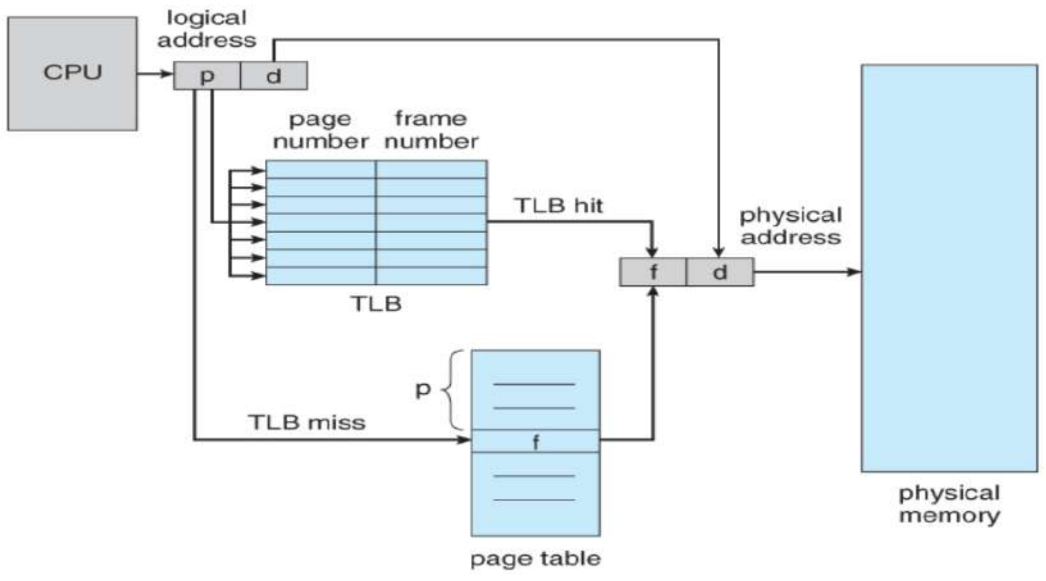


Figure - Paging hardware with TLB

When an associative register is presented with an item, it is compared with all the key values, if found the corresponding value field is return and searching is fast. TLB is used with the page table as follows: TLB contains only few page table entries. When a logical address is generated by the CPU, its page number along with the frame number is added to TLB. If the page number is found its frame memory is used to access the actual memory. If the page number is not in the TLB (TLB miss) the memory reference to the page table is made. When the frame number is obtained use can use it to access the memory. If the TLB is full of entries the OS must select anyone for replacement. Each time a new page table is selected the TLB must be flushed [erased] to ensure that next executing process do not use wrong information. The percentage of time that a page number is found in the TLB is called HIT ratio.

4.5.3 Protection

Memory protection in paged environment is done by protection bits that are associated with each frame these bits are kept in page table. x One bit can define a page to be read-write or read-only. To find the correct frame number every reference to the memory should go through page table. At the same time physical address is computed. The protection bits can be checked to verify that no writers are made to read-only page. Any attempt to write in to read-only page causes a hardware trap to the OS. This approach can be used to provide protection to read-only, read-write or execute-only pages. One more bit is generally added to each entry in the page table: a valid-invalid bit.

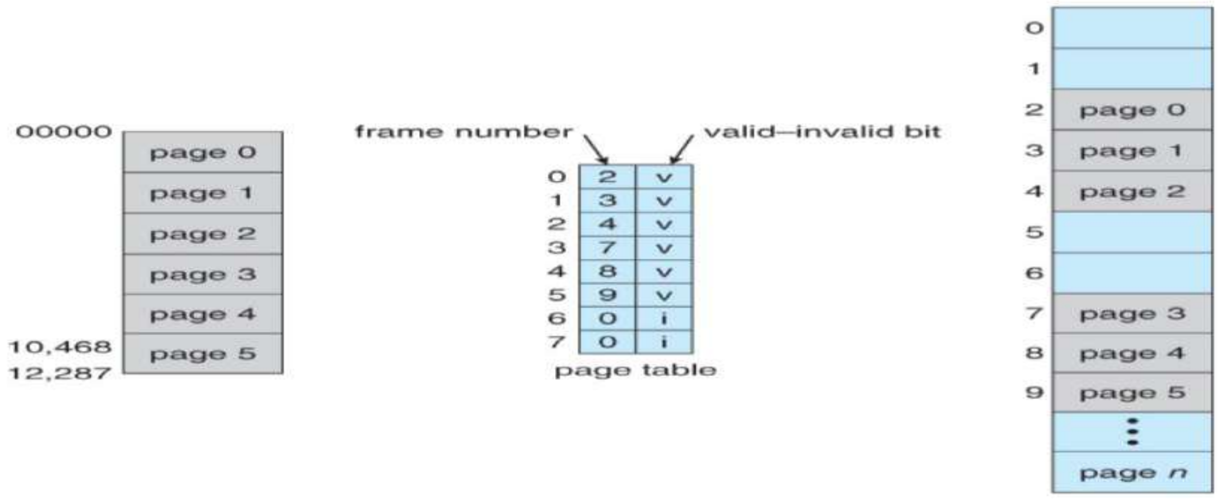


Figure - Valid (v) or invalid (i) bit in page table

A valid bit indicates that associated page is in the processes logical address space and thus it is a legal or valid page.

If the bit is invalid, it indicates the page is not in the processes logical addressed space and illegal. Illegal

addresses are trapped by using the valid-invalid bit.

The OS sets this bit for each page to allow or disallow accesses to that page.

4.5.4 Shared Pages

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

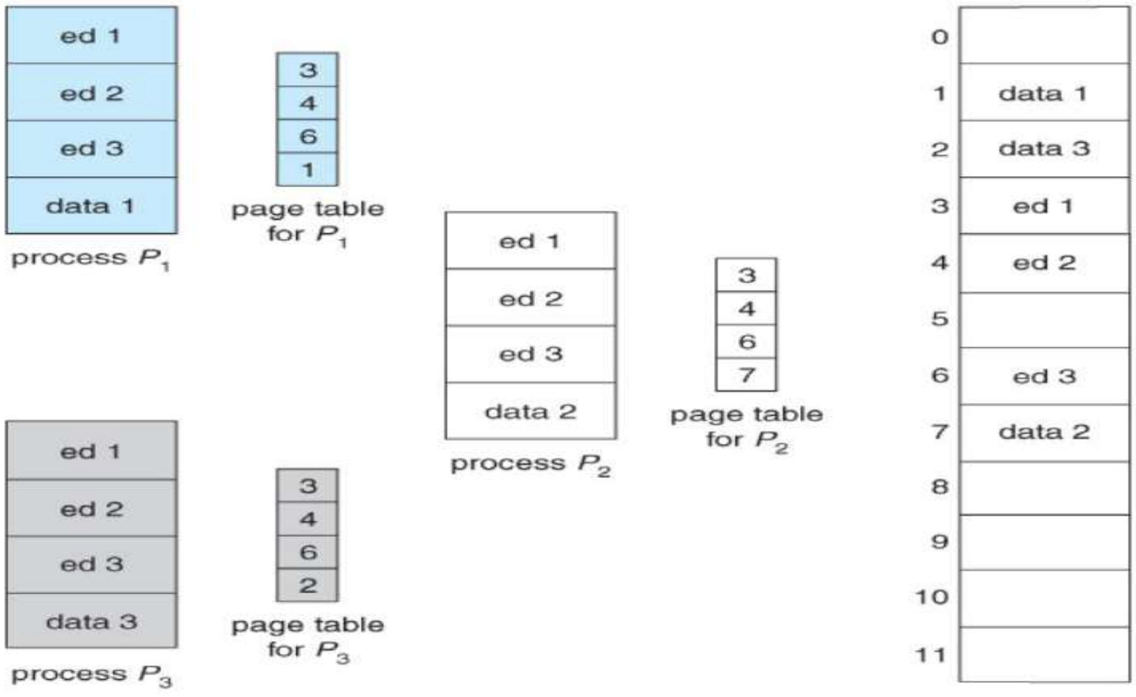


Figure - Sharing of code in a paging environment

4.6 Structure of the Page Table

4.6.1 Hierarchical Paging

Recent computer system support a large logical address apace from 2^32 to 2^64. In this system the page table becomes large. So it is very difficult to allocate contiguous main memory for page table. One simple solution to this problem is to divide page table in to smaller pieces. There are several ways to accomplish this division.

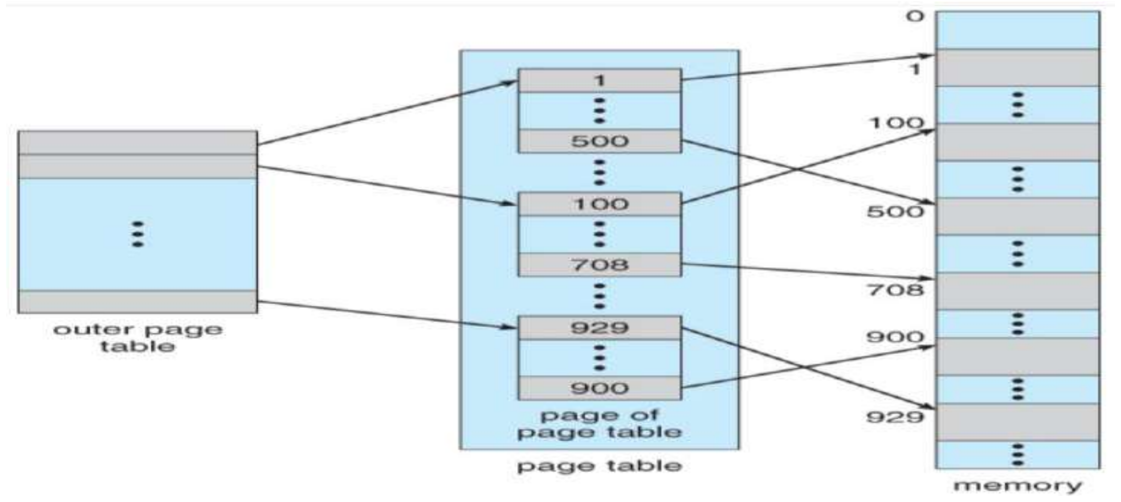


Figure A two-level page-table scheme

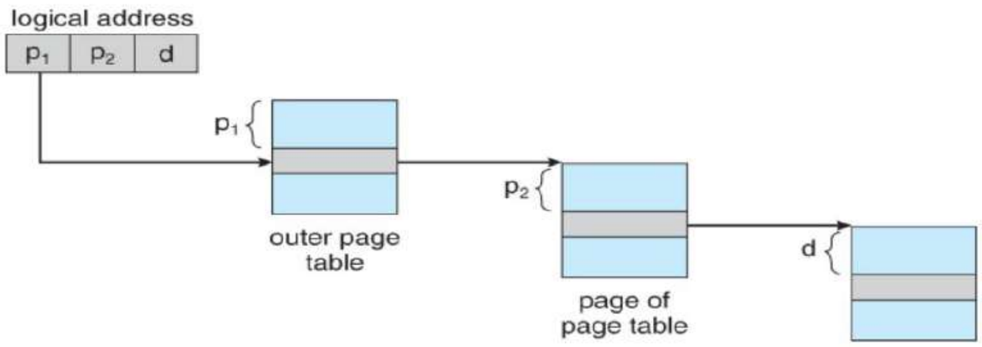


Figure - Address translation for a two-level 32-bit paging architecture

One way is to use **two-level paging algorithm** in which the page table itself is also paged. Eg:- In a 32 bit machine with page size of 4kb. A logical address is divided in to a page number consisting of 20 bits and a page offset of 12 bit. The page table is further divided since the page table is paged, the page number is further divided in to 10 bit page number and a 10 bit offset.

4.6.2 Hashed page table:

Hashed page table handles the address space larger than 32 bit. The virtual page number is used as hashed value. Linked list is used in the hash table which contains a list of elements that hash to the same location.

Each element in the hash table contains the following three fields: Virtual page number x Mapped page frame value x Pointer to the next element in the linked list

Working

Virtual page number is taken from virtual address. Virtual page number is hashed in to hash table.

Virtual page number is compared with the first element of linked list. Both the values are matched, that value is (page frame) used for calculating the physical address. If not match then entire linked list is searched for matching virtual page number. Clustered pages are similar to hash table but one difference is that each entity in the hash table refer to several pages.

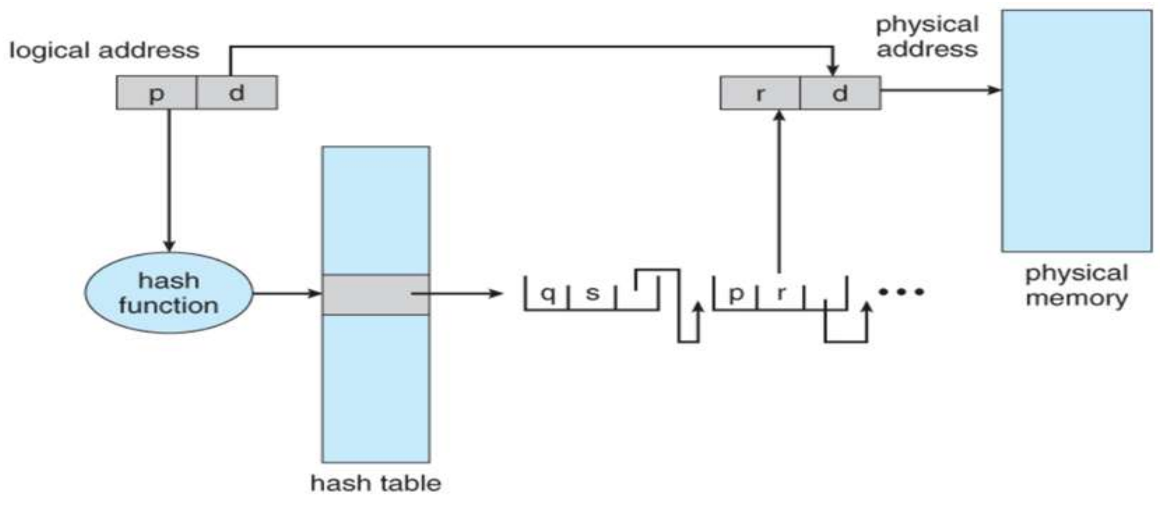


Figure - Hashed page table

4.6.3 Inverted Page Tables:

Since the address spaces have grown to 64 bits, the traditional page tables become a problem. Even with two level page tables. The table can be too large to handle. An inverted page table has only entry for each page in memory. Each entry consisted of virtual address of the page stored in that read-only location with information about the process that owns that page.

Each virtual address in the Inverted page table consists of triple <process-id , page number , offset >. The inverted page table entry is a pair <process-id , page number>. When a memory reference is made, the part of virtual address i.e., <process-id , page number> is presented in to memory sub-system.

The inverted page table is searched for a match. If a match is found at entry I then the physical address <i, offset> is generated. If no match is found then an illegal address access has been attempted.

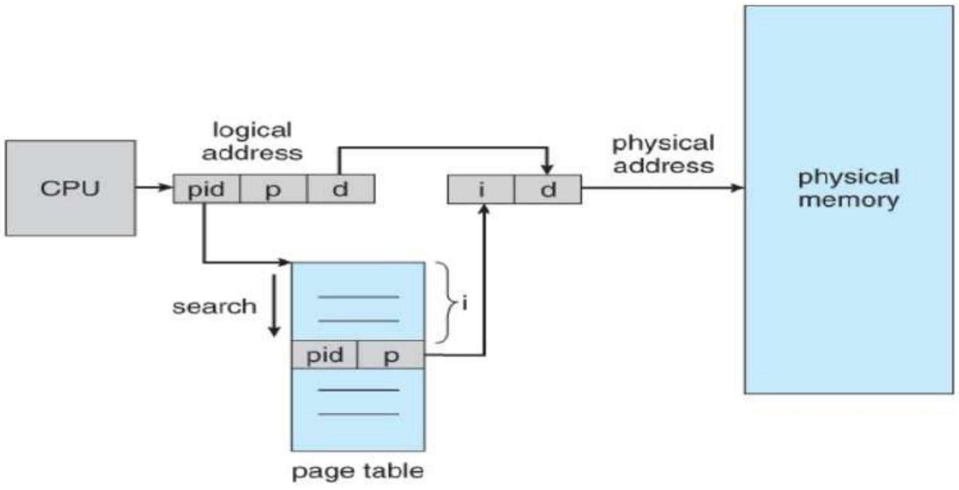


Figure - Inverted page table

This scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. If the whole table is to be searched it takes too long.

Advantage:

• Eliminates fragmentation.

• Support high degree of multiprogramming.

• Increases memory and processor utilization.

• Compaction overhead required for the re-locatable partition scheme is also eliminated.

Disadvantage:

• Page address mapping hardware increases the cost of the computer.

• Memory must be used to store the various tables like page tables, memory map table etc.

• Some memory will still be unused if the number of available block is not sufficient for the address space of the jobs to be run.

