

Operating System

UNIT -1

❖ Introduction to Operating Systems:

- An operating System (OS) is an intermediary between users and computer hardware. It provides users an environment in which a user can execute programs conveniently and efficiently.
- . An operating System controls the allocation of resources and services such as memory, processors, devices and information.
- A computer system has many resources (hardware and software), which may be require to complete a task. The commonly required resources are input/output devices, memory, file storage space, CPU etc.
- The operating system acts as a manager of the above resources and allocates them to specific programs and users, whenever necessary to perform a particular task.
- Therefore operating system is the resource manager i.e. it can manage the resource of a computer system internally.
- The resources are processor, memory, files, and I/O devices. In simple terms, **an operating system is the interface between the user and the machine.**

What is Operating System?

An operating system is an important part of almost every computer system. A computer system can be divided roughly into **four components**: the **hardware**, the **operating system**, the **applications programs**, and the **users (figure)**.

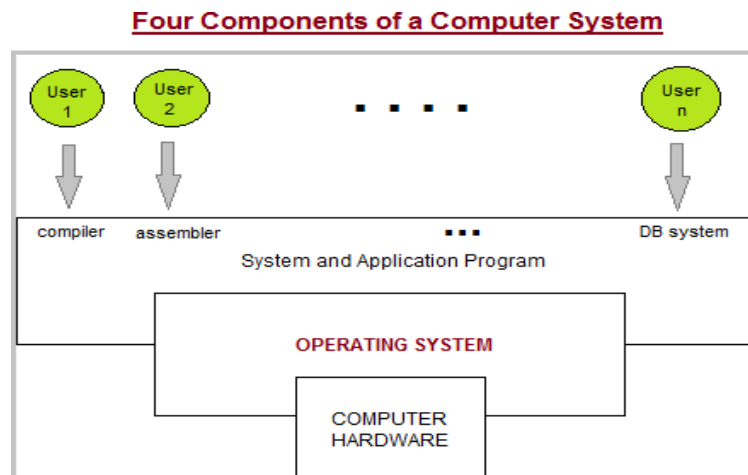


Figure: Abstract view of the components of a computer system

- The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system.
- The **application programs**—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve the computing problems of the users.
- The operating system controls and coordinates the use of the hardware among the various application programs for the various users.
- The components of a computer system are its hardware, software, and data.

- The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a *government*.
- Operating systems can be explored from two viewpoints: the user and the system.

Two Views of Operating System

- User's View
- System View

User View:

- The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources.
- The goal is to maximize the work (or play) that the user is performing.
- A user sits at a terminal connected to a **mainframe** or **minicomputer**.
- In still other cases, users sit at **workstations** connected to networks of other workstations and **servers**.
- For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

System View:

- From the computer's point of view, the operating system is the program most intimately involved with the hardware. we can view an operating system as a **resource allocator**.
- A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. An operating system is a control program.
- A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices

❖ Functions of Operating System:

- An operating system is a program that acts as a user-computer GUI (Graphical user interface). It controls the execution of all types of applications
- .The operating system performs the following functions in a device.

1. Instruction
2. Input/output Management
3. Memory Management
4. File Management
5. Processor Management
6. Job Priority
7. Special Control Program
8. Scheduling of resources and jobs
9. Security
10. Monitoring activities
11. Job accounting

- ✚ **Instruction:** The [operating system](#) establishes a mutual understanding between the various instructions given by the user.
- ✚ **Input/output Management:** What output will come from the input given by the user, the operating system runs this program.
- ✚ This management involves coordinating various input and output devices. It assigns the functions of those devices where one or more applications are executed.
- ✚ **Memory Management:** The operating system handles the responsibility of storing any data, system programs, and user programs in memory.
- ✚ This function of the operating system is called memory management.
- ✚ **File Management:** The operating system is helpful in making changes in the stored files and in replacing them. It also plays an important role in transferring various files to a device.
- ✚ **Processor Management:** The processor is the execution of a program that accomplishes the specified work in that program. It can be defined as an execution unit where a program runs.
- ✚ **Job Priority:** The work of job priority is creation and promotion. It determines what action should be done first in a computer system.
- ✚ **Special Control Program:** The operating systems make automatic changes to the task through specific control programs. These programs are called Special Control Program.
- ✚ **Scheduling of resources and jobs:** The operating system prepares the list of tasks to be performed for the device of the computer system.
- ✚ **Security:** Computer security is a very important aspect of any operating system. The reliability of an operating system is determined by how much better security it provides us. Modern operating systems use a firewall for security. A firewall is a security system that monitors every activity happening in the computer and blocks that activity in case of any threat.
- ✚ **Monitoring activities:** The operating system takes care of the activities of the computer system during various processes. This aborts the program if there are errors. The operating system sends instant messages to the user for any unexpected error in the input/output device. It also provides security to the system when the operating system is used in systems operated by multiple users. So that illegal users cannot get data from the system.
- ✚ **Job accounting:** It keeps track of time & resources used by various jobs and users.

Types of Operating Systems:

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

- ❖ Simple Batch System
- ❖ Multiprogramming Batch System
- ❖ Multiprocessor System
- ❖ Desktop System
- ❖ Distributed Operating System
- ❖ Clustered System
- ❖ Real time Operating System
- ❖ Handheld System

1. Simple Batch Systems:

The user has to submit a job (written on cards or tape) to a computer operator.

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

Advantages of Simple Batch Systems

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



Figure: Memory layout for a simple batch system

2. Multiprogramming Batch Systems:

In this the operating system picks up and begins to execute one of the jobs from memory.

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

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

Time Sharing Systems are very similar to Multiprogramming batch systems. In fact time sharing systems are an extension of multiprogramming systems.

In Time sharing systems the prime focus is on **minimizing the response time**, while in multiprogramming the prime focus is to maximize the CPU usage.



Figure: Memory layout for a multiprogramming system.

3. Multiprocessor system

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

Advantages of Multiprocessor Systems

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

4. Desktop Systems:

- Earlier, CPUs and PCs lacked the features needed to protect an operating system from user programs.
- PC operating systems therefore were neither **multiuser** nor **multitasking**.
- However, the goals of these operating systems have changed with time; instead of maximizing CPU and peripheral utilization, the systems opt for maximizing user convenience and responsiveness.
- These systems are called **Desktop Systems** and include PCs running **Microsoft Windows** and the **Apple Macintosh**.
- Operating systems for these computers have benefited in several ways from the development of operating systems for **mainframes**.
- **Microcomputers** were immediately able to adopt some of the technology developed for larger operating systems.
- On the other hand, the hardware costs for microcomputers are sufficiently **low** that individuals have sole use of the computer, and CPU utilization is no longer a prime concern.

Thus, some of the design decisions made in operating systems for mainframes may not be appropriate for smaller systems.

5. Distributed Operating System

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

Advantages Distributed Operating System:

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

Types of Distributed Operating Systems

Following are the two types of distributed operating systems used:

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

Client-Server Systems

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

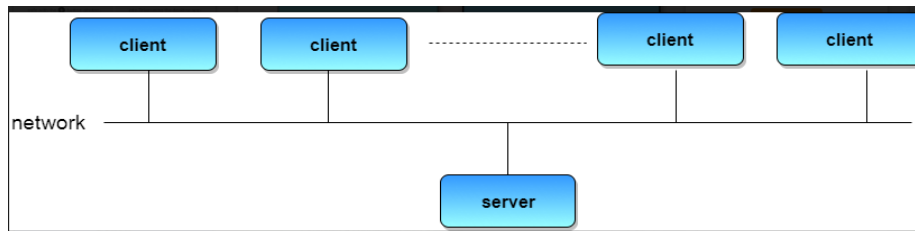


Figure: General structure of a client-server system.

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

Peer-to-Peer Systems

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

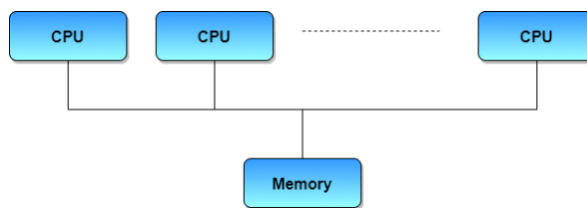


Figure: Symmetric multiprocessing architecture.

6. Clustered Systems

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

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

7. Real Time Operating System

- It is defined as an operating system known to give maximum time for each of the critical operations that it performs, like OS calls and interrupt handling.

- The Real-Time Operating systems which guarantees the maximum time for critical operations and complete them on time are referred to as **Hard Real-Time Operating Systems**.
- While the real-time operating systems that can only guarantee a maximum of the time, i.e. the critical task will get priority over other tasks, but no assurity of completing it in a defined time. These systems are referred to as **Soft Real-Time Operating Systems**.

8. Handheld Systems

- Handheld systems include **Personal Digital Assistants(PDAs)**, such as **Palm-Pilots** or **Cellular Telephones** with connectivity to a network such as the Internet.
- They are usually of limited size due to which most handheld devices have a small amount of memory, include slow processors, and feature small display screens.
- Many handheld devices have between **512 KB** and **8 MB** of memory. As a result, the operating system and applications must manage memory efficiently.
- This includes returning all allocated memory back to the memory manager once the memory is no longer being used.
- Currently, many handheld devices do **not use virtual memory** techniques, thus forcing program developers to work within the confines of limited physical memory.
- Processors for most handheld devices often run at a fraction of the speed of a processor in a PC. Faster processors require **more power**. To include a faster processor in a handheld device would require a **larger battery** that would have to be replaced more frequently.
- The last issue confronting program designers for handheld devices is the small display screens typically available. One approach for displaying the content in web pages is **web clipping**, where only a small subset of a web page is delivered and displayed on the handheld device.

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

Chapter-1

1. Computer System Operation:

- I/O devices and the CPU can execute concurrently.
- Each device controller is in charge of a particular device type.
- Each device controller has a local buffer , a temporary data storage holding place.
- Devices include such things as a printer, a disk, a tape drive, etc.
- CPU moves data from/to main memory to/from local buffers along the system bus, which is just a set of wire connections along which data can be sent.
- I/O is from the device to local buffer of controller.
- Device controller informs CPU that it has finished its operation by causing an interrupt.

The modern computer system consists of the CPU and a number of device controllers are connected through a common bus that provides access to shared memory (figure).

Each device controller is in charge of a specific type of device (for example: disk drives, audio devices, and video display).

The CPU and the device controllers can execute concurrently, competing for memory cycles.

To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

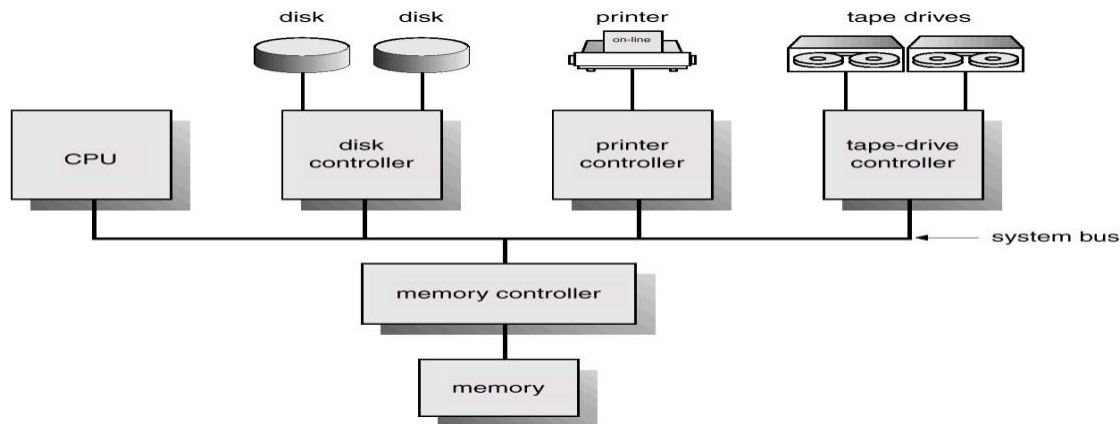


Figure: A modern computer system

- For a computer to start running – for instance, when it is powered up or rebooted—it needs to have an initial program to run.
- This initial program, or **bootstrap program**, tends to be simple.
- Typically, it is stored in **read-only memory (ROM)** such as firmware or EEPROM within the computer hardware.
- It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
- The bootstrap program must know how to load the operating system and to start executing that system.
- To accomplish this goal, the bootstrap program must locate and load into memory the operating-system kernel.
- The OS then starts executing the first process, such as “init,” and waits for some event to occur.
- The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.
- H/W may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
- S/W may trigger an interrupt by executing a special called a **system call** (also called a **monitor call**).

Modern OS are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to responds, an OS will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (**for example**, division by zero or invalid memory access) or by a specific request from a user program that an OS service be performed.

Common Functions of Interrupts:

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction.
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt.
- A trap is a software-generated interrupt caused either by an error or a user request.

- An operating system is interrupt driven.

Interrupt Handling:

The operating system preserves the state of the CPU by storing registers and the program counter.

- Determines which type of interrupt has occurred:
 - ◆ Polling
 - ◆ vectored interrupt system
- Separate segments of code determine what action should be taken for each type of interrupt

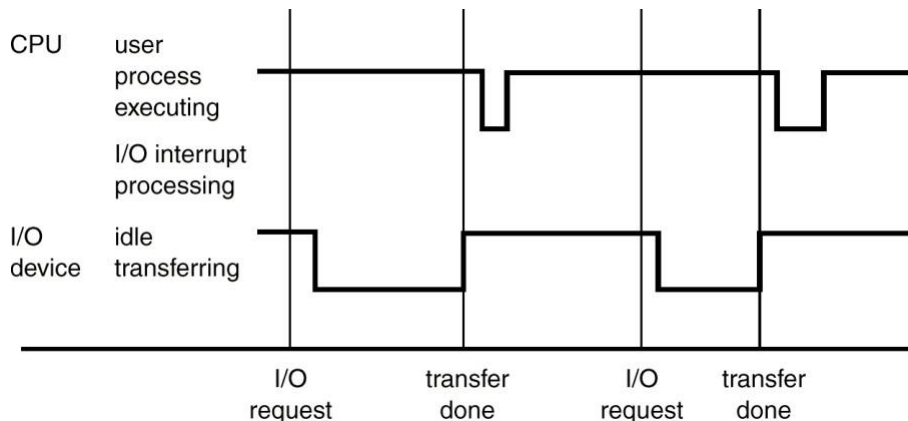


Fig: Interrupt Time Line for a Single Process Doing Output

2. I/O Structure:

Depending on the controller, there may be more than one attached device. For instance, seven or more devices can be attached to the small computer-systems interface (SCSI) controller.

A device controller maintains some local buffer storage and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Typically, operating systems have a device driver for each device controller. This device driver understands the device controller and presents a uniform interface to the device to the rest of the operating system.

After I/O starts, control returns to user program only upon I/O completion.

- ◆ Wait instruction idles the CPU until the next interrupt
- ◆ Wait loop (contention for memory access).
- ◆ At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control returns to user program without waiting for I/O completion.
- ◆ System call – request to the operating system to allow user to wait for I/O completion.
- ◆ Device-status table contains entry for each I/O device indicating its type, address, and state.
- ◆ Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt.