Virtual Memory

5.1 Background

- Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.
- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
    1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
    2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
    3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. :-)
- The ability to load only the portions of processes that were actually needed (and only *when* they were needed) has several benefits:
    1. Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
    2. Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
    3. Less I/O is needed for swapping processes in and out of RAM, speeding things up.
- Figure 5.1 shows the general layout of ***virtual memory***, which can be much larger than physical memory:

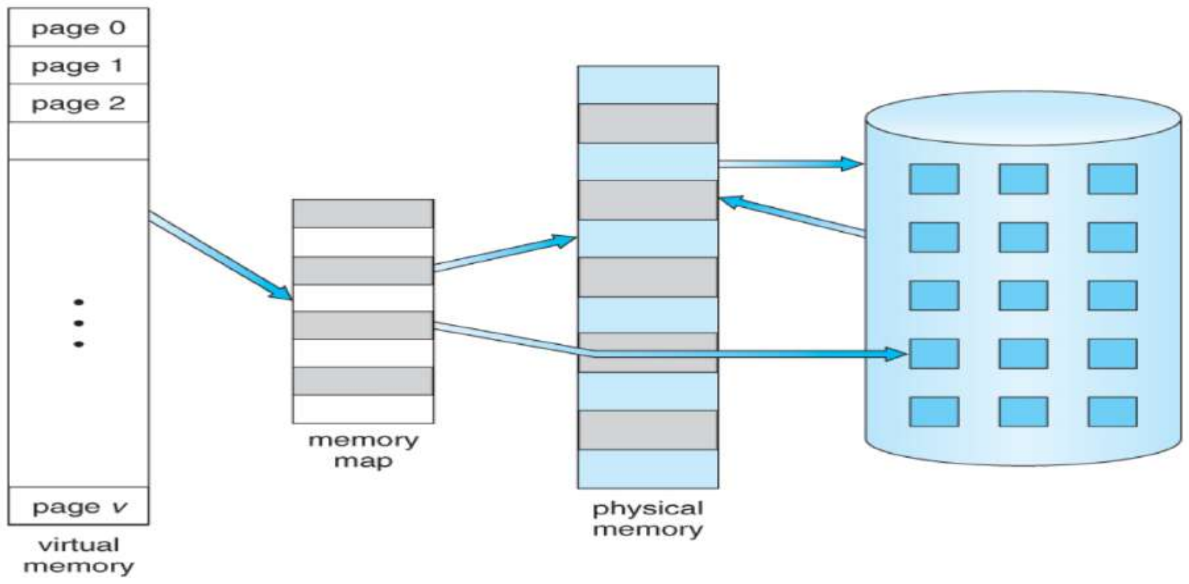


Figure 5.1 - Diagram showing virtual memory that is larger than physical memory

- Figure 5.2 shows ***virtual address space***, which is the programmers logical view of process memory storage. The actual physical layout is controlled by the process's page table.
- Note that the address space shown in Figure 5.2 is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

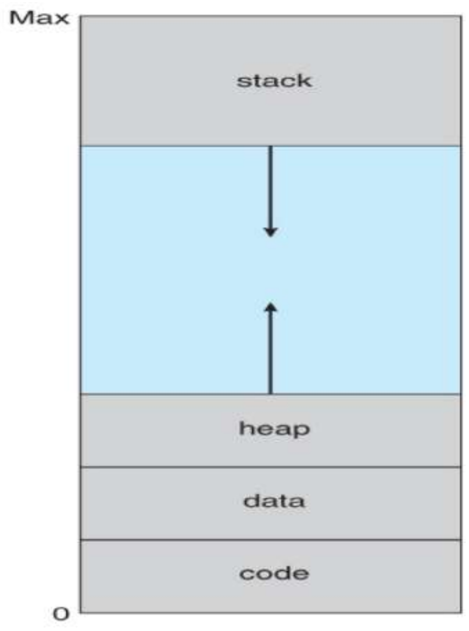


Figure 5.2 - Virtual address space

Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:

- System libraries can be shared by mapping them into the virtual address space of more than one process.
- Processes can also share virtual memory by mapping the same block of memory to more than one process.
- Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original (parent) process.

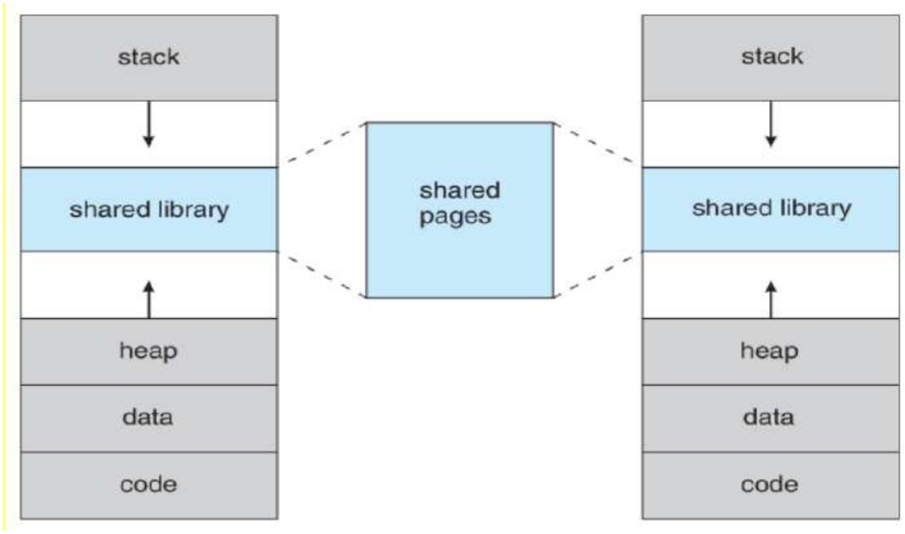


Figure 5.3 - Shared library using virtual memory

5.2 Demand Paging

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

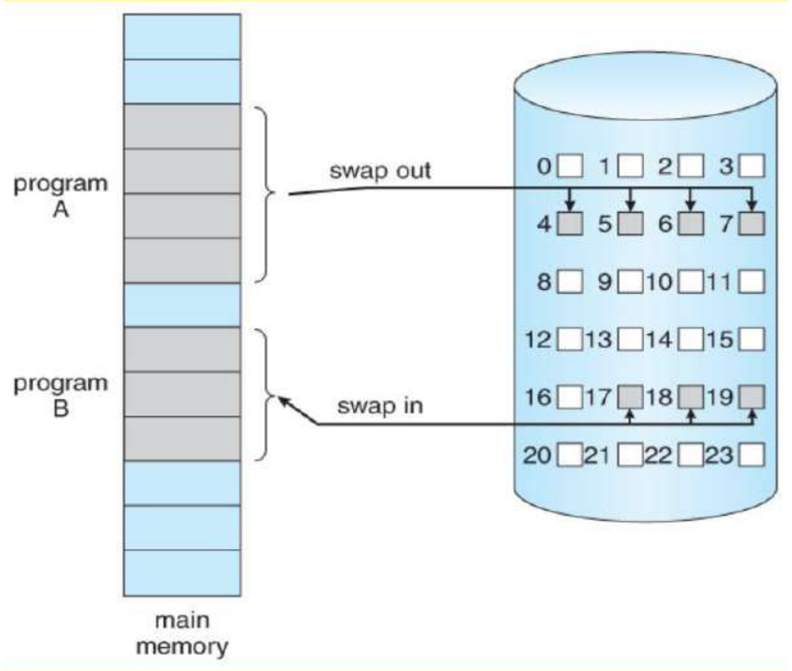


Figure 5.4 - Transfer of a paged memory to contiguous disk space

5.2.1 Basic Concepts

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- If the process only ever accesses pages that are loaded in memory (***memory resident*** pages), then the process runs exactly as if all the pages were loaded in to memory.

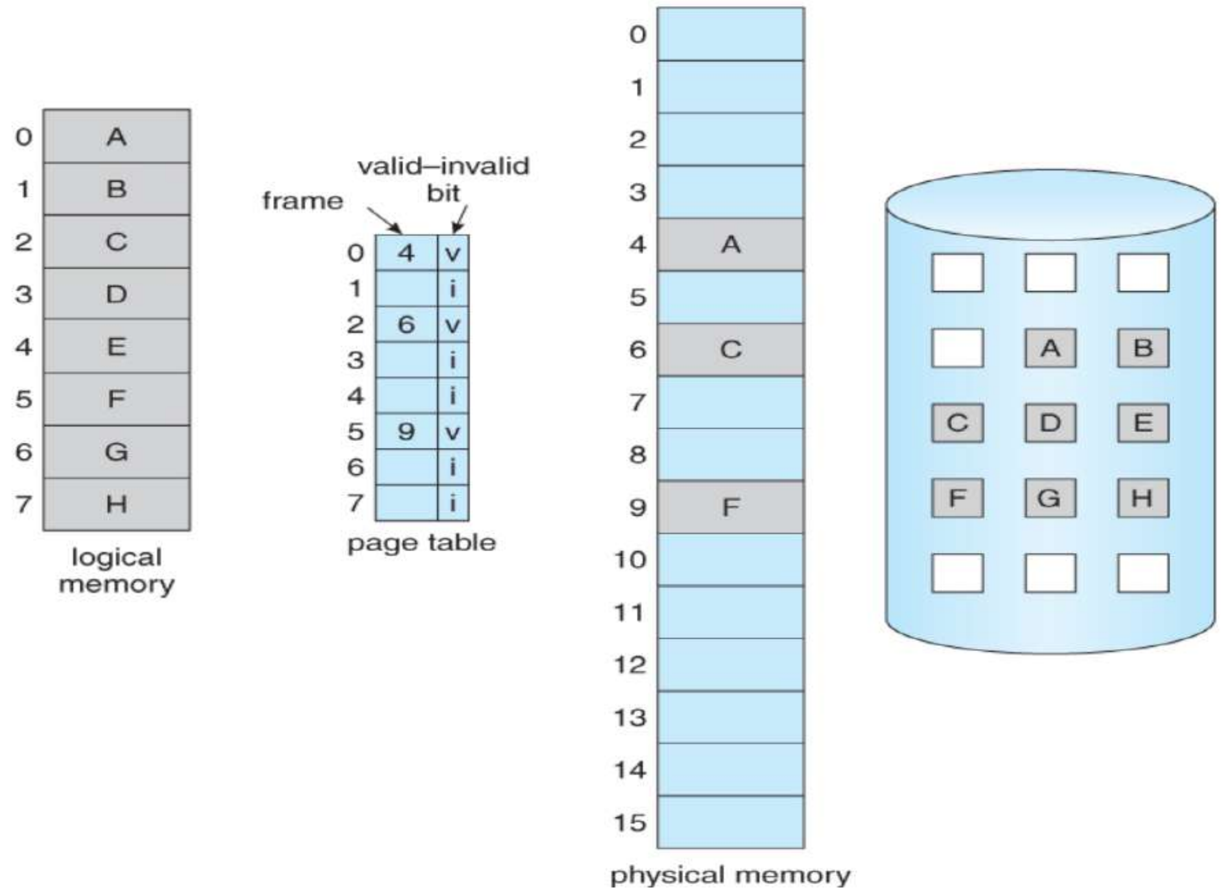


Figure 5.5 - Page table when some pages are not in main memory.

On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:

1. The memory address requested is first checked, to make sure it was a valid memory request.
2. If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.
3. A free frame is located, possibly from a free-frame list.
4. A disk operation is scheduled to bring in the necessary page from disk. (This will usually block the process on an I/O wait, allowing some other process to use the CPU in the meantime.)
5. When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.

6. The instruction that caused the page fault must now be restarted from the beginning, (as soon as this process gets another turn on the CPU.)

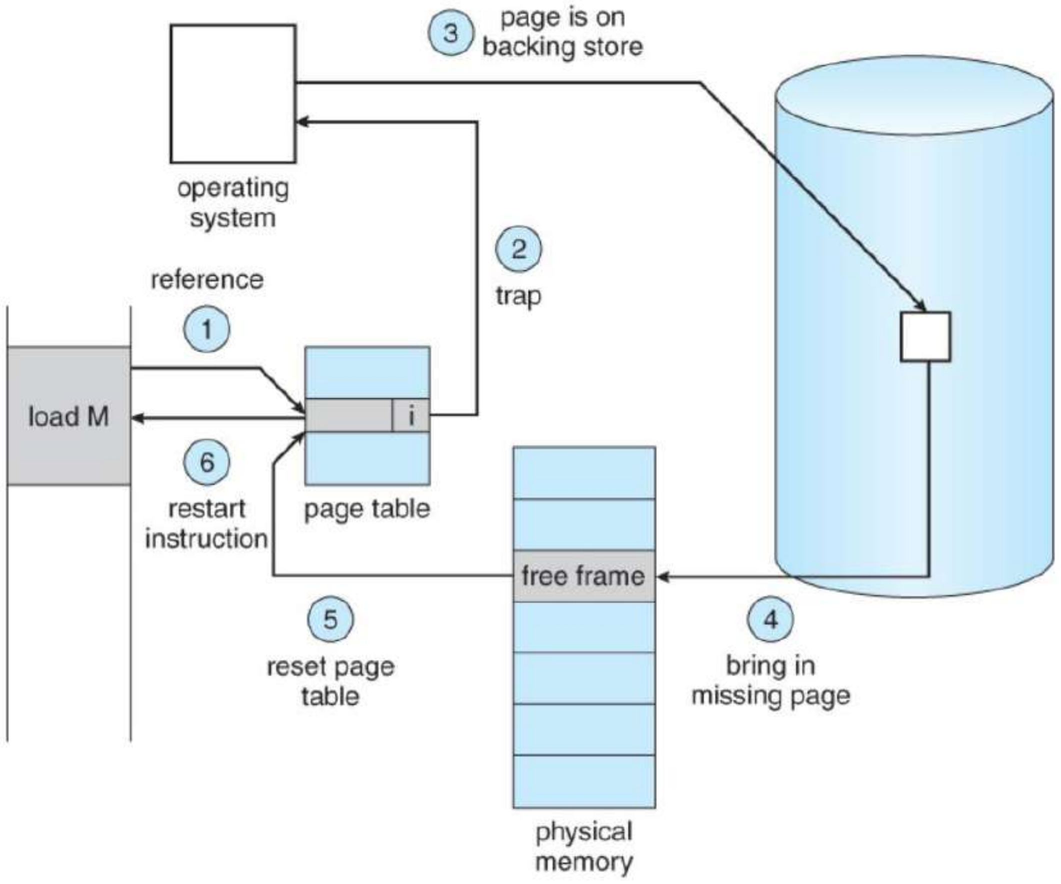


Figure 5.6 - Steps in handling a page fault

- In an extreme case, NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- In theory each instruction could generate multiple page faults. In practice this is very rare, due to ***locality of reference***, covered in section 5.6.1.
- The hardware necessary to support virtual memory is the same as for paging and swapping: A page table and secondary memory.
- A crucial part of the process is that the instruction must be restarted from scratch once the desired page has been made available in memory. For most simple instructions this is not a major difficulty. However there are some architectures that allow a single instruction to modify a fairly large block of data, and if some of the data gets modified before the page

fault occurs, this could cause problems. One solution is to access both ends of the block before executing the instruction, guaranteeing that the necessary pages get paged in before the instruction begins.

5.3 Page Replacement Algorithm

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

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

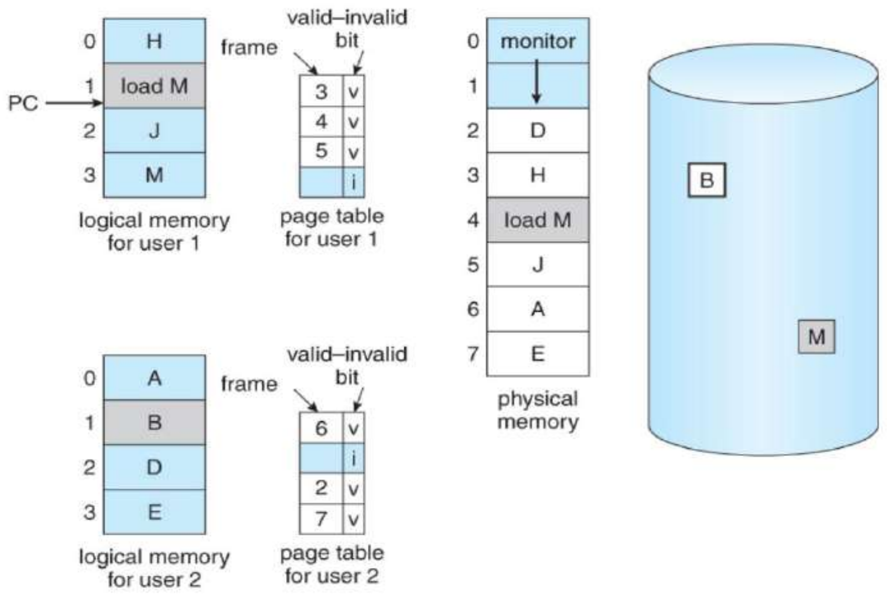


Figure 5.7 - Ned for page replacement.

5.3.1 Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:
    1. Find the location of the desired page on the disk, either in swap space or in the file system.
    2. Find a free frame:
        a. If there is a free frame, use it.
        b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the ***victim frame***.

c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.

3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
4. Restart the process that was waiting for this page.

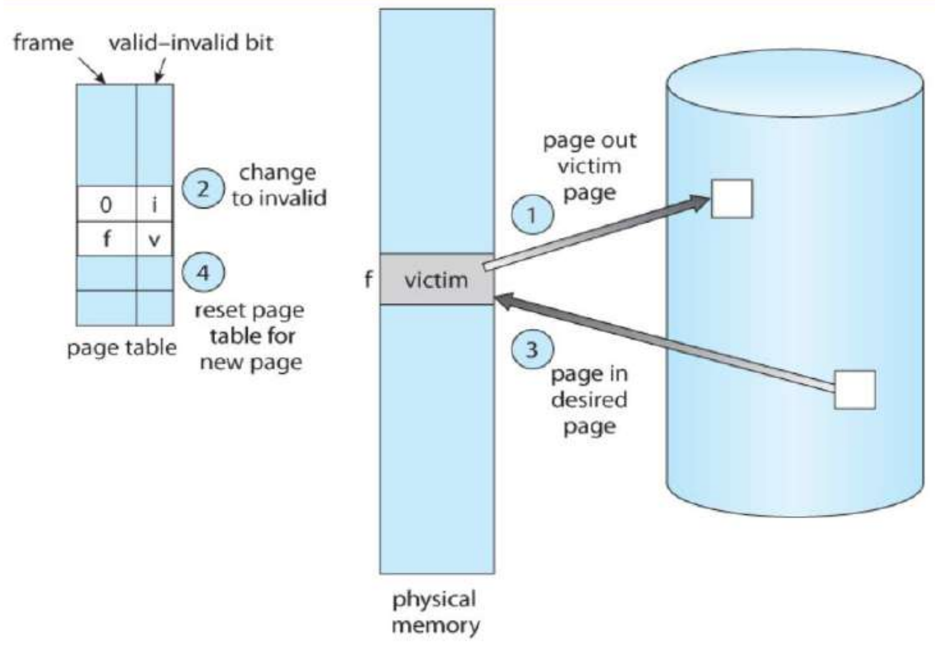


Figure 5.8 - Page replacement.