- I/O Interrupts
- DMA Structure

I/O Interrupts:

- To start an I/O operation, the device driver loads the appropriate registers within the device controller.
- The device controller, in turn, examines the contents of these registers to determine what action to take.
- For example, if it find a read request, the controller starts the transfer of data from the device to its local buffer.
- Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation.
- The device driver then returns control to the operating system, possibly returning the data or a pointer to the data if the operation was a read.
- For other operations, the device driver returns status information.

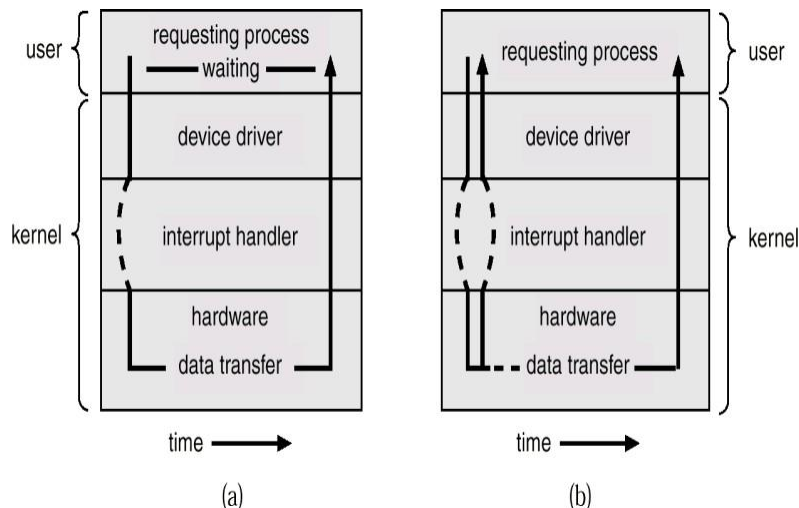


Figure: Two i/o methods (a) synchronous and (b) asynchronous

- Once the I/O is started, two courses of action are possible. In the simple case, the I/O is started; then, at I/O completion, control is returned to the user process. This case is known as **synchronous** I/O.
- The other possibility, called **asynchronous** I/O, returns control to the user program without waiting for the I/O to complete. The I/O then can continue while other system operations occur shown in figure.
- Waiting for I/O completion may be accomplished in one of two ways. Some computers have a special **wait** instruction that idles the CPU until the next interrupt.
- Machines that do not have such an instruction may have a wait loop:
Loop: jmp Loop
 - The CPU always waits for I/O completion, at most one I/O request is outstanding at a time.
 - A system call is then needed to allow the user program to wait for I/O completion, if desired.
 - If no user program are ready to run, and the operating system has no other work to do, we still require the **wait** instruction or idle loop, as before.
 - We also need to be able to keep track of many I/O requests at the same time.
 - The operating system uses a table containing an entry indicates the device: the **device-status table**(figure).
 - Each table entry indicates the device's type, address, and state.

- If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.
- Since it is possible for other processes to issues **requests** to the same device, the operating system also maintain a wait queue-a list of waiting requests-for each I/O device.
- If there additional requests waiting in the queue for this device, the operating system starts processing the next request.
- Finally,control is returned from the I/O interrupt.

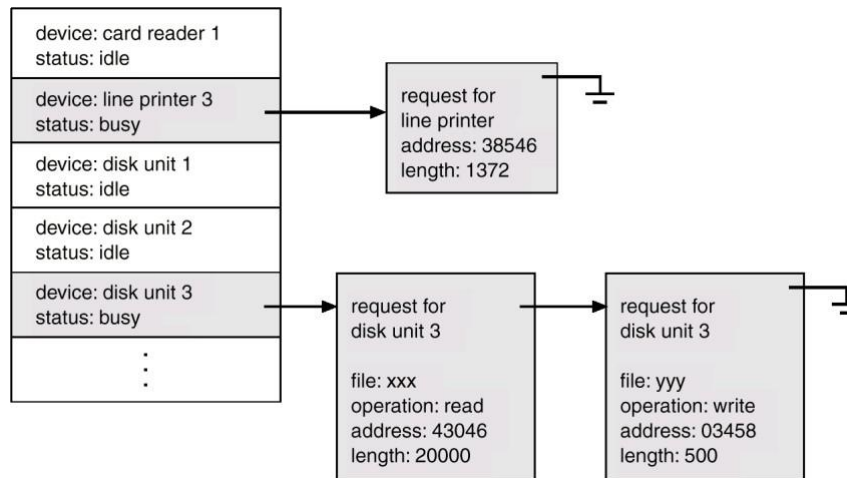


Figure: Device-Status Table

Direct Memory Access Structure:

- Direct memory access(DMA) is a method that allows an input /output (i/o) device to send or receive data directly to or from the main memory,by passing the CPU to speed up memory operations.
- The process is managed by a chip known as a DMA controller(DMAC).

➤ Used for high-speed I/O devices able to transmit information at close to memory speeds.

■ Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.

■ Only one interrupt is generated per block, rather than the one interrupt per byte.

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.

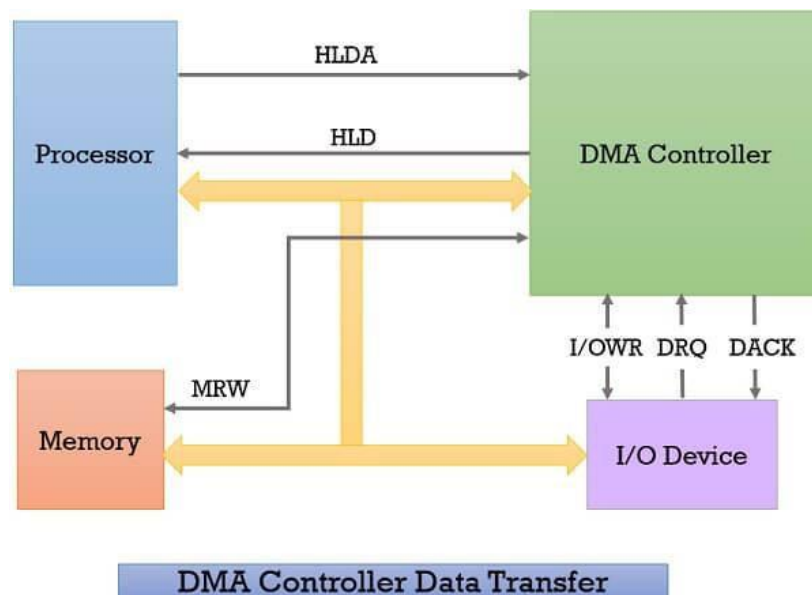
The above two modes of data transfer are not useful for transferring a large block of data. But, the DMA controller completes this task at a faster rate and is also effective for transfer of large data block.

The DMA controller transfers the data in **three modes**:

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.

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.

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



Whenever an I/O device wants to transfer the data to or from memory, it sends the DMA request (**DRQ**) to the DMA controller. DMA controller accepts this DRQ and asks the CPU to hold for a few clock cycles by sending it the Hold request (**HLD**).

CPU receives the Hold request (HLD) from DMA controller and relinquishes the bus and sends the Hold acknowledgement (**HLDA**) to DMA controller.

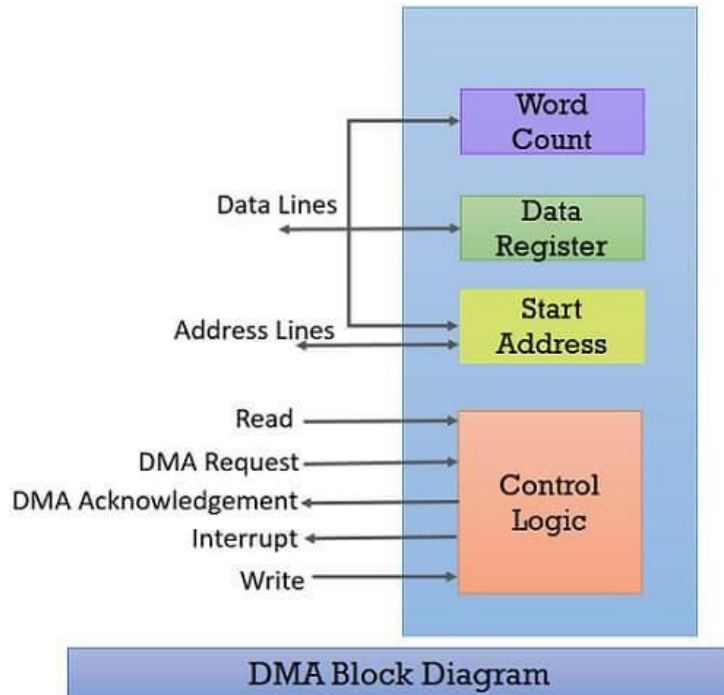
After receiving the Hold acknowledgement (HLDA), DMA controller acknowledges I/O device (**DACK**) that the data transfer can be performed and DMA controller takes the charge of the system bus and transfers the data to or from memory.

When the data transfer is accomplished, the DMA raise an **interrupt** to let know the processor that the task of data transfer is finished and the processor can take control over the bus again and start processing where it has left.

Now the DMA controller can be a separate unit that is shared by various I/O devices, or it can also be a part of the I/O device interface.

Direct Memory Access Diagram:

After exploring the working of DMA controller, let us discuss the block diagram of the DMA controller. Below we have a block diagram of DMA controller.



Whenever a processor is requested to read or write a block of data, i.e. transfer a block of data, it instructs the DMA controller by sending the following information.

The first information is whether the data has to be read from memory or the data has to be written to the memory. It passes this information via **read or write control lines** that is between the processor and DMA controllers **control logic unit**.

The processor also provides the **starting address** of/ for the data block in the memory, from where the data block in memory has to be read or where the data block has to be written in memory. DMA controller stores this in its **address register**. It is also called the **starting address register**.

The processor also sends the **word count**, i.e. how many words are to be read or written. It stores this information in the **data count** or the **word count register**.

The most important is the **address of I/O device** that wants to read or write data. This information is stored in the **data register**.

Advantages:

- Transferring the data without the involvement of the processor will **speed up** the read-write task.
- DMA **reduces the clock cycle** requires to read or write a block of data.
- Implementing DMA **also reduces the overhead of the processor**.

Disadvantages:

- As it is a hardware unit, it would **cost** to implement a DMA controller in the system.
- Cache **coherence** problem can occur while using DMA controller.

DMA controller transfers the data block at the **faster rate** as data is directly accessed by I/O devices and is not required to pass through the processor which save the clock cycles.

DMA controller transfers the block of data to and from memory in three modes **burst mode**, **cycle steal mode** and **transparent mode**.

DMA can be configured in various ways it can be a part of individual I/O devices, or all the peripherals attached to the system may share the same DMA controller.

Thus the DMA controller is a convenient mode of data transfer. It is preferred over the programmed I/O and Interrupt-driven I/O mode of data transfer.

3. Storage Structure:

- Computer programs must be in main memory(also called random-access memory or RAM) to be executed.
- Main memory – only large storage area(millions to billions of bytes) that the CPU can access directly.
- It is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**, which forms an array of memory words.
- Each word has its own address. Interaction is achieved through a sequence of load or store instruction to specific memory addresses.
- The load instruction moves a word ,from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.
- Aside from explicit load and store, the CPU automatically loads instructions from main memory for execution.

A typically instruction-execution cycle, as executed on a system with a **von Neumann architecture**, will first fetch instruction from memory and will store that instruction in the instruction register.

- The programs and data to reside in main memory permanently.
- This arrangement is not possible for the following two reasons:
 1. **Main memory is usually too small to store all needed programs and data permanently.**
 2. **Main memory is a volatile storage device that loses its contents when power is turned off or otherwise lost.**
- The most computer system provide **secondary storage** as an extension of main memory.
- The main requirements for secondary storage is that it be able to hold large quantities of data permanently.
- ◆ Disk surface is logically divided into tracks, which are subdivided into sectors.
- ◆ The disk controller determines the logical interaction between the device and the computer.
- The most common secondary –storage device is a **magnetic disk**, which provides storage of both programs and data.
- Most programs(web browsers, compilers, word processors, spread sheets, etc) are store on a disk until they are loaded into memory.

1. Main Memory:

- Main memory refers to a physical memory that is the internal memory to the computer.
- Main memory is also known as RAM and primary storage.
- The computer is able to change only data that is in **main memory**.
- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- Usually, special I/O instructions allow data transfers between these **registers** and **system memory**.
- To allow more convenient access to I/O devices, many computer architectures provide **memory-mapped I/O**.
- Read and writes to these memory addresses cause the data to be transferred to and from the device registers.
- Memory-mapped I/O is also convenient for other devices, such as the serial and parallel ports used to connect modem and printers to a computer.
- The CPU transfers data through these kinds of devices by reading and writing a few device registers, called an **I/O Port**.
- A memory buffer used to accommodate a speed differential, called a **cache**.

2. Magnetic Disks:

- Magnetic disks provide the bulk of secondary storage for modern computer systems.
- Conceptually, disks are relatively simple as shown in figure.

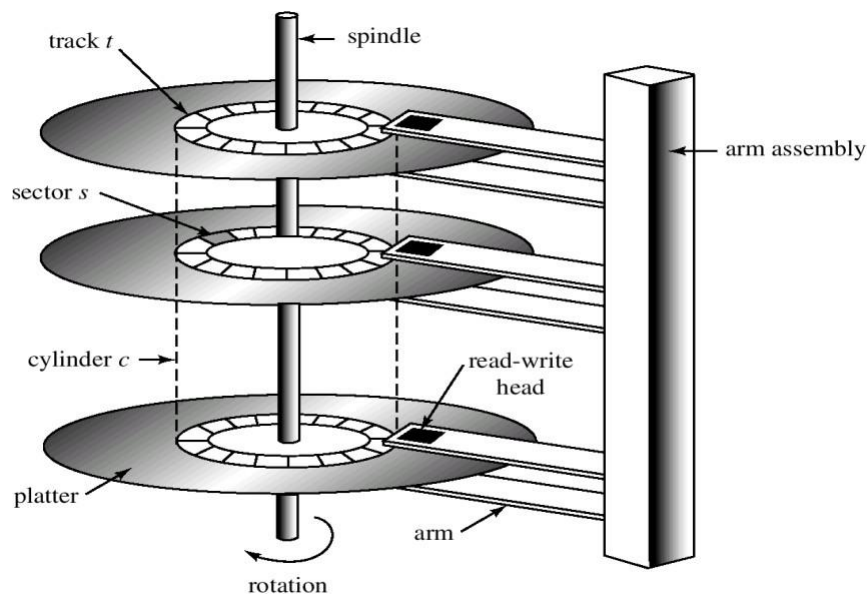


Figure: Moving-Head Disk Mechanism.

- Each disk **platter** has a flat circular shape, like a CD.
- Common platter diameters range from 1.8 to 5.25 inches.
- The two surfaces of a platter are covered with a magnetic material.
- We store information by recording it magnetically on the platters.