- Note that step 3c adds an extra disk write to the page-fault handling, effectively doubling the time required to process a page fault. This can be alleviated somewhat by assigning a ***modify bit,*** or ***dirty bit*** to each page, indicating whether or not it has been changed since it was last loaded in from disk. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. It should come as no surprise that many page replacement strategies specifically look for pages that do not have their dirty bit set, and preferentially select clean pages as victim pages. It should also be obvious that unmodifiable code pages never get their dirty bits set.

- There are two major requirements to implement a successful demand paging system. We must develop a ***frame-allocation algorithm*** and a ***page-replacement algorithm.*** The former centers around how many frames are allocated to each process (and to other needs), and the latter deals with how to select a page for replacement when there are no free frames available.
- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of memory accesses known as a ***reference string***.

5.3.2 FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim. In the following example, 20 page requests result in 15 page faults:
- Although FIFO is simple and easy, it is not always optimal, or even efficient.

Belady's anomaly– Belady’s anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

For Example

Consider the page reference string of size 12: 1, 2, 3, 4, 5, 1, 3, 1, 6, 3, 2, 3 with frame size 4(i.e. maximum 4 pages in a frame).

1	2	3	4	5	1	3	1	6	3	2	3

1	1	1	1	5	5	5	5	5	5	2	2
	2	2	2	2	1	1	1	1	1	1	1
		3	3	3	3	3	3	6	6	6	6
			4	4	4	4	4	4	3	3	3

M	M	M	M	M	M	H	H	M	M	M	H

M = Miss
H = Hit

Total Page Fault = 9

Initially, all 4 slots are empty, so when 1, 2, 3, 4 came they are allocated to the empty slots in order of their arrival. This is page fault as 1, 2, 3, 4 are not available in memory.

When 5 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 1.

When 1 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 2.

When 3,1 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

When 6 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 3.

When 3 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 4.

When 2 comes, it is not available in memory so page fault occurs and it replaces the oldest page in memory, i.e., 5.

When 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

Page Fault ratio = 9/12 i.e. total miss/total possible cases

Advantages

- Simple and easy to implement.
- Low overhead.

Disadvantages

- Poor performance.
- Doesn’t consider the frequency of use or last used time, simply replaces the oldest page.
- Suffers from Belady’s Anomaly(i.e. more page faults when we increase the number of page frames).

5.3.3 Optimal Page Replacement

Optimal Page Replacement algorithm is the best page replacement algorithm as it gives the least number of page faults. It is also known as OPT, clairvoyant replacement algorithm, or Belady’s optimal page replacement policy.

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future, i.e., the pages in the memory which are going to be referred farthest in the future are replaced.

This algorithm was introduced long back and is difficult to implement because it requires future knowledge of the program behaviour. However, it is possible to implement optimal page replacement on the second run by using the page reference information collected on the first run.

For Example

1	2	3	4	5	1	3	1	6	3	2	3

1	1	1	1	1	1	1	1	6	6	6	6
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3
			4	5	5	5	5	5	5	5	5

M	M	M	M	M	H	H	H	M	H	H	H

M = Miss
H = Hit

Total Page Fault = 6

Initially, all 4 slots are empty, so when 1, 2, 3, 4 came they are allocated to the empty slots in order of their arrival. This is page fault as 1, 2, 3, 4 are not available in memory.

When 5 comes, it is not available in memory so page fault occurs and it replaces 4 which is going to be used farthest in the future among 1, 2, 3, 4.

When 1,3,1 comes, they are available in the memory, i.e., Page Hit, so no replacement occurs.

When 6 comes, it is not available in memory so page fault occurs and it replaces 1.

When 3, 2, 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

Page Fault ratio = 6/12

Advantages

- Easy to Implement.
- Simple data structures are used.
- Highly efficient.

Disadvantages

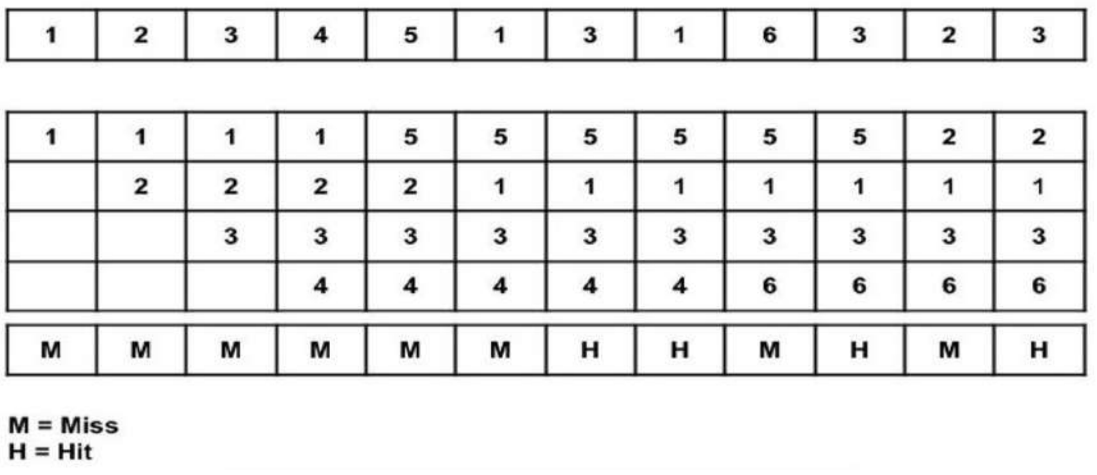
- Requires future knowledge of the program.
- Time-consuming.

5.3.4 LRU Page Replacement

Least Recently Used page replacement algorithm keeps track of page usage over a short period of time. It works on the idea that the pages that have been most heavily used in the past are most likely to be used heavily in the future too.

In LRU, whenever page replacement happens, the page which has not been used for the longest amount of time is replaced.

For Example



1	2	3	4	5	1	3	1	6	3	2	3

1	1	1	1	5	5	5	5	5	5	2	2
	2	2	2	2	1	1	1	1	1	1	1
		3	3	3	3	3	3	3	3	3	3
			4	4	4	4	4	6	6	6	6

M	M	M	M	M	M	H	H	M	H	M	H

M = Miss
H = Hit

Total Page Fault = 8

Initially, all 4 slots are empty, so when 1, 2, 3, 4 came they are allocated to the empty slots in order of their arrival. This is page fault as 1, 2, 3, 4 are not available in memory.

When 5 comes, it is not available in memory so page fault occurs and it replaces 1 which is the least recently used page.

When 1 comes, it is not available in memory so page fault occurs and it replaces 2.

When 3,1 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

When 6 comes, it is not available in memory so page fault occurs and it replaces 4.

When 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

When 2 comes, it is not available in memory so page fault occurs and it replaces 5.

When 3 comes, it is available in the memory, i.e., Page Hit, so no replacement occurs.

Page Fault ratio = 8/12

Advantages

- Efficient.
- Doesn't suffer from Belady’s Anomaly.

Disadvantages

- Complex Implementation.
- Expensive.
- Requires hardware support.

5.3.5 Second-Chance Algorithm

The ***second chance algorithm*** is essentially a FIFO, except the reference bit is used to give pages a second chance at staying in the page table.

- When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner.
- If a page is found with its reference bit not set, then that page is selected as the next victim.
- If, however, the next page in the FIFO **does** have its reference bit set, then it is given a second chance:
  - The reference bit is cleared, and the FIFO search continues.
  - If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table.
  - If , however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.
- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.

- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the clock algorithm, from the hands of the clock moving around the circular queue.

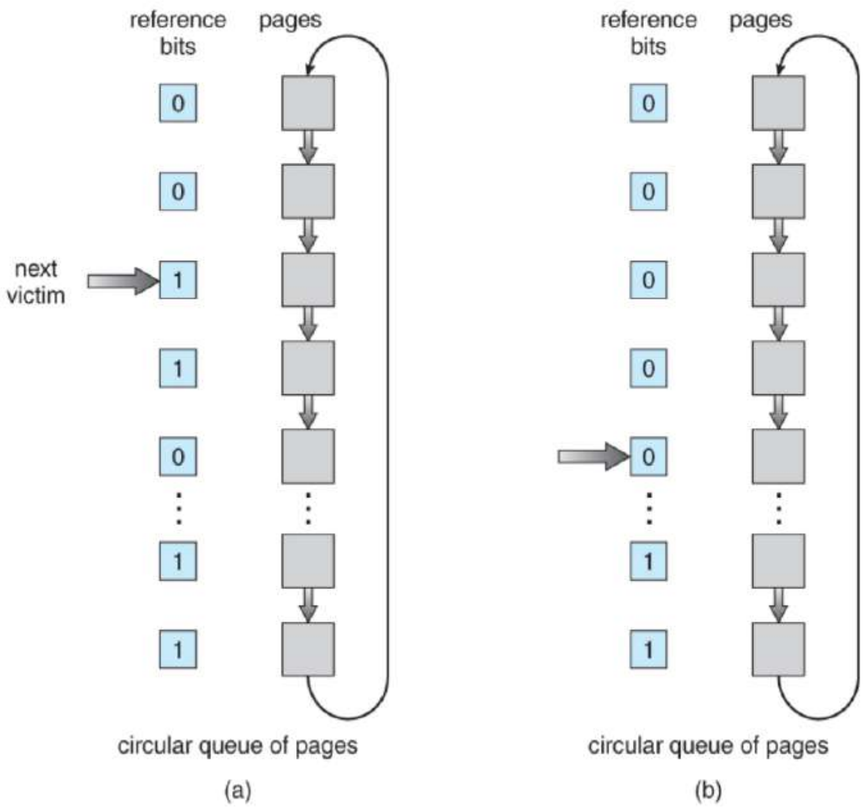


Figure 5.9 - Second-chance (clock) page-replacement algorithm.

5.3.6 Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:
    1. (0, 0) - Neither recently used nor modified.
    2. (0, 1) - Not recently used, but modified.
    3. (1, 0) - Recently used, but clean.
    4. (1, 1) - Recently used and modified.
- This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. I.e. it first makes a

pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.

- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

5.3.7 Counting-Based Page Replacement

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

5.4 Thrashing

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

5.4.1 Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.

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

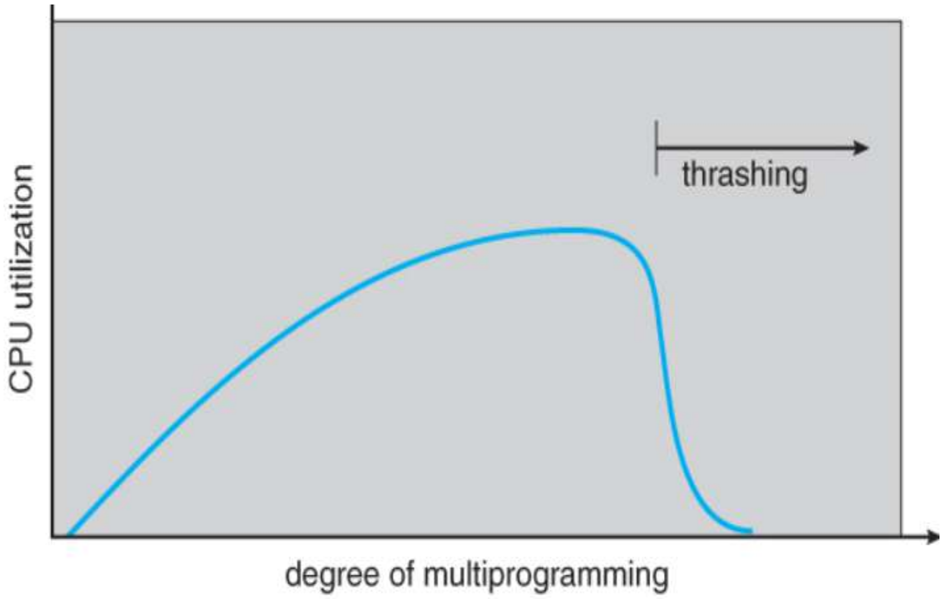


Figure 5.10 – Thrashing

- To prevent thrashing we must provide processes with as many frames as they really need "right now", but how do we know what that is?
- The ***locality model*** notes that processes typically access memory references in a given ***locality,*** making lots of references to the same general area of memory before moving periodically to a new locality, If we could just keep as many frames as are involved in the current locality, then page faulting would occur primarily on switches from one locality to another. (E.g. when one function exits and another is called.)

5.4.1 Working-Set Model

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

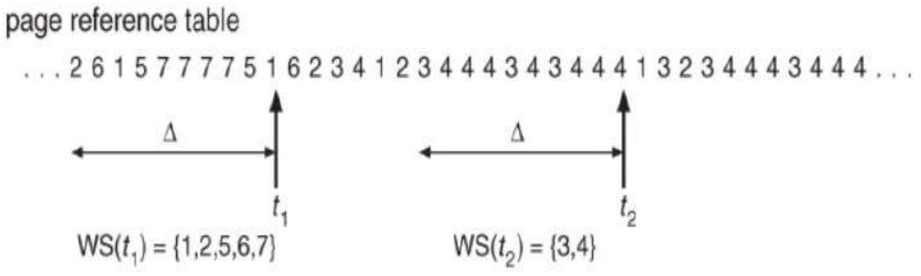


Figure 5.11 - Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand, D, is the sum of the sizes of the working sets for all processes. If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.
- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page. An approximation can be made using reference bits and a timer that goes off after a set interval of memory references:
  - For example, suppose that we set the timer to go off after every 5000 references (by any process), and we can store two additional historical reference bits in addition to the current reference bit.
  - Every time the timer goes off, the current reference bit is copied to one of the two historical bits, and then cleared.
  - If any of the three bits is set, then that page was referenced within the last 15,000 references, and is considered to be in that processes reference set.
- Finer resolution can be achieved with more historical bits and a more frequent timer, at the expense of greater overhead.

I/O Hardware

4.1 I/O devices

One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.

An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories −

- **Block devices** − A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
- **Character devices** − A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc

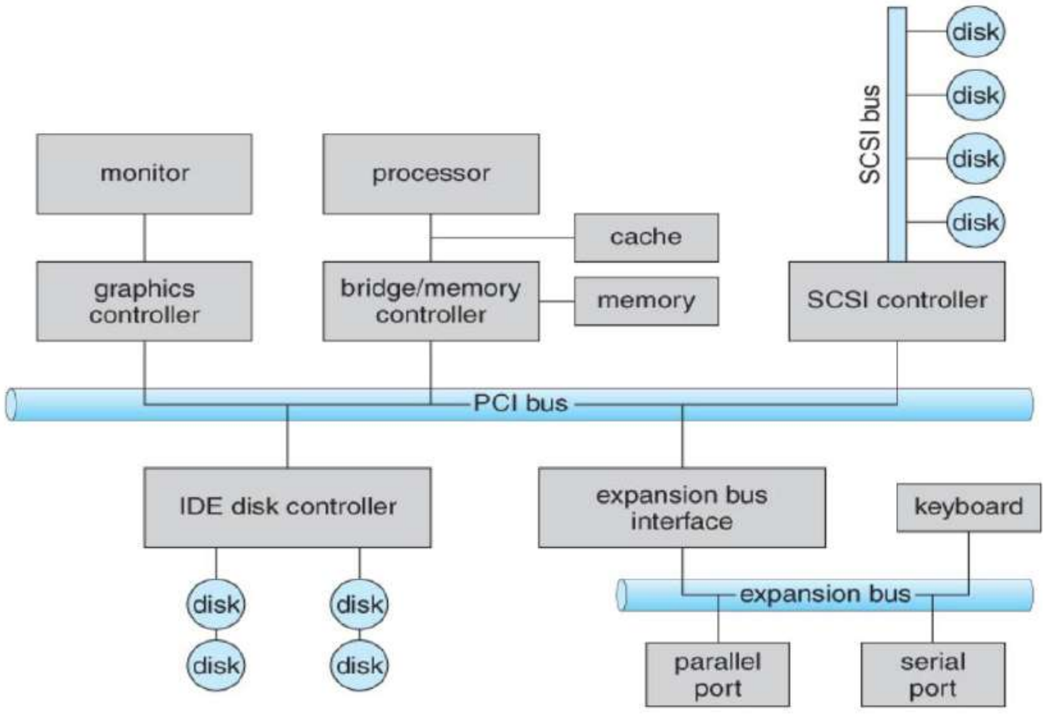


Figure - A typical PC bus structure.

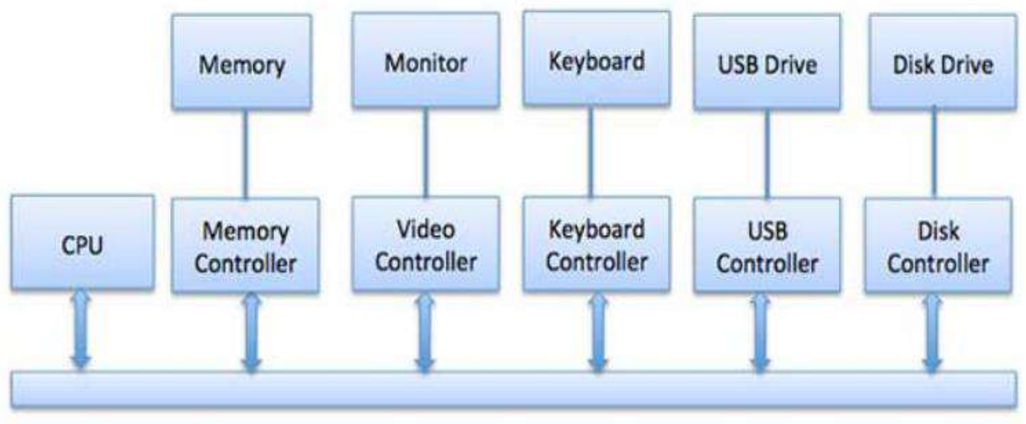
4.2 Device Controllers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

The Device Controller works like an interface between a device and a device driver. I/O units (Keyboard, mouse, printer, etc.) typically consist of a mechanical component and an electronic component where electronic component is called the device controller.

There is always a device controller and a device driver for each device to communicate with the Operating Systems. A device controller may be able to handle multiple devices. As an interface its main task is to convert serial bit stream to block of bytes, perform error correction as necessary.

Any device connected to the computer is connected by a plug and socket, and the socket is connected to a device controller. Following is a model for connecting the CPU, memory, controllers, and I/O devices where CPU and device controllers all use a common bus for communication.



Synchronous vs asynchronous I/O

- **Synchronous I/O** − In this scheme CPU execution waits while I/O proceeds
- **Asynchronous I/O** − I/O proceeds concurrently with CPU execution

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

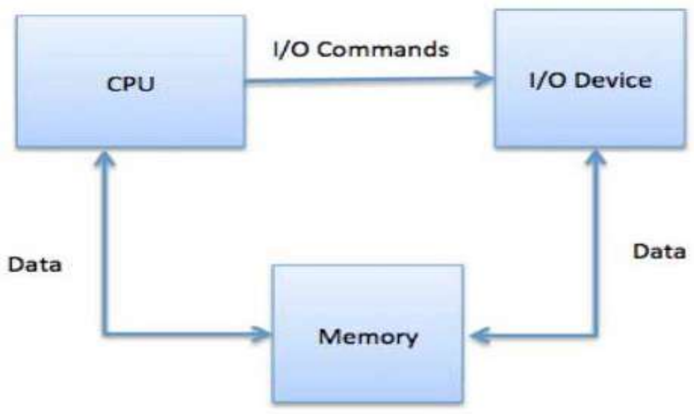
- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Special Instruction I/O

This uses CPU instructions that are specifically made for controlling I/O devices. These instructions typically allow data to be sent to an I/O device or read from an I/O device.

Memory-mapped I/O

When using memory-mapped I/O, the same address space is shared by memory and I/O devices. The device is connected directly to certain main memory locations so that I/O device can transfer block of data to/from memory without going through CPU.



While using memory mapped IO, OS allocates buffer in memory and informs I/O device to use that buffer to send data to the CPU. I/O device operates asynchronously with CPU, interrupts CPU when finished.

The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. Memory mapped IO is used for most high-speed I/O devices like disks, communication interfaces.

4.3 Direct Memory Access (DMA)

Direct Memory Access (DMA) transfers the block of data between the *memory* and *peripheral devices* of the system, without the participation of the processor. The unit that controls the activity of accessing memory directly is called a DMA controller. The processor relinquishes the system bus for a few clock cycles. So, the DMA controller can accomplish the task of data transfer via the system bus.

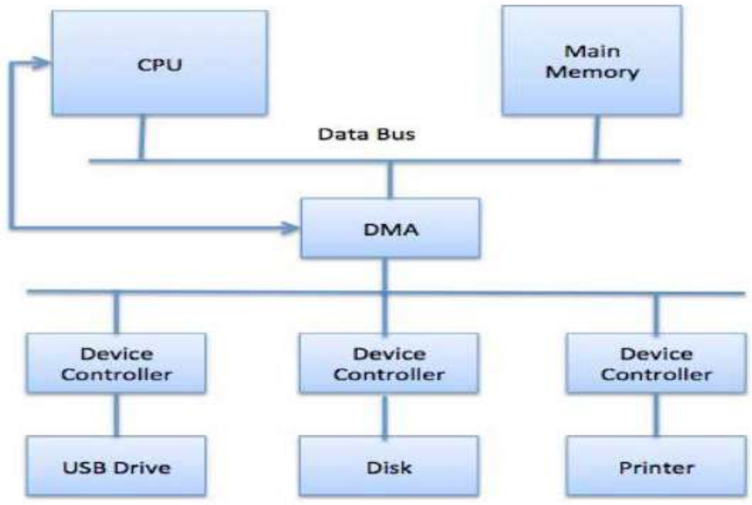
What is DMA and Why it is used?

Direct memory access (DMA) is a mode of data transfer between the memory and I/O devices. This happens without the involvement of the processor. We have two other methods of data transfer, programmed I/O and Interrupt driven I/O. Let’s revise each and get acknowledge with their drawbacks.

In programmed I/O, the processor keeps on scanning whether any device is ready for data transfer. If an I/O device is ready, the processor fully dedicates itself in transferring the data between I/O and memory. It transfers data at a high rate, but it can’t get involved in any other activity during data transfer. This is the major drawback of programmed I/O.

In Interrupt driven I/O, whenever the device is ready for data transfer, then it raises an interrupt to processor. Processor completes executing its ongoing instruction and saves its current state. It then switches to data transfer which causes a delay. Here, the processor doesn’t keep scanning for peripherals ready for data transfer. But, it is fully involved in the data transfer process. So, it is also not an effective way of data transfer.

The above two modes of data transfer are not useful for transferring a large block of data. But, the DMA controller completes this task at a faster rate and is also effective for transfer of large data block.



The DMA controller transfers the data in three modes:

1. **Burst Mode:** Here, once the DMA controller gains the charge of the system bus, then it releases the system bus only after completion of data transfer. Till then the CPU has to wait for the system buses.

2. **Cycle Stealing** Mode: In this mode, the DMA controller forces the CPU to stop its operation and relinquish the control over the bus for a short term to DMA controller. After the transfer of every byte, the DMA controller releases the bus and then again requests for the system bus. In this way, the DMA controller steals the clock cycle for transferring every byte.

3. **Transparent Mode:** Here, the DMA controller takes the charge of system bus only if the processor does not require the system bus.

Direct Memory Access Advantages and Disadvantages

Advantages:

1. Transferring the data without the involvement of the processor will **speed up** the read-write task.

2. DMA **reduces the clock cycle** requires to read or write a block of data.

3. Implementing DMA also **reduces the overhead** of the processor.

Disadvantages

1. As it is a hardware unit, it would **cost** to implement a DMA controller in the system.

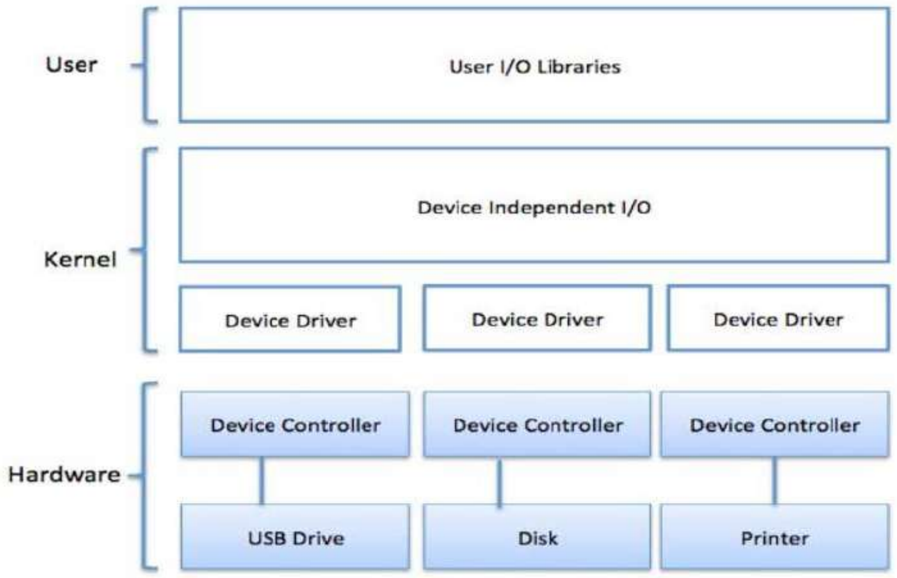
2. Cache **coherence** problem can occur while using DMA controller.

4.4 I/O Software

I/O software is often organized in the following layers −

- **User Level Libraries** − This provides simple interface to the user program to perform input and output. For example, **stdio** is a library provided by C and C++ programming languages.
- **Kernel Level Modules** − This provides device driver to interact with the device controller and device independent I/O modules used by the device drivers.
- **Hardware** − This layer includes actual hardware and hardware controller which interact with the device drivers and makes hardware alive.

A key concept in the design of I/O software is that it should be device independent where it should be possible to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.



4.5 Device Drivers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices. Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes. Device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs −

- To accept request from the device independent software above to it.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully

How a device driver handles a request is as follows: Suppose a request comes to read a block N. If the driver is idle at the time a request arrives, it starts carrying out the request immediately. Otherwise, if the driver is already busy with some other request, it places the new request in the queue of pending requests.

4.6 Interrupts Handlers

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an ***interrupt-request line*** that is sensed after every instruction.
    - A device's controller ***raises*** an interrupt by asserting a signal on the interrupt request line.
    - The CPU then performs a state save, and transfers control to the ***interrupt handler*** routine at a fixed address in memory.
    - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a ***return from interrupt*** instruction to return control to the CPU.

- Figure below illustrates the interrupt-driven I/O procedure:

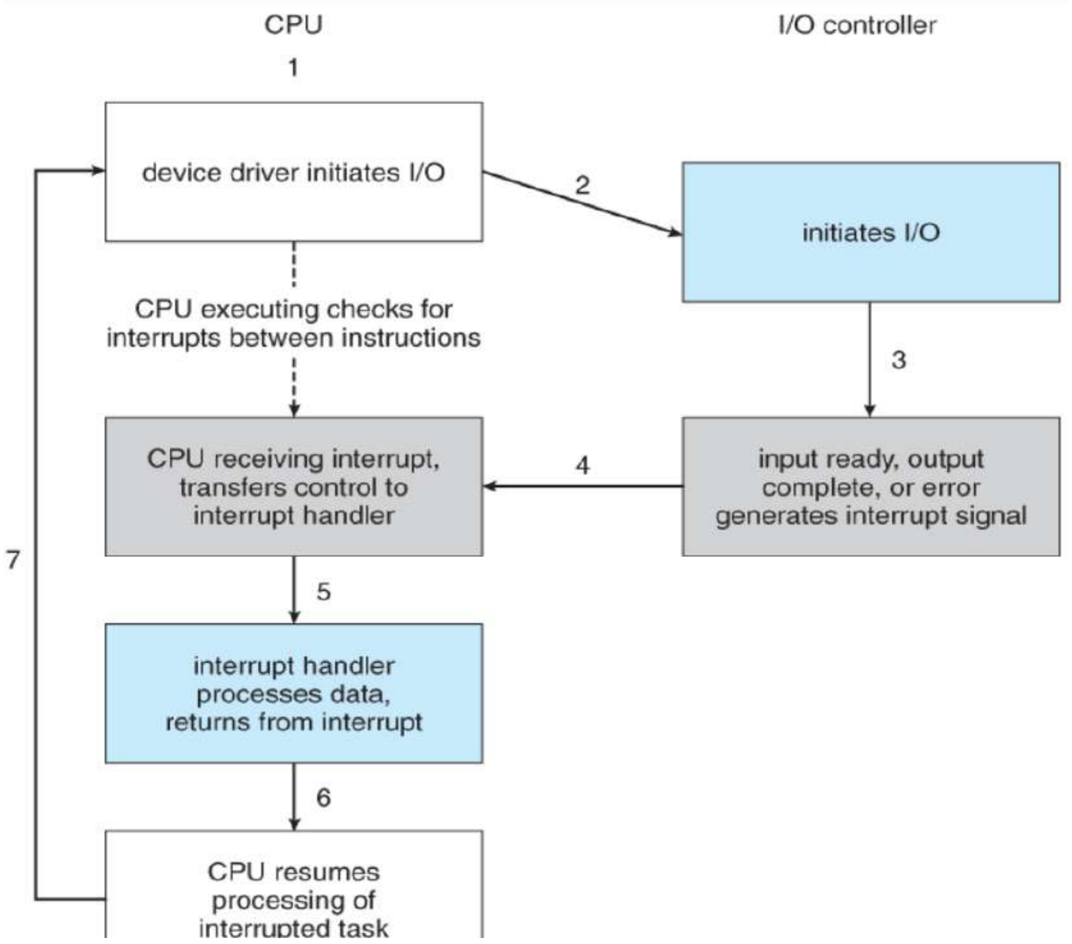

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
7

Figure - Interrupt-driven I/O cycle.

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:

    1. The need to defer interrupt handling during critical processing,
    2. The need to determine which interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
    3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

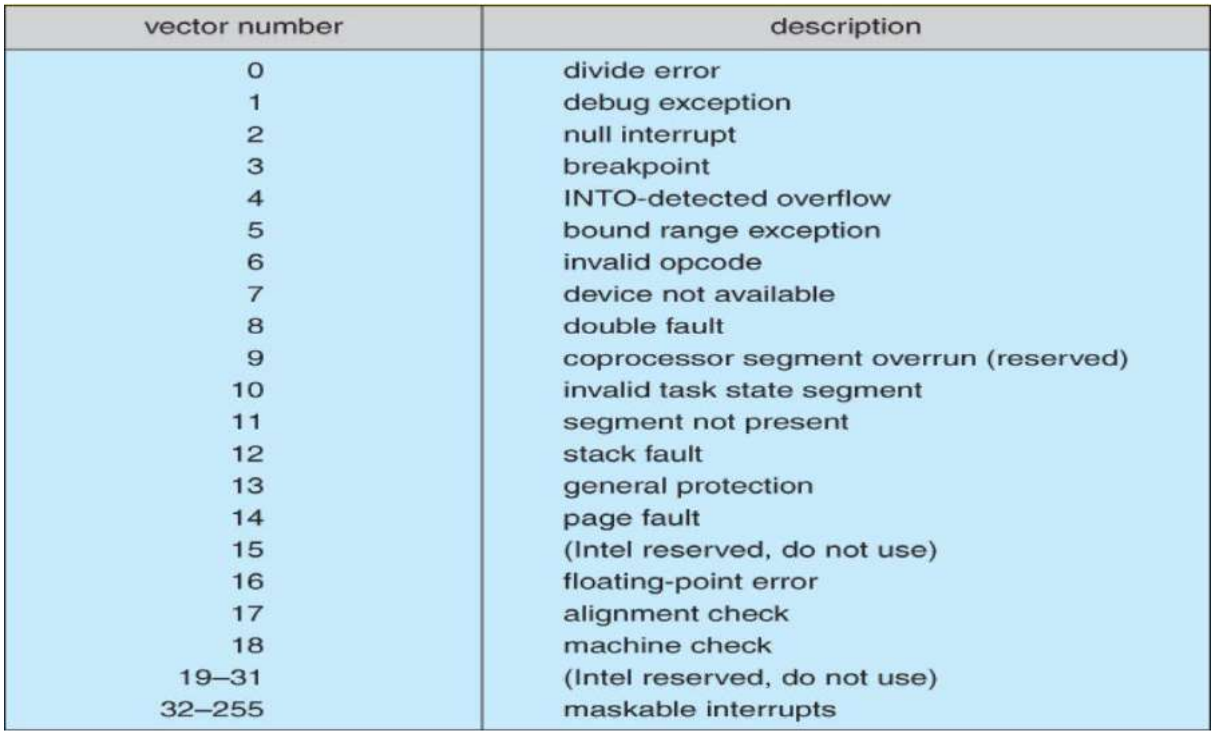
- These issues are handled in modern computer architectures with interrupt-controller hardware.

1. Most CPUs now have two interrupt-request lines: One that is ***non-maskable*** for critical error conditions and one that is ***maskable,*** that the CPU can temporarily ignore during critical processing.

2. The interrupt mechanism accepts an address, which is usually one of a small set of numbers for an offset into a table called the interrupt vector. This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.

3. Figure below shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.

4. Modern interrupt hardware also supports interrupt priority levels, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure - Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.

- During operation, devices signal errors or the completion of commands via interrupts.

- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
    - The scheduler sets a hardware timer before transferring control over to a user process.
    - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
    - The scheduler does a state-restore of a different process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the **paging s**ystem for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue,
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.

Difference between Maskable and Non Maskable Interrupt

1. Maskable Interrupt :

An Interrupt that can be disabled or ignored by the instructions of CPU are called as Maskable Interrupt.The interrupts are either edge-triggered or level-triggered or level-triggered.

Eg:

RST6.5,RST7.5,RST5.5 of 8085

2. Non-Maskable Interrupt :

An interrupt that cannot be disabled or ignored by the instructions of CPU are called as Non-Maskable Interrupt.A Non-maskable interrupt is often used when response time is critical or when an interrupt should never be disable during normal system operation. Such uses include reporting non-recoverable hardware errors, system debugging and profiling and handling of species cases like system resets.

Eg: Trap of 8085 **.**

Maskable Interrupt	**Non Maskable Interrupt**
Maskable interrupt is a hardware Interrupt that can be disabled or ignored by the instructions of CPU.	A non-maskable interrupt is a hardware interrupt that cannot be disabled or ignored by the instructions of CPU.
When maskable interrupt occur, it can be handled after executing the current instruction.	When non-maskable interrupts occur, the current instructions and status are stored in stack for the CPU to handle the interrupt.
Maskable interrupts help to handle lower priority tasks.	Non-maskable interrupt help to handle higher priority tasks such as watchdog timer.
Maskable interrupts used to interface with peripheral device.	Non maskable interrupt used for emergency purpose e.g power failure, smoke detector etc .
In maskable interrupts, response time is high.	In non maskable interrupts, response time is low.
It may be vectored or non-vectored.	All are vectored interrupts.
Operation can be masked or made pending.	Operation Cannot be masked or made pending.
RST6.5, RST7.5, and RST5.5 of 8085 are some common examples of maskable Interrupts.	Trap of 8085 microprocessor is an example for non-maskable interrupt.