A read – write head “flies” just above each surface of every platter. The heads are attached to a **disk arm**, which moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into sectors. The set of tracks that are at one arm position forms a **cylinder**.

- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 200 times per second. Disk speed has two parts. The **transfer rate** is the rate at which dataflow between the drive and the computer.
- The **positioning time**, sometimes called the **random-access time**, consists of the time to move the disk arm to the desired cylinder, called the **seek time**, and the time for the desired sector to rotate to the disk head, called **rotational latency**.
- The disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a head crash.
 - A disk can be removable, allowing different disks to be mounted as needed.
 - Removable magnetic disks generally consists of one platter, held in a plastic case to prevent damage while not in the disk drive.
 - Floppy disks are inexpensive removable magnetic disks that a soft plastic case containing a flexible platter.
- A disk drive is attached to a computer by a set of wires called an I/O bus. Several kinds of buses are available, including **enhanced integrated drive electronics (EIDE)**, **advanced technology attachment (ATA)**, and **SCSI** buses.
- The data transfers on a bus are carried out by special electronic processors called **controllers**. The host controller is the controller at the computer end of the bus. A **disk controller** is build into each disk drive.

3. Magnetic Tapes:

- **Magnetic tape** is a medium for magnetic recording, made of a thin, coating on a long, narrow strip of plastic film.
- A device that stores computer data on magnetic tape is known as a tape drive.
- **Magnetic tape** was used as an early secondary-storage medium.
- Although it is relatively permanent and can hold large quantities of data, its access time is slow in comparision to that of main memory.
- A tape is kept in a spool, and is wound or rewound past a read – write head.
- Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.
- Tape capacities vary greatly, depending on the particular kind of tape drive. Tape hold 2 to 3 times more data than some does a large disk drive.
- Tapea and their drivers are usually categorized by width, including 4,8, and 19 millimeters, 1/2 and 1/2 inch.

4. Storage Hierarchy:

- Storage systems organized in hierarchy.
 - Speed
 - Cost
 - Volatility
- Caching – copying information into faster storage system; main memory can be viewed as a last cache for secondary storage.
- In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper.
- The top three levels of memory shown in figure may be constructed using semiconductor memory.
- In addition to having differing speed and cost, the various storage systems are either **volatile or nonvolatile**.

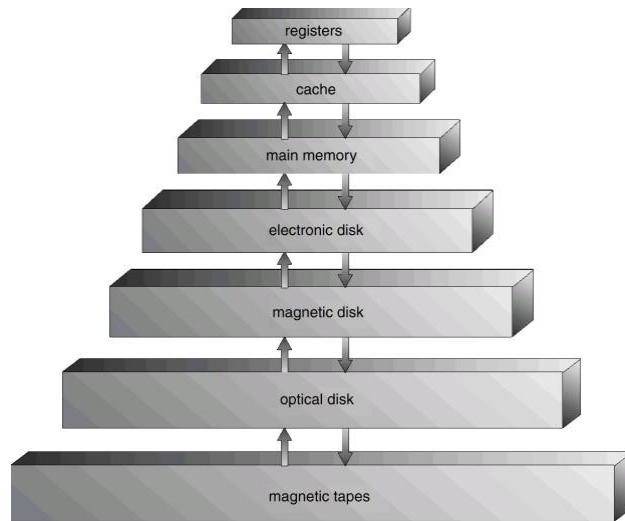
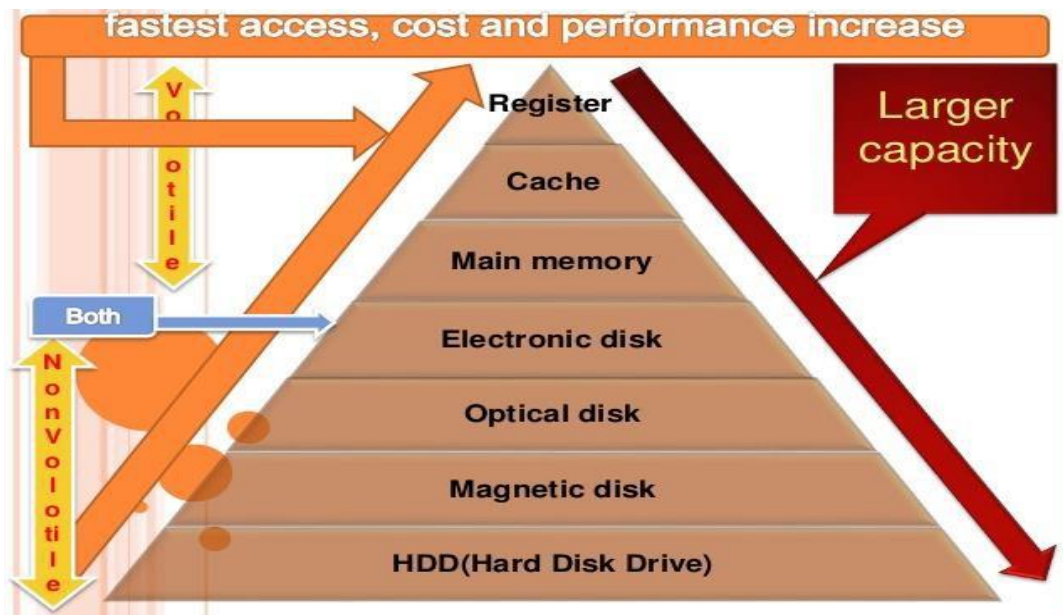


Figure: Storage-device hierarchy.

- **Volatile Storage** is a type of computer memory that needs power to preserve stored data. If the computer is switched off, anything stored in the volatile memory is removed or deleted.
- All random access memory (RAM) other than the CMOS RAM used in the BIOS is volatile. RAM is typically used as a primary storage or main memory in computer systems.
- Volatile storage is also known as volatile memory or temporary memory.
- **Non-volatile** memory is typically used for the task of **secondary storage**, or long-term persistent storage.



- The most widely used form of **primary storage** today is a volatile form of **random access memory (RAM)**, meaning that when the computer is shut down, anything contained in RAM is lost.
- In the hierarchy shown in figure, the storage systems above the **electronic disk** are **volatile**, whereas those below are nonvolatile.
- An electronic disk can be designed to be either volatile or nonvolatile.
- Caches can be installed to improve performance where a large access-time or transfer-rate disparity exists between **two components**.

- **1. Caching**
- **2. Coherency and Consistency**

Caching:

- **Caching** is an important principle of computer systems. Information is normally kept in some storage system (such as main memory).
- As it is used, it is copied into a faster storage system-the cache-on a temporary basis.
- If it is, we use the information directly from the cache; if it is not, we use the information from the main storage system, putting a copy in the cache under the assumption that we will need it again.
- Because caches have limited size, cache management is an important design problem.
- Careful selection of the cache size and of a replacement policy can result in **80 to 99 percent** of all accesses being in the cache, greatly increasing performance.
- Main memory can be viewed as a fast cache for secondary storage, since data in secondary storage must be copied into main memory for use, and data must be in main memory before being moved to secondary storage for safekeeping.
- The file-system data, which resides permanently on secondary storage, may appear on several levels in the storage hierarchy.
- At the highest level, the operating system may maintain a cache of file-system data in main memory.
- Also, electronic RAM disks (also known as solid-state disks) may be used for high-speed storage that is accessed through the file-system interface.
- The bulk of secondary storage is on magnetic disks.
- Use of high-speed memory to hold recently-accessed data.
 - Requires a cache management policy.
 - Caching introduces another level in storage hierarchy.
- This requires data that is simultaneously stored in more than one level to be consistent.