4.7 Device-Independent I/O Software

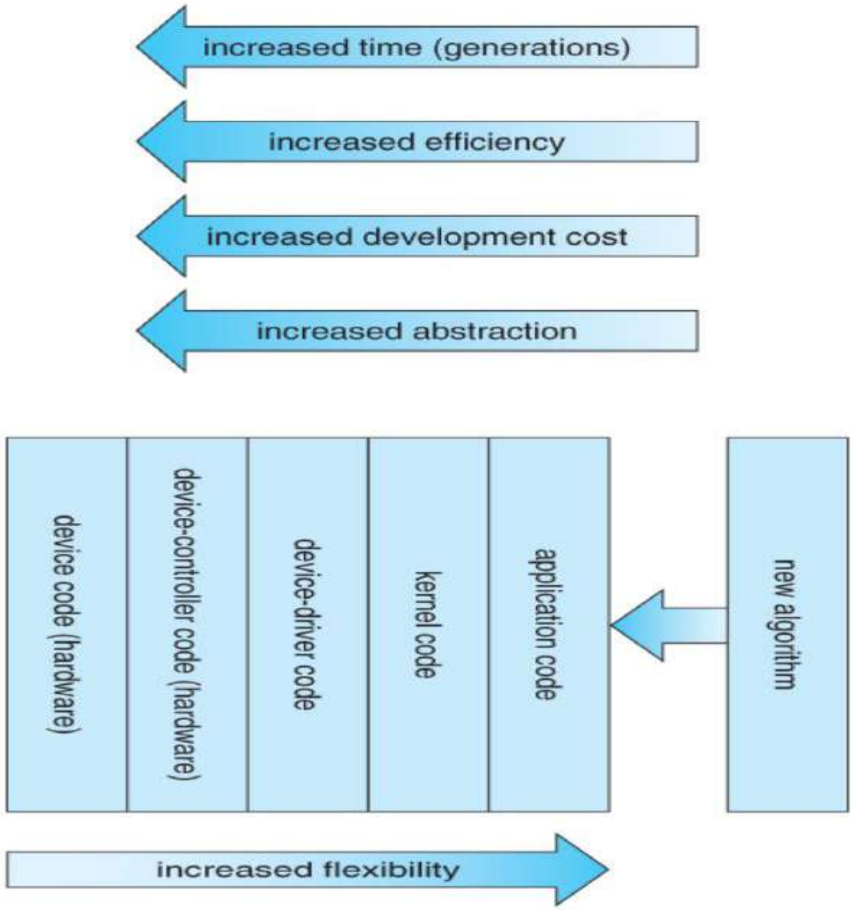
The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices. Following is a list of functions of device-independent I/O Software-

- Uniform interfacing for device drivers
- Device naming - Mnemonic names mapped to Major and Minor device numbers
- Device protection
- Providing a device-independent block size
- Buffering because data coming off a device cannot be stored in final destination.
- Storage allocation on block devices
- Allocation and releasing dedicated devices
- Error Reporting

4.8 I/O Performance

- The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few.)
- Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.
- Network traffic can also put a heavy load on the system. Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure 13.15. (And the fact that a similar set of events must happen in reverse to echo back the character that was typed.) Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.
- **Several principles can be employed to increase the overall efficiency of I/O processing:**
    1. Reduce the number of context switches.
    2. Reduce the number of times data must be copied.
    3. Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
    4. Increase concurrency using DMA.
    5. Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
    6. Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.

- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure 13.16. Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify. Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.



FILE MANAGEMENT

5.1File Concept

File: A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From user's perspective a file is the smallest allotment of logical secondary storage.

5.1.1 File Attributes

 Different OSes keep track of different file attributes, including

- **Name** - Some systems give special significance to names, and particularly extensions (.exe, .txt, etc.), and some do not. Some extensions may be of significance to the OS (.exe), and others only to certain applications (.jpg)
- **Identifier** (e.g. inode number)
- **Type** - Text, executable, other binary, etc.
- **Location** - on the hard drive.
- **Size**
- **Protection**
- **Time & Date**
- **User ID**

5.1.2 File Operations

· The file ADT supports many common operations:

o Creating a file

o Writing a file

o Reading a file

o Repositioning within a file

o Deleting a file

o Truncating a file.

· Most OSes require that files be opened before access and closed after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an open file table, containing for example:

o **File pointer** - records the current position in the file, for the next read or write access.

o **File-open count** - How many times has the current file been opened (simultaneously by different processes) and not yet closed? When this counter reaches zero the file can be removed from the table.

o **Disk location** of the file.

o **Access rights**

· Some systems provide support for **file locking**.

o A **shared lock** is for reading only.

o A **exclusive lock** is for writing as well as reading.

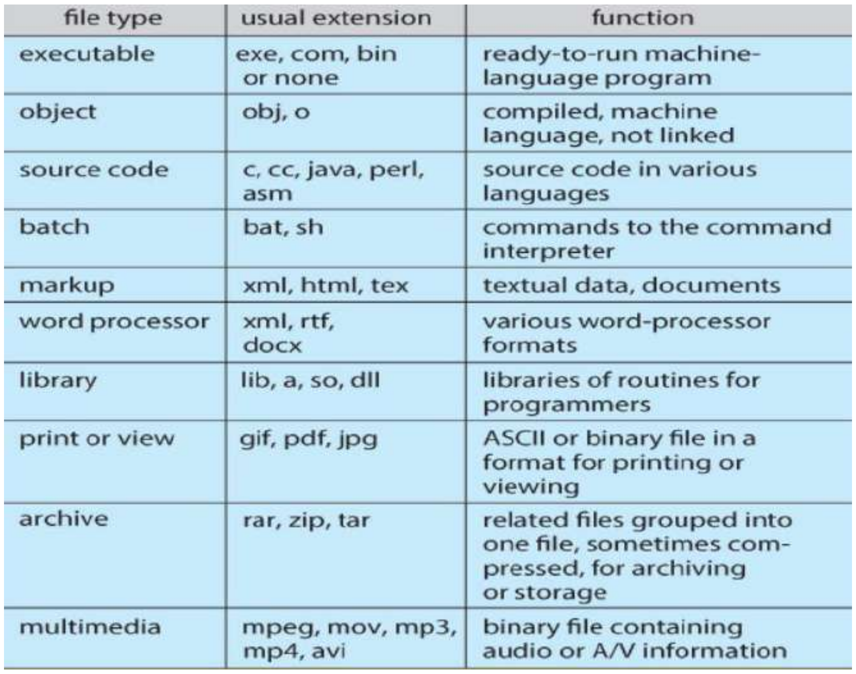
o An **advisory lock** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored.)

o A **mandatory lock** is enforced. (A truly locked door.)

o UNIX used advisory locks, and Windows uses mandatory locks.

5.1.3 File Types

Windows (and some other systems) use special file extensions to indicate the type of each file:



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

Figure - Common file types.

· Macintosh stores a creator attribute for each file, according to the program that first created it with the create( ) system call.
· UNIX stores magic numbers at the beginning of certain files. (Experiment with the "file" command, especially in directories such as /bin and /dev)

5.1.4 File Structure

· Some files contain an internal structure, which may or may not be known to the OS.

· For the OS to support particular file formats increases the size and complexity of the OS.

· UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)

· Macintosh files have two forks - a resource fork, and a data fork. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

5.1.5 Internal File Structure

· Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple there of. (Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer.)

· Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.

· The number of logical units which fit into one physical block determines its packing, and has an impact on the amount of internal fragmentation (wasted space) that occurs.

· As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

5.2 Access Methods

5.2.1Sequential Access

· A sequential access file emulates magnetic tape operation, and generally supports a few operations:

o **read next** - read a record and advance the tape to the next position.

o **write next** - write a record and advance the tape to the next position.

o **rewind**

o **skip n records** - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

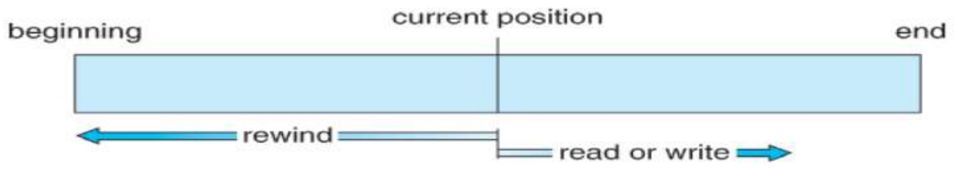


Fig: Sequential File Access

5.2.2 Direct Access

· Jump to any record and read that record. Operations supported include:

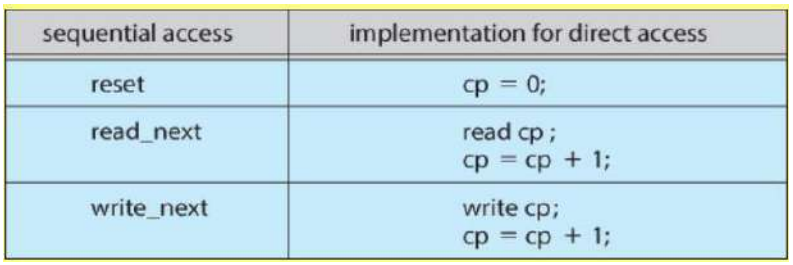
o read n - read record number n. (Note an argument is now required.)

o write n - write record number n. (Note an argument is now required.)

o jump to record n - could be 0 or the end of file.

o Query current record - used to return back to this record later.

o Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.



sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;

write_next	write cp;
	cp = cp + 1;

Fig: Simulation of sequential access on a direct-access file

5.2.3 Other Access Methods

An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

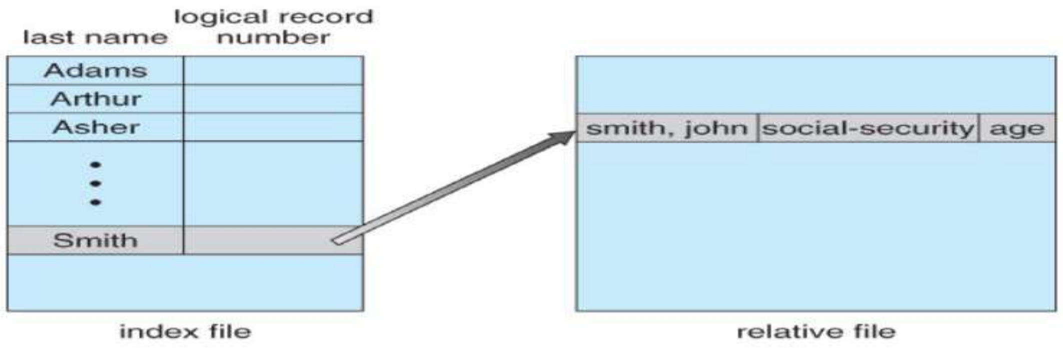


Fig: Example of index and relative files.

5.3 Directory Structure

5.3.1Storage Structure

· A disk can be used in its entirety for a file system.

· Alternatively a physical disk can be broken up into multiple partitions, slices, or minidisks, each of which becomes a virtual disk and can have its own file system. (or be used for raw storage, swap space, etc.)

· Or, multiple physical disks can be combined into one volume, i.e. a larger virtual disk, with its own file system spanning the physical disks.

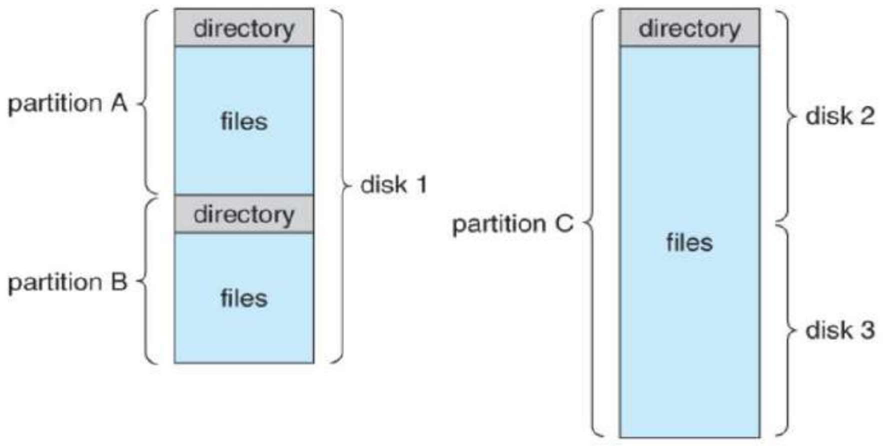


Fig: A typical file-system organization

5.3.2 Directory Overview

· Directory operations to be supported include:

o Search for a file

o Create a file - add to the directory

o Delete a file - erase from the directory

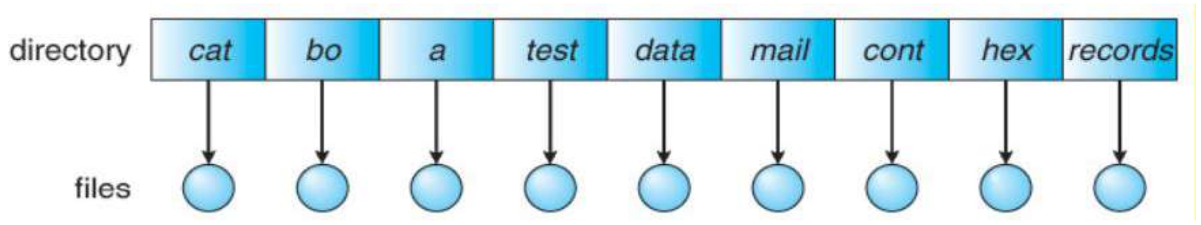
o List a directory - possibly ordered in different ways.

o Rename a file - may change sorting order

o Traverse the file system

5.3.3 Single-Level Directory

- Simple to implement, but each file must have a unique name.



Draw Backs:

· Naming Problem

· Grouping Problem

5.3.4 Two-Level Directory

· Each user gets their own directory space.

· File names only need to be unique within a given user's directory.

· A master file directory is used to keep track of each user’s directory, and must be maintained when users are added to or removed from the system.

· A separate directory is generally needed for system (executable) files.

· Systems may or may not allow users to access other directories besides their own

o If access to other directories is allowed, then provision must be made to specify the directory being accessed.

o If access is denied, then special consideration must be made for users to run programs located in system directories. A search path is the list of directories in which to search for executable programs, and can be set uniquely for each user.

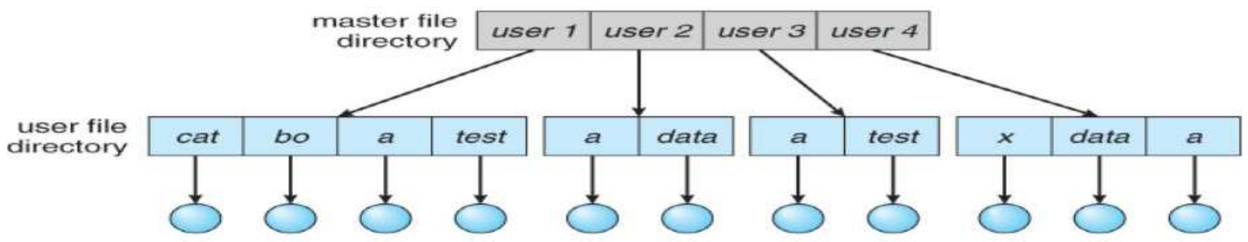


Figure - Two-level directory structure.

5.3.5 Tree-Structured Directories

· An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar.
· Each user / process has the concept of a current directory from which all (relative) searches take place.
· Files may be accessed using either absolute pathnames (relative to the root of the tree) or relative pathnames (relative to the current directory.)
· Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.
· One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

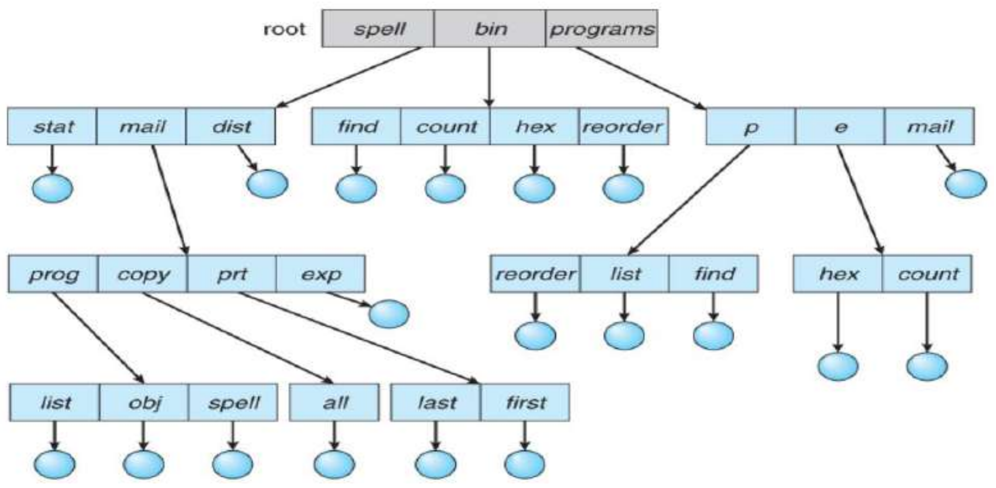


Fig: Tree Structured Directory Structure

5.3.6 Acyclic-Graph Directories

· When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an acyclic-graph structure. (Note the directed arcs from parent to child.)

· UNIX provides two types of links for implementing the acyclic-graph structure. (See "man ln" for more details.)

o A **hard link** (usually just called a link) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.

o A **symbolic link** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

· Windows only supports symbolic links, termed shortcuts.

· Hard links require a reference count, or link count for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

· For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:

o One option is to find all the symbolic links and adjust them also.

o Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.

o What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

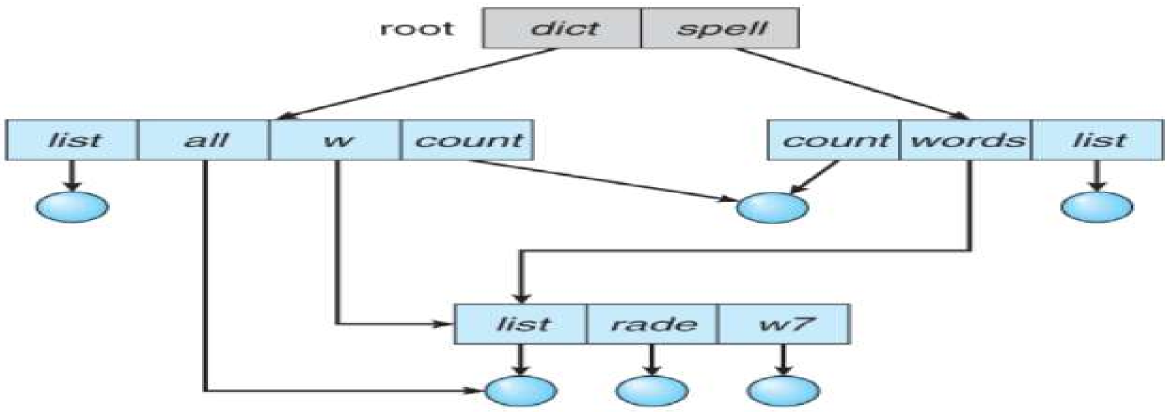


Figure - Acyclic-graph directory structure**.**

5.3.7 General Graph Directory

· If cycles are allowed in the graphs, then several problems can arise:

o Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories.)

o Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safelydeleted.)

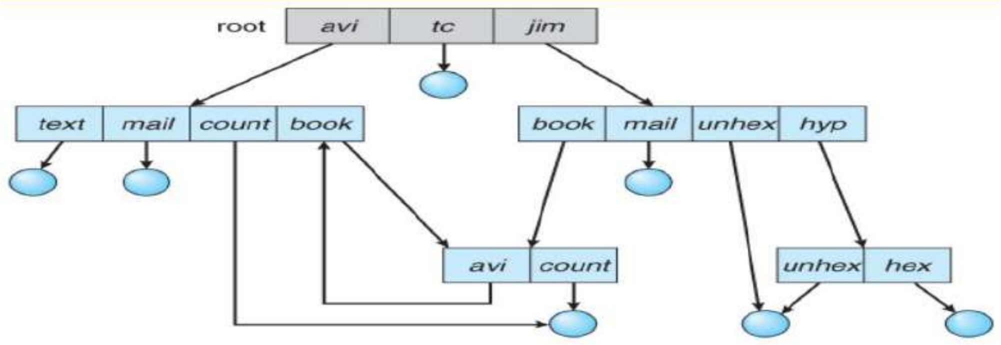


Figure - General graph directory.

5.4 File-System Mounting

· The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.

· The mount command is given a file system to mount and a mount point (directory) on which to attach it.

· Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

· Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.

· File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted.

· File systems may be mounted read-only, or have other restrictions imposed.

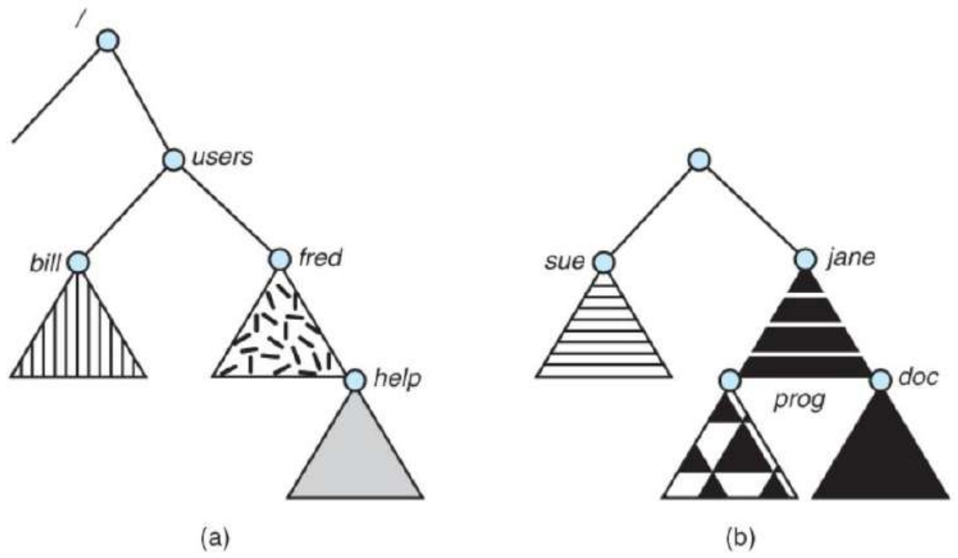


Figure - File system. (a) Existing system. (b) Unmounted volume.

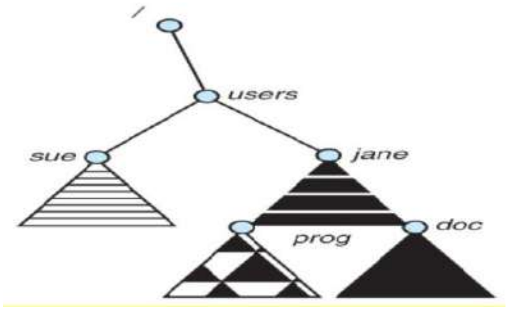


Figure - Mount point.

· The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.

· Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.

· More recent Windows systems allow file systems to be mounted to any directory in the file system, much like UNIX.

5.5 File Systems

File system is the part of the operating system which is responsible for file management. It provides a mechanism to store the data and access to the file contents including data and programs. Some Operating systems treats everything as a file for example Ubuntu.

The File system takes care of the following issues

File Structure

We have seen various data structures in which the file can be stored. The task of the file system is to maintain an optimal file structure.

Recovering Free space

Whenever a file gets deleted from the hard disk, there is a free space created in the disk. There can be many such spaces which need to be recovered in order to reallocate them to other files.

disk space assignment to the files

The major concern about the file is deciding where to store the files on the hard disk. There are various disks scheduling algorithm which will be covered later in this tutorial.

tracking data location

A File may or may not be stored within only one block. It can be stored in the non contiguous blocks on the disk. We need to keep track of all the blocks on which the part of the files reside.

5.6 File-System Structure

File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way. A file System must be able to store the file, locate the file and retrieve the file.

Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

The image shown below, elaborates how the file system is divided in different layers, and also the functionality of each layer.

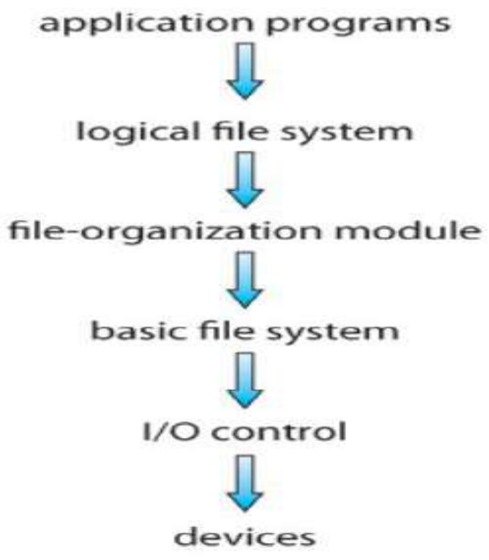


Figure - Layered file system.

- When an application program asks for a file, the first request is directed to the logical file system. The logical file system contains the Meta data of the file and directory structure. If the application program doesn't have the required permissions of the file then this layer will throw an error. Logical file systems also verify the path to the file.

- Generally, files are divided into various logical blocks. Files are to be stored in the hard disk and to be retrieved from the hard disk. Hard disk is divided into various tracks and sectors. Therefore, in order to store and retrieve the files, the logical blocks need to be mapped to physical blocks. This mapping is done by File organization module. It is also responsible for free space management.
- Once File organization module decided which physical block the application program needs, it passes this information to basic file system. The basic file system is responsible for issuing the commands to I/O control in order to fetch those blocks.
- I/O controls contain the codes by using which it can access hard disk. These codes are known as device drivers. I/O controls are also responsible for handling interrupts.

5.7 Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.
- There are number of algorithms by using which, the directories can be implemented. However, the selection of an appropriate directory implementation algorithm may significantly affect the performance of the system.
- The directory implementation algorithms are classified according to the data structure they are using. There are mainly two algorithms which are used in these days.

5.7.1 Linear List

· A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.

· Finding a file (or verifying one does not already exist upon creation) requires a linear search.

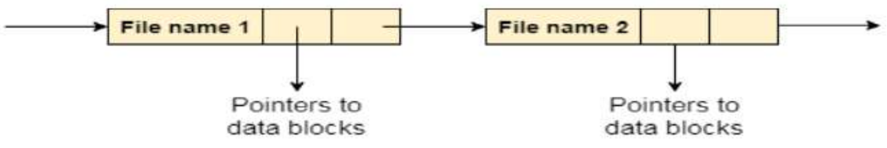
· Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.

· Sorting the list makes searches faster, at the expense of more complex insertions and deletions.

· A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.

· More complex data structures, such as B-trees, could also be considered.

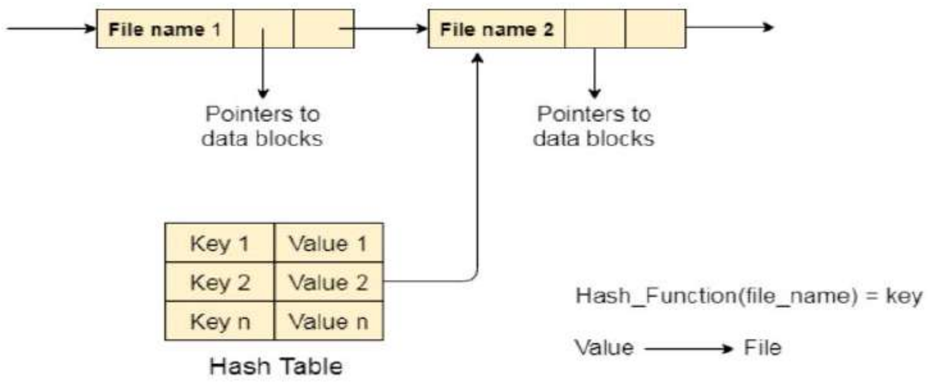
- When a new file is created, then the entire list is checked whether the new file name is matching to a existing file name or not. In case, it doesn't exist, the file can be created at the beginning or at the end. Therefore, searching for a unique name is a big concern because traversing the whole list takes time.
- The list needs to be traversed in case of every operation (creation, deletion, updating, etc) on the files therefore the systems become inefficient.



Linear List

5.7.1 Hash Table

- To overcome the drawbacks of singly linked list implementation of directories, there is an alternative approach that is hash table. This approach suggests to use hash table along with the linked lists.
- A key-value pair for each file in the directory gets generated and stored in the hash table. The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory.
- Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.



5.8 Allocation Methods

There are three major methods of storing files on disks:

· contiguous,

· linked, and

· indexed.

5.8.1 Contiguous Allocation

· Contiguous Allocation requires that all blocks of a file be kept together contiguously.

· Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.

· Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory (first fit, best fit, fragmentation problems, etc.) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.

· (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)

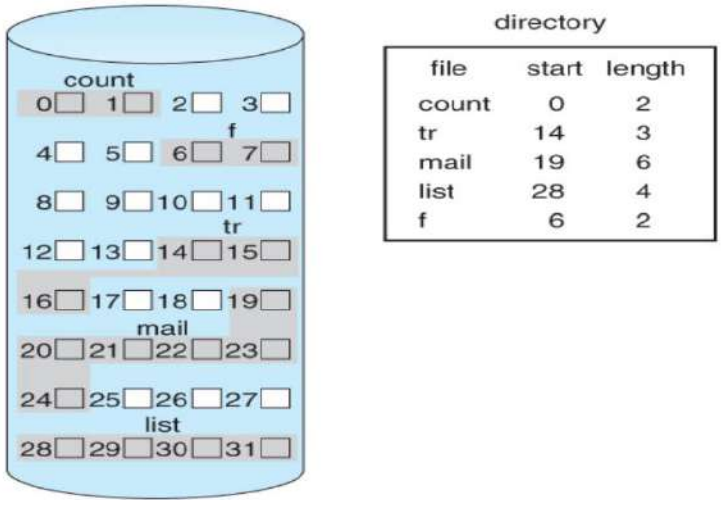
· Problems can arise when files grow, or if the exact size of a file is unknown at creation time:

Over-estimation of the file's final size increases external fragmentation and wastes disk space.

o Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.

o If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.

· A variation is to allocate file space in large contiguous chunks, called extents. When a file outgrows its original extent, then an additional one is allocated. (For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary.) The high-performance files system Veritas uses extents to optimize performance.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Figure - Contiguous allocation of disk space

Advantages

1. It is simple to implement.
2. We will get Excellent read performance.
3. Supports Random Access into files.

Disadvantages

1. The disk will become fragmented.
2. It may be difficult to have a file grow.

5.8.2 Linked Allocation

· Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. (E.g. a block may be 508 bytes instead of 512.)

· Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.

· Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.

· Allocating clusters of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.

· Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

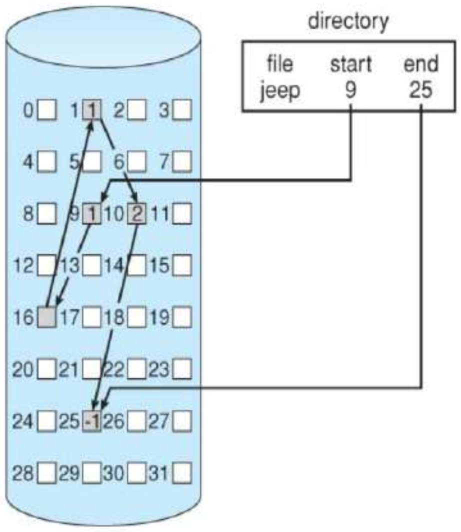


Figure - Linked allocation of disk space.

· The File Allocation Table, FAT, used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.

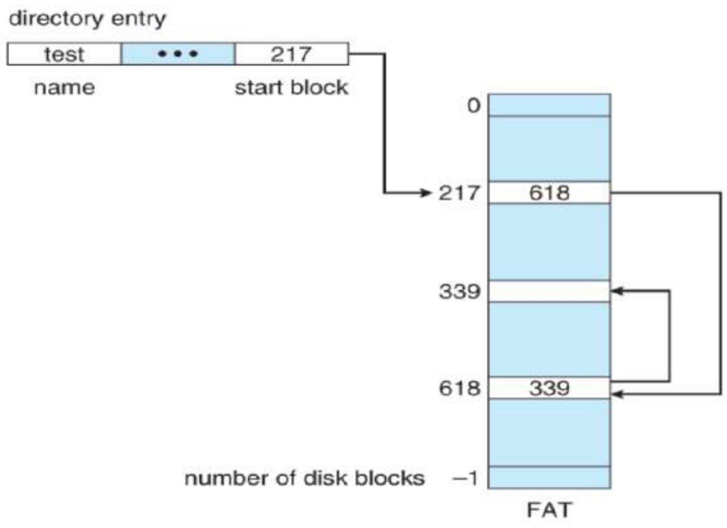


Figure- File-allocation table.

Advantages

1. There is no external fragmentation with linked allocation.
2. Any free block can be utilized in order to satisfy the file block requests.
3. File can continue to grow as long as the free blocks are available.

4. Directory entry will only contain the starting block address.

Disadvantages

1. Random Access is not provided.
2. Pointers require some space in the disk blocks.
3. Any of the pointers in the linked list must not be broken otherwise the file will get corrupted.
4. Need to traverse each block.

5.8.3 Indexed Allocation

· Indexed Allocation combines all of the indexes for accessing each file into a common block (for that file), as opposed to spreading them all over the disk or storing them in a FAT table.

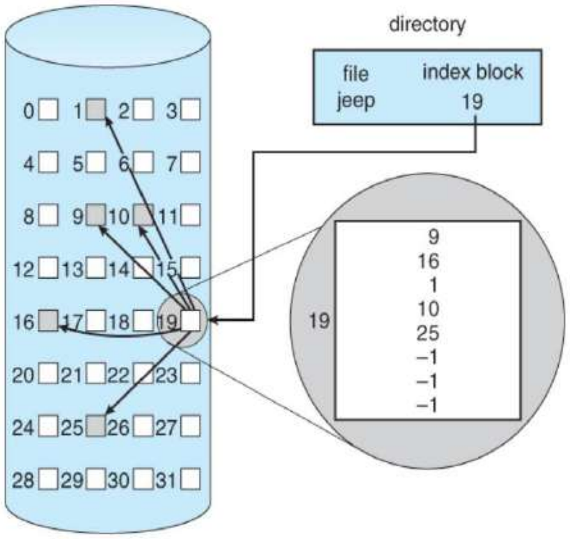


Figure - Indexed allocation of disk space.

· Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

o **Linked Scheme** - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.