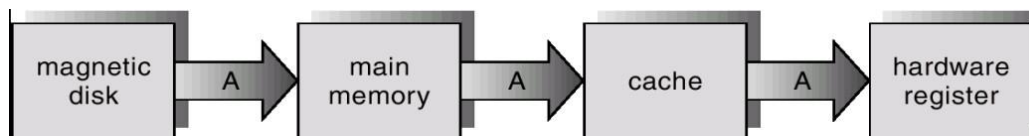


Figure: Migration of A from Disk to Register.

Coherency and Consistency:

- In a hierarchical storage structure, the same data may appear in different levels of the storage system.
- **For example**, suppose that an integer **A** is located in file **B** that is to be incremented by **1**, and file **B** resides on magnetic disk.
- The increment operation proceeds by first issuing an I/O operation to copy the disk block on which **A** resides to main memory.
- This operation is followed by copying **A** to the cache and to an internal register.
- Thus, the copy of **A** appears in several places:
 - On the magnetic disk,
 - In main memory,
 - In the cache and
 - In an internal register
- shown in figure. Once the increment takes place in the internal register, the value of **A** differs in the various storage systems.
- The value of **A** becomes the same only after the new value of **A** is written from the internal register back to the magnetic disk.
- ❖ The situation becomes more complicated in a multiprocessor environment where, in addition to maintaining internal registers, each of the CPUs also contains a local cache.
- ❖ In such an environment, a copy of **A** may exist simultaneously in several caches.

- ❖ Since the various CPUs can all execute concurrently, we must make sure that an update to the value of **A** in one cache is immediately reflected in all other caches where **A** resides.
- ❖ This situation is called **cache coherency**, and it is usually a hardware problem.

5. Hardware Protection

- Early computer systems were single-user programmer-operated systems.
- When the programmers operated the computer from the console, they had complete control over the system.
- As operating systems developed, however, this control was given to the operating system.
- Early operating systems were called **resident monitors**, and starting with the resident monitor, the operating system began to perform many of the functions, especially I/O, for which the programmer had previously been responsible.
- Without protection against these sorts of errors, either the computer must execute only one process at a time, or all output must be suspect.
- A properly designed operating system must ensure that an incorrect program cannot cause other programs to execute incorrectly.
- Many programming errors are detected by the hardware.
- These errors are normally handled by the operating system.

There are four components of hardware protection.

- **Dual-Mode Operation**
- **I/O Protection**
- **Memory Protection**
- **CPU Protection**

1. Dual-Mode Operation:

- Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly.
- Provide hardware support to differentiate between at least two modes of operations.

1. **User mode** – execution done on behalf of a user.

2. **Monitor mode** (also called kernel mode or system mode, supervisor mode, or privileged mode)

execution done on behalf of operating system.

- Mode bit added to computer hardware to indicate the current mode: monitor (0) or user (1)
- When an interrupt or fault occurs hardware switches to monitor mode.
- Privileged instructions can be issued only in monitor mode.
- The dual mode of operation provides us with the means for protecting the
- Operating system from errant users—and errant users from one another.
- We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**.
- The hardware allows privileged instructions to be executed only in kernel mode.

2. I/O Protection:

- All I/O instructions are privileged instructions.
- Must ensure that a user program could never gain control of the computer in monitor mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).

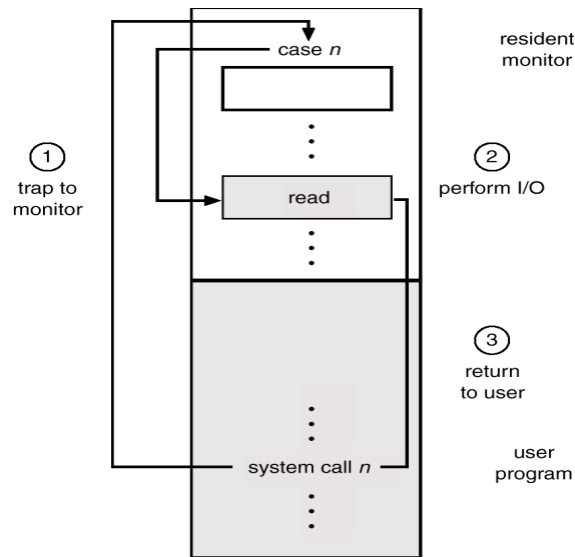


Figure: Use of a system call to perform I/O.

- A user program may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to give up the CPU.
- To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions.
- Thus, users cannot issue I/O instructions directly; they must do it through the operating system.
- For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode.
- If it could, I/O protection could be compromised.
- Consider a computer executing in user mode.
- It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt vector.
- If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program.
- The user program could gain control of the computer in monitor mode.
- Thus, to do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf shown in figure.
- The operating system, executing in monitor mode, checks that the request is valid, and does the I/O requested.
- The operating system then returns to the user.

3. Memory Protection:

- Even if the user did not gain unauthorized control of the computer, modifying the interrupt service routines would probably disrupt the proper operation of the computer system and of its spooling and buffering.
- Must provide memory protection at least for the interrupt vector and the interrupt service routines.
- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:
- We can provide this protection by using two registers, usually **a base and a limit**, as illustrated in figure.
- **Base register** – holds the smallest legal physical memory address.
 - **Limit register** – contains the size of the range.

- Memory outside the defined range is protected.
- For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 inclusive.
 - This protection is accomplished by the CPU hardware comparing every address generated in user mode with the registers.
 - Any attempt by a program executing in user mode to access monitor memory or other user's memory results in a trap to the monitor, which treats the attempt as a fatal error in figure.
 - This scheme prevents the user program from modifying the code or data structures of either the operating system or other users.

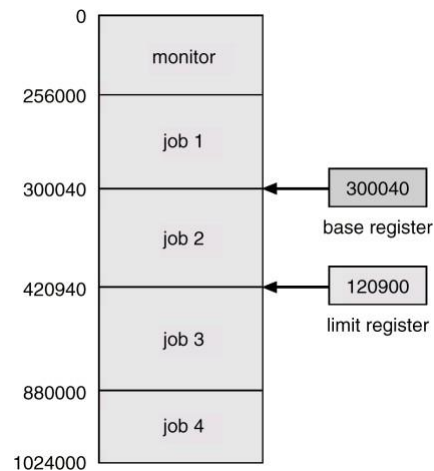


Figure: A base and a limit register define a logical address space.

- The base and limit registers can be loaded by only the operating system, which uses a special privileged instruction.
- Since privileged instructions can be executed in only monitor mode, and since only the operating system executes in monitor mode, only the operating system can load the base and limit registers.
- This scheme allows the monitor to change the value, but prevents user programs from changing the registers contents. The operating system, executing in monitor mode, is given unrestricted access to both monitor and users memory.

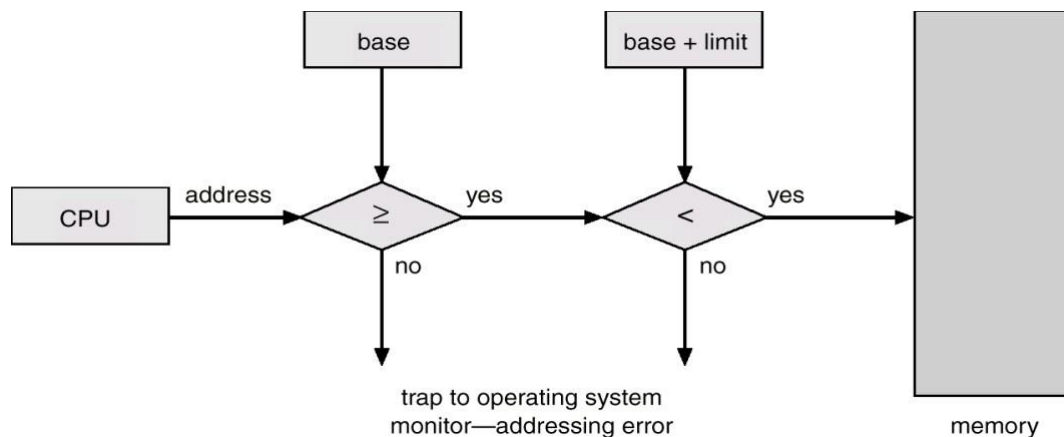


Figure: Hardware address protection with base and limit registers.