- o **Multi-Level Index** - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
- o **Combined Scheme** - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below.) The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)

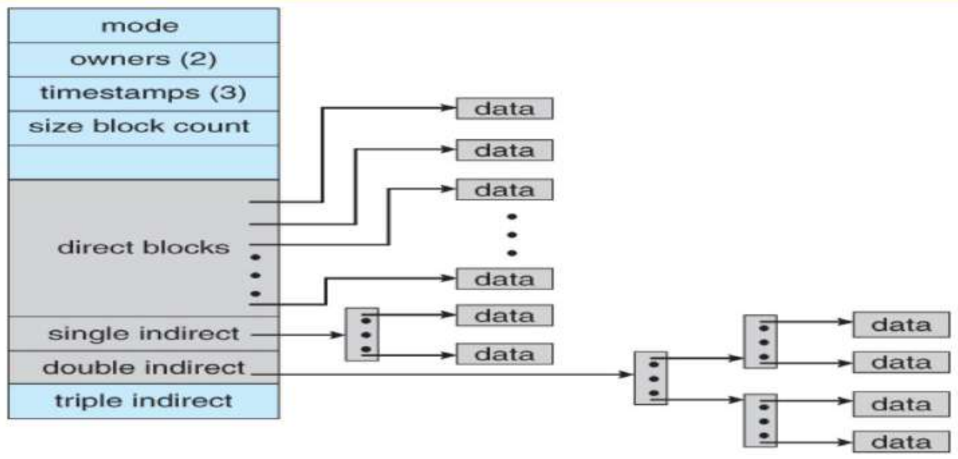


Figure - The UNIX inode.

Advantages

1. Supports direct access
2. A bad data block causes the lost of only that block.

Disadvantages

1. A bad index block could cause the lost of entire file.
2. Size of a file depends upon the number of pointers, a index block can hold.
3. Having an index block for a small file is totally wastage.
4. More pointer overhead

5.8.4 Performance

· The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.

· Some systems support more than one allocation method, which may require specifying how the file is to be used (sequential or random access) at the time it is allocated. Such systems also provide conversion utilities.

· Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

· And of course some systems adjust their allocation schemes (e.g. block sizes) to best match the characteristics of the hardware for optimum performance.

5.9 Free-Space Management

Another important aspect of disk management is keeping track of and allocating free space.

5.9.1 Bit Vector

· One simple approach is to use a bit vector, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

· Fast algorithms exist for quickly finding contiguous blocks of a given size

· The down side is that a 40GB disk requires over 5MB just to store the bitmap.

5.9.2 Linked List

· A linked list can also be used to keep track of all free blocks.

· Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.

· The FAT table keeps track of the free list as just one more linked list on the table.

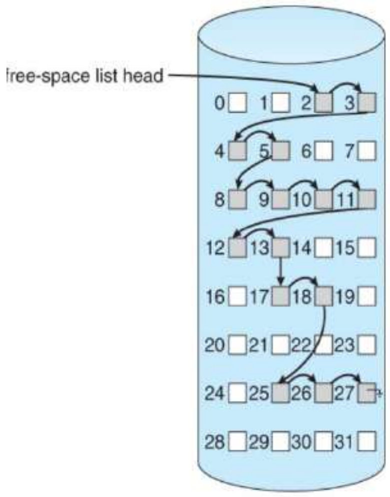


Figure - Linked free-space list on disk.

5.9.3 Grouping

A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

5.9.4 Counting

When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. (Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered.)

5.10 Efficiency and Performance

5.10.1 Efficiency

· UNIX pre-allocates inodes, which occupies space even before any files are created.

· UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.

· Some systems use variable size clusters depending on the file size.

· The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.

· As technology advances, addressing schemes have had to grow as well.

o Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms!)

· Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

5.10.2 Performance

· Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads (reducing latency.) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.

· Some OSes cache disk blocks they expect to need again in a buffer cache.

· A page cache connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.

· Some systems (Solaris, Linux, Windows 2000, NT, XP) use page caching for both process pages and file data in a unified virtual memory.

· Figures below show the advantages of the unified buffer cache found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.

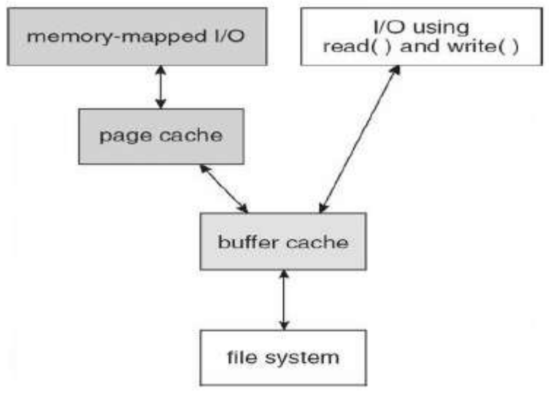


Figure : I/O without a unified buffer cache.

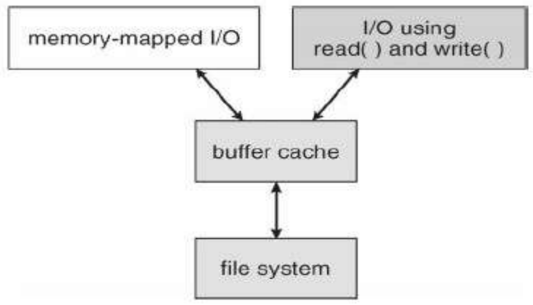


Figure 12.12 - I/O using a unified buffer cache.

· Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in priority paging giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.

· Another issue affecting performance is the question of whether to implement synchronous writes or asynchronous writes. Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are cached, allowing the disk subsystem to schedule writes in a more efficient order Metadata writes are often done synchronously. Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.

· The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these Types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, (if it is ever needed at all.) On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:

o **Free-behind** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need again for a long time.

o **Read-ahead** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller,
except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.

· The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times. Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.

Secondary-Storage Structure

6.1 Device management

The main functions of device management in the operating system

An operating system or the OS manages communication with the devices through their respective drivers. The operating system component provides a uniform interface to access devices of varied physical attributes. For device management in operating system:

- Keep tracks of all devices and the program which is responsible to perform this is called I/O controller.
- Monitoring the status of each device such as storage drivers, printers and other peripheral devices.
- Enforcing preset policies and taking a decision which process gets the device when and for how long.
- Allocates and Deallocates the device in an efficient way.De-allocating them at two levels: at the process level when I/O command has been executed and the device is temporarily released, and at the job level, when the job is finished and the device is permanently released.
- Optimizes the performance of individual devices.

Types of devices

The OS peripheral devices can be categorized into 3: Dedicated, Shared, and Virtual. The differences among them are the functions of the characteristics of the devices as well as how they are managed by the Device Manager.

Dedicated devices:-

Such type of devices in the device management in operating system are dedicated or assigned to only one job at a time until that job releases them. Devices like printers, tape drivers, plotters etc. demand such allocation scheme since it would be awkward if several users share them at the same point of time. The disadvantages of such kind of devices is the inefficiency resulting from the allocation of the device to a single user for the entire duration of job execution even though the device is not put to use 100% of the time.

Shared devices:-

These devices can be allocated to several processes. Disk-DASD can be shared among several processes at the same time by interleaving their requests. The interleaving is carefully controlled by the Device Manager and all issues must be resolved on the basis of predetermined policies.

Virtual Devices:-

These devices are the combination of the first two types and they are dedicated devices which are transformed into shared devices. For example, a printer converted into a shareable device via spooling program which re-routes all the print requests to a disk. A print job is not sent straight to the printer, instead, it goes to the disk(spool)until it is fully prepared with all the necessary sequences and formatting, then it goes to the printers. This technique can transform one printer into several virtual printers which leads to better performance and use.

Basics of I/O Devices

Three categories

- A **block device** stores information in fixed-size blocks, each one with its own address

  e.g., **disks**
- A **character device** delivers or accepts a stream of characters, and individual characters are not addressable e.g., **keyboards, printers**
- A **network device** transmit data packets

Device Controller

- Converts between serial bit stream and a block of bytes
- Performs error correction if necessary
- **Components:**

  Device registers to communicate with the CPU

  Data buffer that an OS can read or write

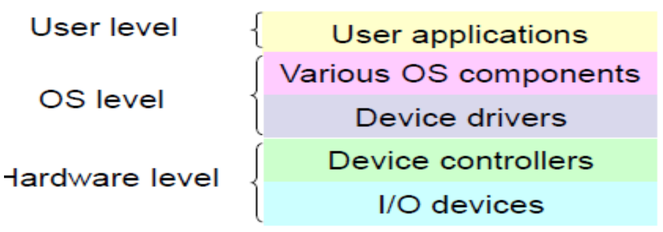
Device Driver

A device driver is a special kind of software program that controls a specific hardware device attached to a computer. Device drivers are essential for a computer to work properly. These programs may be compact, but they provide the all-important means for a computer to interact with hardware, for everything from mouse, keyboard and display to working with networks, storage and graphics.

How Do Device Drivers Work?

Device drivers generally run at a high level of privilege within the operating system runtime environment. Some device drivers, in fact, may be linked directly to the operating system **kernel**, a portion of an OS such as Windows, Linux or Mac OS, that remains memory resident and handles execution for all other code, including device drivers. Device drivers relay requests for device access and actions from the operating system and its active applications to their respective hardware devices. They also deliver outputs or status/messages from the hardware devices to the operating system (and thence, to applications).

Device Driver Illustrated



Ways to Access a Device

- **Polling:** a CPU repeatedly checks the status of a device for exchanging data

  + Simple

  - Busy-waiting

- **Interrupt-driven I/Os**: A device controller notifies the corresponding device driver when the device is available

  + More efficient use of CPU cycles

  - Data copying and movements

  - Slow for character devices (i.e., one interrupt per keyboard input)

- **Direct memory access (DMA):** uses an additional controller to perform data movements

  + CPU is not involved in copying data

  - A process cannot access in-transit data

- **Double buffering:** uses two buffers. While one is being used, the other is being filled

  Analogy: pipelining

  Extensively used for graphics and animation

  So a viewer does not see the line-by-line scanning

Disk Characteristics

- **Disk platters**: coated with magnetic materials for recording
- **Disk arm:** moves a comb of disk heads

  Only one disk head is active for reading/writing

- Each disk platter is divided into concentric **tracks**
- A track is further divided into **Sectors.**

  A sector is the smallest unit of disk storage

- A **cylinder** consists of all tracks with a given disk arm position

  Cylinders are further divided into **zones**

- **Zone-bit recording:** zones near the edge of a disk store more information (higher bandwidth)

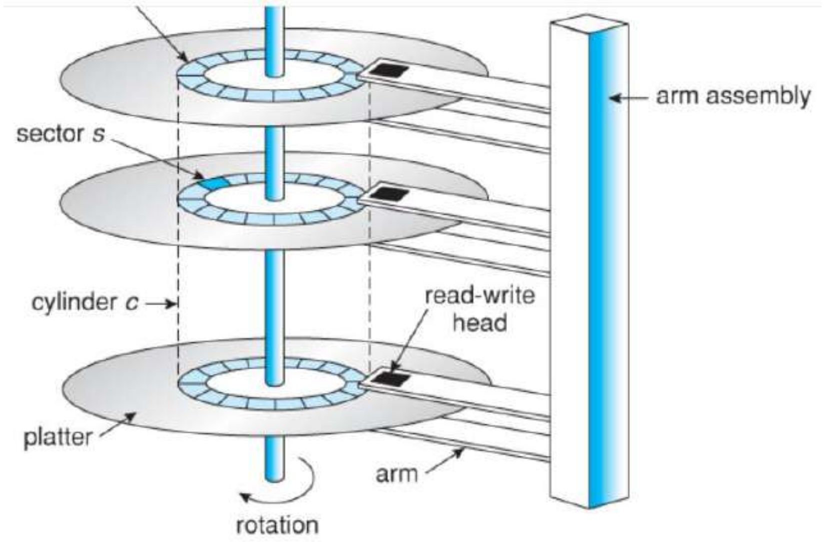


Figure - Moving-head disk mechanism.

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

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

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

• The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. (Some authors may also use the term transfer rate to refer to the overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate.)

• **Disk access time**

Seek time + rotational delay + transfer time

Disk Performance Metrics

• **Latency :** Seek time + rotational delay

• **Bandwidth:** Bytes transferred / disk access time

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

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

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

6.2 Disk Structure

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

1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

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

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

- With ***Constant Linear Velocity, CLV,*** the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
- With ***Constant Angular Velocity, CAV,*** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

6.3 Disk Scheduling

A process needs two type of time, CPU time and IO time. For I/O, it requests the Operating system to access the disk.

However, the operating system must be fare enough to satisfy each request and at the same time, operating system must maintain the efficiency and speed of process execution.

The technique that operating system uses to determine the request which is to be satisfied next is called disk scheduling.

Seek time

Seek time is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.

Rotational latency

It is the time taken by the desired sector to rotate itself to the position from where it can access the R/W heads.

Transfer Time

It is the time taken to transfer the data.

Disk Access Time

Disk access time is given as,

Disk Access Time = Rotational Latency + Seek Time + Transfer Time

Disk Response Time

It is the average of time spent by each request waiting for the IO operation.

Purpose of Disk Scheduling

The main purpose of disk scheduling algorithm is to select a disk request from the queue of IO requests and decide the schedule when this request will be processed.

Goal of Disk Scheduling Algorithm

- Fairness
- High throughout
- Minimal traveling head time

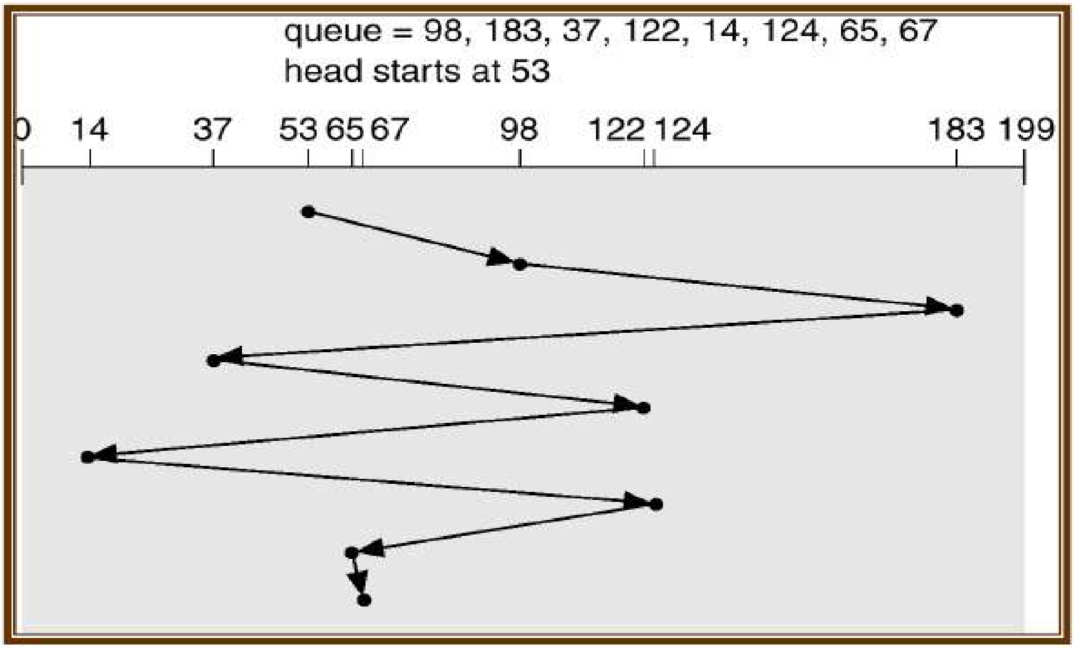
6.3.1 Disk Scheduling Algorithms

The list of various disks scheduling algorithm is given below. Each algorithm is carrying some advantages and disadvantages. The limitation of each algorithm leads to the evolution of a new algorithm.

- FCFS scheduling algorithm
- SSTF (shortest seek time first) algorithm
- SCAN scheduling
- C-SCAN scheduling
- LOOK Scheduling
- C-LOOK scheduling

FCFS: FIRST COME FIRST SERVES Scheduling

- First-come first served scheduling follow first in first out method
- It serves the request in the same order as they are received
- The average waiting time for FCFS is often quite long. It is non-preemptive
- In the below example the disk head is initially at cylinder 53, it will first move 53 to 98 then to 183,37,122,14,124,65 and finally to 67 for a total head movement of 640 cylinders.



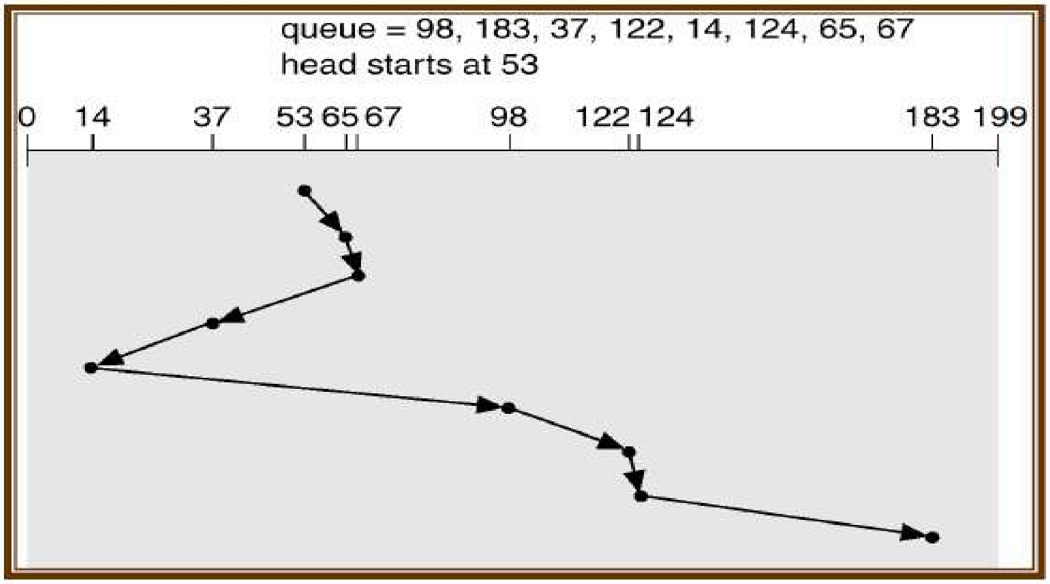
ex – 98-53 = 45

Total head movement = 45+85+146+85+108+110+59+2
= 640

- **Advantages :**
  - Simple ,fair to all request.
  - No starvation.
- **Disadvantages :**
  - Not efficient because the average seek time is high.