4. CPU PROTECTION:

- Timer – interrupts computer after specified period to ensure operating system maintains control.
 - A timer can be set to interrupt the computer after a specified period.
 - The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second).
 - A **variable timer** is generally implemented by a fixed-rate clock and a counter.
 - The operating system sets the counter.
 - Every time the clock ticks, the counter is decremented.
 - When the counter reaches 0, an interrupt occurs.
 - Timer commonly used to implement time sharing.
 - Time also used to compute the current time.
 - Load-timer is a privileged instruction.
 - Thus, we can use the timer to prevent a user program from running too long.
 - A simple technique is to initialize a counter with the amount of time that a program is allowed to run.
 - A program with a 7-minute time limit, for example, would have its counter initialized to 420.
 - Every second, the timer interrupts and the counter is decremented by 1.
 - As long as the counter is positive, control is returned to the user program.
 - When the counter becomes negative, the operating system terminates the program for exceeding the assigned time limit.
-
- A more common use of a timer is to implement time sharing.
 - In the most straightforward case, the timer could be set to interrupt every N millisecond, where N is the **time slice** that each user is allowed to execute before the next user gets control of the CPU.
 - The operating system is invoked at the end of each time slice to perform various housekeeping tasks, such as adding the value N to the record that specifies the amount of time the user program has executed.
 - The operating system also saves registers, internal variables, and buffers, and changes several other parameters to prepare for the next program to run. This procedure is known as a **context switch**.
-
- Another use of the timer is to compute the current time. A timer interrupt signals the passage of second period, allowing the operating system to compute the current time in reference to some initial time.

6. NETWORK STRUCTURE:

Network: A network consists of two or more computers that are linked in order to share resources (such as printers and CDs), exchange files, or allow electronic communications.

- The Computers on a network may be linked through cables, telephone lines, radio waves, satellite, or infrared light beams.

There are basically two types of Networks.

1. **Local-Area Network**
2. **Wide-Area Network**

Local-Area Network:

- A Local Area Network (LAN) Is a private network that connects computers and devices within a limited area like a residence, an office, a building or a campus.
- Data Transfer rate is generally high. They range from 100Mbps to 1000Mbps.
- In general a LAN uses only type of transmission medium, commonly category 5 coaxial cables.
- A LAN may be set up using wired or wireless connections.
- A LAN that is completely wireless is called Wireless LAN or WLAN.
- LAN uses either Ethernet or Token-ring technology.

- A Typical LAN may consists of a number of different computers, various shared peripheral devices(such as laser printers or magnetic –tape drives).

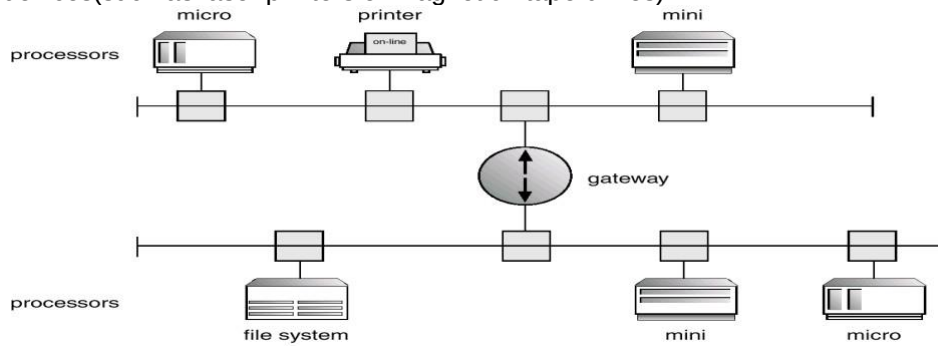


Figure: local-area network.

Wide-Area Network:

- A Wide area network (WAN) is a network that exists over a large-scale geographical area.
- WAN includes the technologies to transmit data, image, audio, and video information over long distances and among different LANs and MANs.
- The world's most popular WAN is the Internet.
- In, Wide Area Network, computers are connected through public networks such as the telephone system, fiber-optic cables, and satellite links or leased line.
- Typically they have low data transfer rate and high propagation delay, i.e. they have low communication speed.
- They generally have a higher bit error rate

Example of WAN

1. The Internet
2. 4G mobile broadband system
3. A network of bank cash dispensers

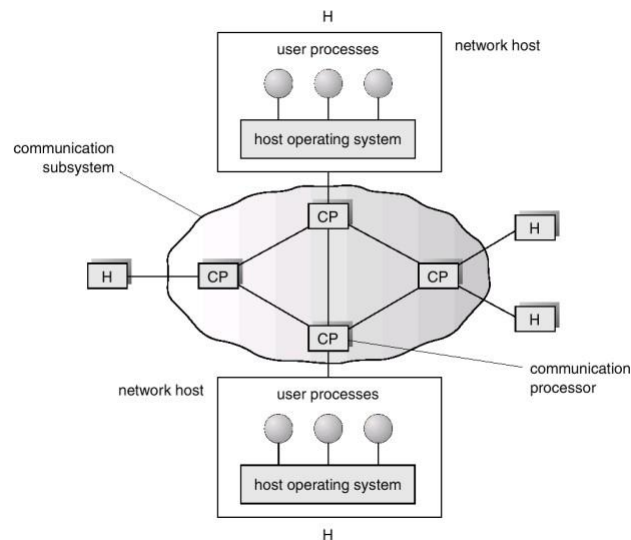


Figure: Wide-Area Network

Chapter 2

1. OPERATING SYSTEM COMPONENTS:

- Operating system provides the environment within which programs are executed.
- To construct such an environment, the system is partitioned into small modules with a well-defined interface.
- The design of a new operating system is a major task. It is very important that the goals of the system be defined before the design begins.
- The type of system desired is the foundation for choices between various algorithms and strategies that will be necessary.
- A system as large and complex as an operating system can only be created by partitioning it into smaller pieces.
- Each of these pieces should be a well defined portion of the system with carefully defined inputs, outputs, and function. Obviously, not all systems have the same structure.
- However, many modern operating systems share the system components outlined below:

1. **Process Management**
2. **Main-Memory Management**
3. **File Management**
4. **I/O-System Management**
5. **Secondary-Storage Management**
6. **Networking**
7. **Protection System**
8. **Command-Interpreter System**

Process Management:-

- The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes.
- A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process.
- In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created.
- For example, a process whose function is to display on the screen of a terminal the status of a file, say F1, will get as an input the name of the file F1 and execute the appropriate program to obtain the desired information.
- A process is the unit of work in a system.
- Such a system consists of a collection of processes, some of which are operating system processes, those that execute system code, and the rest being user processes, those that execute user code.
- All of those processes can potentially execute concurrently.
- The operating system is responsible for the following activities in connection with processes managed.
 - ❖ Creation and deletion of both user and system processes.
 - ❖ Suspending and resuming of processes.
 - ❖ Provision of mechanisms for process synchronization.
 - ❖ Providing mechanisms for process communication.
 - ❖ The provision of mechanisms for deadlock handling.