SSTF : SHORTEST SEEKTIME FIRST SCHEDULING

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders.



Total head movement = 12+2+30+23+84+24+2+59
= 236

- Advantages :
  - More efficient than FCFS.
- Disadvantages :
  - Starvation is possible for request involving longer seek-time.

SCAN SCHEDULING

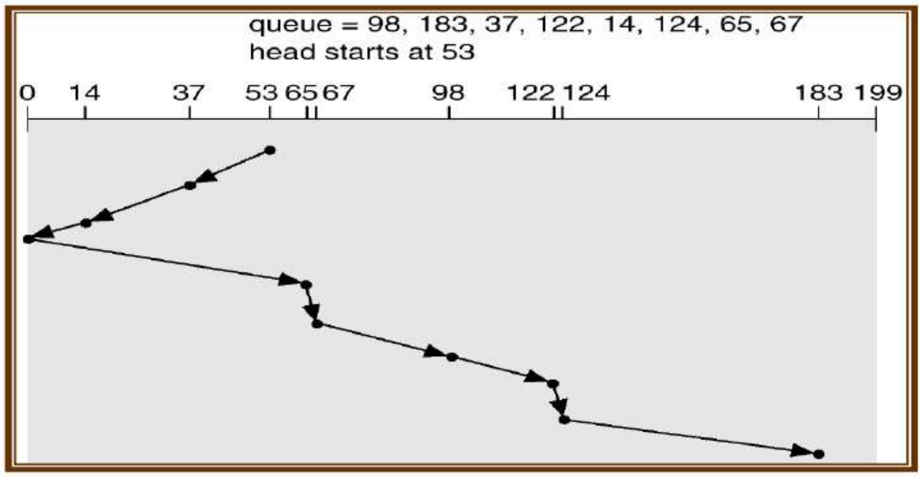
• The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.

• Sometimes called the elevator algorithm.

• The head starts at the one end of the disk and moves toward on the another end. it serves all the request coming in the way.

• After reaching another end the direction of head movement is reverse

• Illustration shows total head movement of 236 cylinders.



Total head movement = 16+23+14+65+2+31+24+2+59
= 236

- **Advantages :**
    - More efficient than FCFS.
    - No starvation for any request.

- **Disadvantages :** Not so fair, because cylinder which are just behind the head will wait longer.
    - Requires extra head movement between two extreme points.
    - For example, after serving 5th cylinder, there is no need to visit 0th cylinder.

C-SCAN [Circular SCAN] SCHEDULING

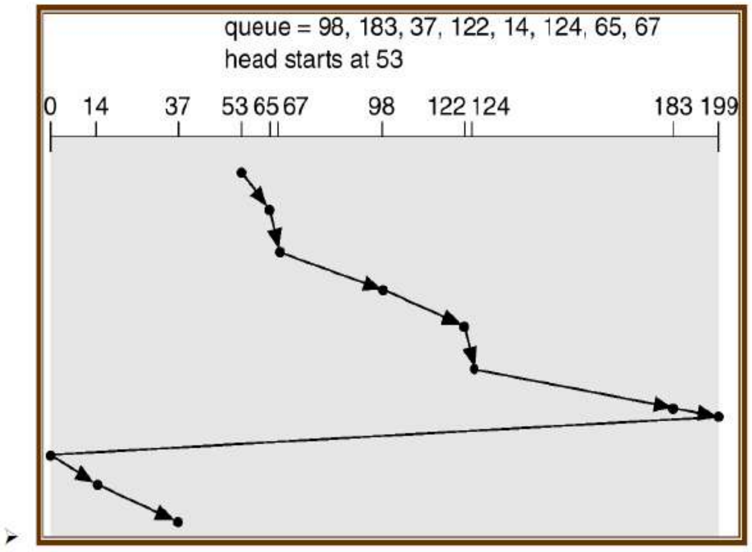
• The circular SCAN algorithm improves upon SCAN b treating all the request in a circular queue fashion.- once the head reaches the end of the disk, it returns to the other end without processing any request, and then starts again from the beginning of the disk.

• The head moves from one end of the disk to the other. servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip

• Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.

• Illustration shows total head movement of 382 cylinders, plus return time.

- Provides a more uniform wait time than SCAN.



Total head movement = 12+2+31+24+2+59+16+199+14+23
= 382

LOOK

•it is same like SCAN scheduling but the difference that end points are not visited unnecessary.

•Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.

•**Advantages :**

•More efficient than FCFS.

•No starvation for any request.

•No extra head movement.

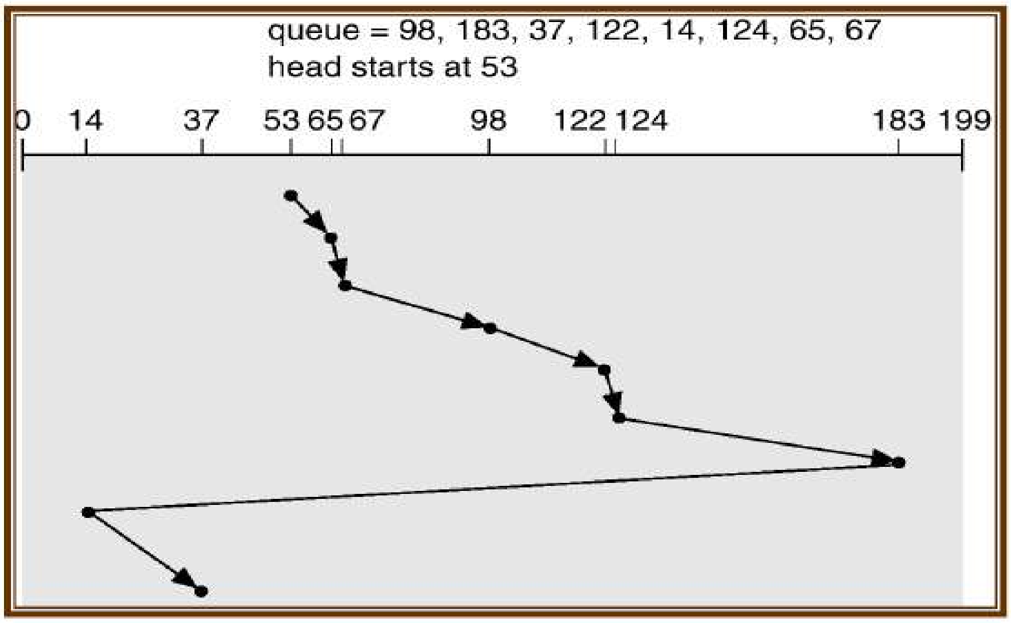
•**Disadvantages :**

•Not so fair, because cylinder behind the head with longer.

C-LOOK [circular – LOOK]

• Version of C-SCAN

• Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.



6.3.2 Selection of a Disk-Scheduling Algorithm

- With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
- For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
- For busier systems, SCAN and LOOK algorithms eliminate starvation problems.
- The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.
- Some improvement to overall filesystem access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.

- On modern disks the rotational latency can be almost as significant as the seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, (particularly when bad blocks have been remapped to spare sectors.)
    - Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, (which do know the actual geometry of the disk as well as any remapping), so that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.
    - Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason OSes may elect to spoon-feed requests to the disk controller one at a time in certain situations.

6.4 Disk Management

6.4.1 Disk Formatting

- Before a disk can be used, it has to be ***low-level formatted***, which means laying down all of the headers and trailers marking the beginning and ends of each sector. Included in the header and trailer are the linear sector numbers, and ***error-correcting codes, ECC,*** which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.) Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.
- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a ***soft error*** has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.)
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.
- After partitioning, then the filesystems must be ***logically formatted,*** which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the filesystem. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the filesystem structure, but requires that the application program manage its own disk storage requirements.)

6.4.1 Boot Block

- Computer ROM contains a ***bootstrap*** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)
- The first sector on the hard drive is known as the ***Master Boot Record, MBR,*** and contains a very small amount of code in addition to the ***partition table.*** The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the ***active*** or ***boot*** partition.
- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a ***dual-boot*** (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.
- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include ***single-user*** a.k.a. ***maintenance*** or ***safe*** modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.

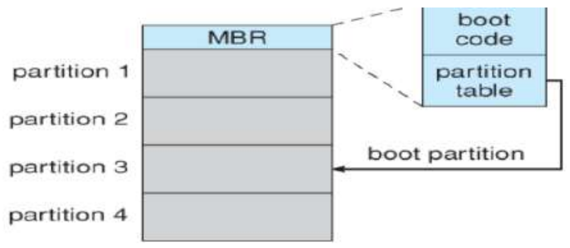


Figure - Booting from disk in Windows 2000.

6.4.2 Bad Blocks

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.

- In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)
- Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (Recall that blocks are tested with every write as well as with every read, so often errors can be detected before the write operation is complete, and the data simply written to a different sector instead.)
- Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimization of the OS, especially if the replacement sector is physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder. Whenever possible a bad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. *Sector slipping* may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.
- If the data on a bad block cannot be recovered, then a *hard error* has occurred., which requires replacing the file(s) from backups, or rebuilding them from scratch.

6.5 RAID Structure

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- *RAID* originally stood for ***Redundant*** *Array of Inexpensive Disks,* and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to *Independent* disks.

6.5.1 Improvement of Reliability via Redundancy

- The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually *decreases* the Mean Time To Failure, MTTF of the system.
- If, however, the same data was copied onto multiple disks, then the data would not be lost unless both (or all) copies of the data were damaged simultaneously, which is a MUCH lower probability than for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the Mean

Time To Repair into play. For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the Mean Time to Data Loss would be 500 * 10^6 hours, or 57,000 years!

- This is the basic idea behind disk *mirroring*, in which a system contains identical data on two or more disks.
    - Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

6.5.2 Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.)
- Another way of improving disk access time is with ***striping***, which basically means spreading data out across multiple disks that can be accessed simultaneously.
    - With ***bit-level striping*** the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.
    - ***Block-level striping*** spreads a filesystem across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when filesystems are accessed in ***clusters*** of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

6.5.3 RAID Levels

Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different ***RAID levels***, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)

1. ***Raid Level 0 -*** This level includes striping only, with no mirroring.
2. ***Raid Level 1 -*** This level includes mirroring only, no striping.
3. ***Raid Level 2 -*** This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)
4. ***Raid Level 3 -*** This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance.
5. ***Raid Level 4 -*** This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.
6. ***Raid Level 5 -*** This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.
7. ***Raid Level 6 -*** This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the ***Reed-Solomon codes***), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

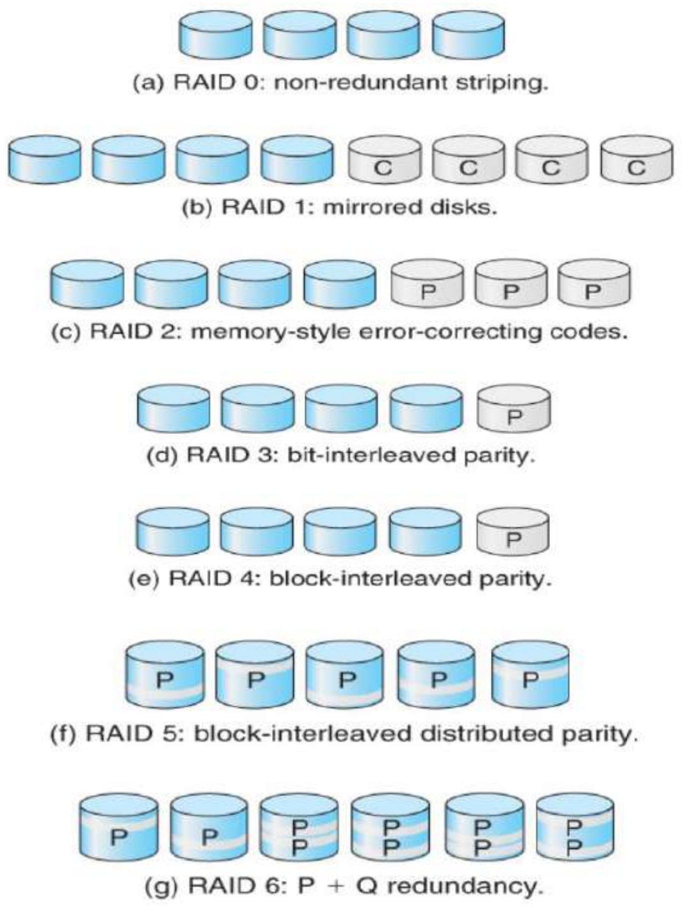

(a) RAID 0: non-redundant striping.
C
(b) RAID 1: mirrored disks.
P
(c) RAID 2: memory-style error-correcting codes.
P
(d) RAID 3: bit-interleaved parity.
P
(e) RAID 4: block-interleaved parity.
P
(f) RAID 5: block-interleaved distributed parity.
P
(g) RAID 6: P + Q redundancy.

Figure 4.5- RAID levels

There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.

- **RAID level 0 + 1** disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
- **RAID level 1 + 0** mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.
    - In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.
        - If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.
        - However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.

- In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.
    - If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.
    - Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.
    - In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

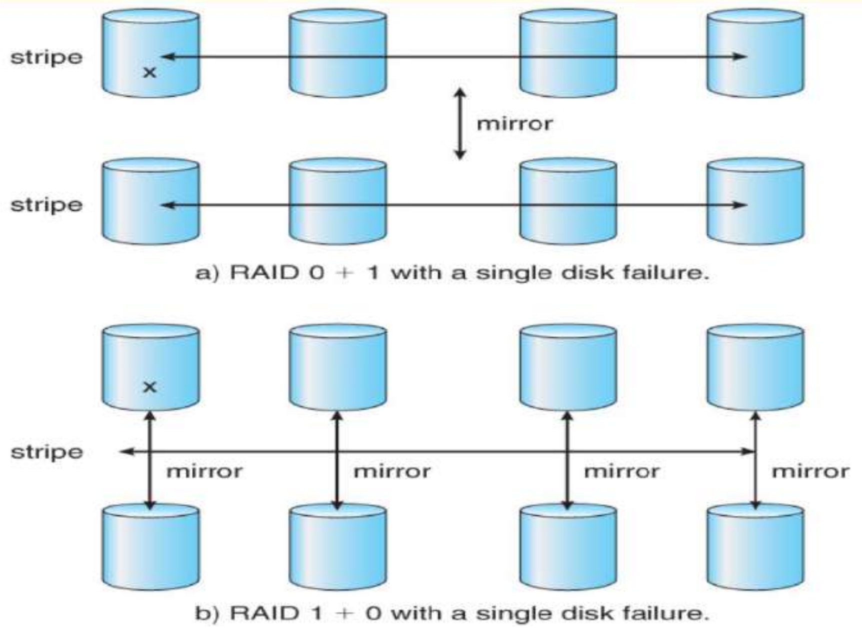

stripe
x
mirror
stripe
a) RAID 0 + 1 with a single disk failure.
x
stripe
mirror
mirror
mirror
mirror
b) RAID 1 + 0 with a single disk failure.

Figure 4.6 - RAID 0+1 and 1+0

6.5.4 Selection of RAID Levels

1. RAID level 0 is a right choice when data safety and its security is not a big case. Thus, level 0 is used in high-performance applications.
2. The designers can go for RAID level 1 for rebuilding the data. It is because rebuilding is the simplest job for level 1. As in RAID level 1, the user can copy the data from another disk. In case of other levels, it is required to access all other disks in the array for rebuilding the data of a failed disk. Build performance is an important factor in high-performance database systems. In fact, the time taken to rebuild the data may become

a significant part of the repair time, so rebuild performance also influence the meantime for data loss.

3. RAID level 3 and RAID level 5 are so powerful that they have restricted the selection of RAID level 2 and RAID level 4 by absorbing them. The block striping feature of RAID level 5 inferiors bit striping feature of the RAID level 3. It is because the block striping provides good data transfer rates for large transfers, and uses a few disks for making small data transfers. In the case of small data transfer, the access time dominates, which, as a result, diminishes the benefits of the parallel reads. RAID level 3 can also be proved as a bad choice for making small data transfers. It is because the data transfer finishes only after each disk has fetched the corresponding sectors over them. It leads the average latency for the disk array closer to the worst latency for a single disk, where the benefits of the high data transfer rates are being ignored.

4. While comparing RAID level 6 with RAID level 5, it offers a good reliability option than RAID level 5. Also, designers can use RAID level 6 in applications where data safety and security is a major concern. But, currently, many RAID implementations do not support RAID level 6.

5. In some cases, it is difficult to choose between RAID level 1 and RAID level 5. RAID level 1 is good for applications like storage of log files in the database system as it offers the best write performance. Such a feature of RAID level 1 is not comparable with the remaining other five RAID levels. On the other hand, RAID level 5 offers low storage overhead in comparison to RAID level 1. But it takes high time overhead for write performance. Thus, it is better to choose RAID level 5 for those applications where data is read frequently but written rarely.

6. Although the disk-storage demand increases with time per year, cost per byte on the other hand, falls at the same rate. Consequently, this has led to the need for the monetary cost of extra storage at a significant level. However, the access time is increasing day by day at a slower rate, which has shown a high increase in the number of input-output operations per second. So, RAID level 1 and RAID level 5 have become the most moderate choices among all other RAID levels because RAID level 5 provides high input-output requirements, and RAID level 1 offers moderate storage requirements for the data.