Main Memory Management:

- Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address.

- Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.
- In order for a program to be executed it must be mapped to absolute addresses and loaded in to memory.
- As the program executes, it accesses program instructions and data from memory by generating these absolute is declared available, and the next program may be loaded and executed.
- There are many different algorithms depends on the particular situation.
- Selection of a memory management scheme for a specific system depends upon many factor, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.
- ❖ The operating system is responsible for the following activities in connection with memory management.
 - ❖ Keep track of which parts of memory are currently being used and by whom.
 - ❖ Decide which processes are to be loaded into memory when memory space becomes available.
 - ❖ Allocating and deallocating memory space as needed.

File Management:

- File management is one of the most visible services of an operating system.
- Computers can store information in several different physical forms; magnetic tape, disk, and drum are the most common forms.
- Each of these devices has it own characteristics and physical organization.
- For convenient use of the computer system, the operating system provides a uniform logical view of information storage.
- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.
- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric.
- Files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.
- The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organized into directories to ease their use.
- Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.
- ❖ The operating system is responsible for the following activities in connection with file management:
 - ❖ Creating and deleting files
 - ❖ Creating and deleting directories
 - ❖ Supporting of primitives for manipulating files and directories
 - ❖ Mapping files onto secondary storage.
 - ❖ Backup of files on stable (non volatile) storage media.

I/O-System Management:

- One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user.

- **For example**, in Unix, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system.
- The I/O system consists of:
 - ❖ A buffer caching system
 - ❖ A general device driver code
 - ❖ Drivers for specific hardware devices.
 - ❖ Only the device driver knows the peculiarities of a specific device.

Secondary-Storage Management

- The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory , or **primary memory** during execution.
- Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory.
- Most modern computer systems use disks as the primary on-line storage of information, of both programs and data.
- Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing.
- Hence the proper management of disk storage is of central importance to a computer system.
- There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern.

The operating system is responsible for the following activities in connection with disk management

- ❖ Free space management
- ❖ Storage allocation
- ❖ Disk scheduling.

Networking:

- ❖ A **distributed system** is a collection of processors that do not share memory or a clock.
- ❖ Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high speed buses or telephone lines.
- ❖ Distributed systems vary in size and function.
- ❖ They may include **microprocessors, workstations, minicomputers, and large general purpose computer systems.**
- ❖ The processors in the system are connected through a communication network, which can be configured in the number of different ways.
- ❖ The network may be fully or partially connected.
- ❖ The communication network design must consider routing and connection strategies, and the problems of connection and security.
- ❖ A distributed system provides the user with access to the various resources the system maintains.
- ❖ Access to a shared resource allows computation speed-up, data availability, and reliability.
- ❖ The protocols that create a distributed system can have a great effects on that system's utility and popularity.
- ❖ The innovation of the **world wide web** was to create a new access method for information sharing.
- ❖ It improved on the existing **file-transfer protocol (FTP)** and **network file-system (NFS)** protocol by removing the need for a user to log in before she is allowed to use a remote resource.
- ❖ It defined a new protocol, hypertext transfer protocol (HTTP), for use in communication between a web server and a web browser.

- ❖ A web browser then just needs to send a request for information to a remote machine's web server, and the information (text, graphics, links to other information) is returned.

Protection System:

- The various processes in an operating system must be protected from each other's activities.
- For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorization from the operating system.
- **For example**, memory addressing hardware ensure that a process can only execute within its own address space.
- The timer ensures that no process can gain control of the CPU without relinquishing it.
- Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices.
- **Protection** is any mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.
- Protection can improve reliability by detecting latent errors at the interfaces between component subsystems.
- Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning.
- An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

Command-Interpreter System:

- ❖ One of the most important system for an operating system is its **command interpreter**. The command interpreter is the primary interface between the user and the rest of the system.
 - ❖ Some operating systems include the command interpreter in the **kernel**. Other operating system , such as a MS-dos and UNIX.
 - ❖ Many commands are given to the operating system by **control statements**. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed.
 - ❖ This program is variously called the **control card interpreter**, the **command line interpreter**, and is often known as the **shell**.
 - ❖ Its function is quite simple: get the next command statement, and execute it.
- The command statement themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

2. OPERATING SYSTEM SERVICES:

- An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs.
- These operating system services are provided for the convenience of the programmer, to make the programming task easier.

Operating –system services function:

User interface:

- Almost all operating systems have a **user interface (UI)**.
- This interface can take several forms. One is a **command-line interface (CLI)**, which uses text commands and a method.

- Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed.
- Most commonly/ a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

Program execution:

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

I/O Operations:

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

File-system manipulation:

- The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information.
- Finally, some programs include permissions management to allow or deny access to files or directories based on file ownership.

Communications:

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

Error detection:

- The operating system needs to be constantly needs to be aware of possible errors.
- Errors may occur in the CPU and memory hardware (**such as a memory error or a power failure**),
- In I/O devices (**such as a parity error on tape, a connection failure on a network, or lack of paper in the printer**),
- And in the user program (**such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time**).
- For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

Resource allocation:

- When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.
- Many different types of resources are managed by the operating system.
- Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code.
- There may also be routines to allocate printers, modems, USB storage drives, and other peripheral devices.

Accounting:

- We want to keep track of which users use how much and what kinds of computer resources.
- This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

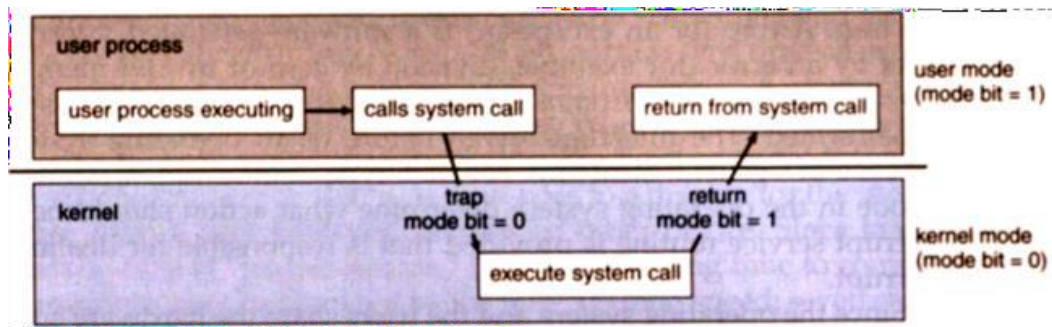
- Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.

Protection and security:

- **Protection** involves ensuring that all access to system resources is controlled.
- **Security** of the system from outsiders is also important.
- If a system is to be protected and secure, precautions must be instituted throughout it.

3. SYSTEM CALLS:

- **System calls** provide the interface between a process and the operating system. These calls are generally available as assembly-language instruction, and they are usually listed in the various manuals used by the assembly-language programmers.
- Several languages-such as C, C++.and Perl- have been defined to replace assembly language for systems programming.
- These languages allows system calls to be made directly.
- **For example:** UNIX system calls may be invoked directly from a C or C++ program.
- As an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file.
- The first input that the program will need is the names of the two files: the input file and the output file.



- Application developers often do not have direct access to the system calls, but can access them through an **Application Programming Interface (API)**.
- The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:
 - Portability: as long a system supports an API, any program using that API can compile and run.
 - Ease of Use: using the API can be significantly easier then using the actual system call.

System Call Parameters

Three general methods exist for passing parameters to the OS:

1. Parameters can be passed in registers.
2. When there are more parameters than registers, parameters can be stored in a block and the block address can be passed as a parameter to a register.
3. Parameters can also be pushed on or popped off the stack by the operating system.

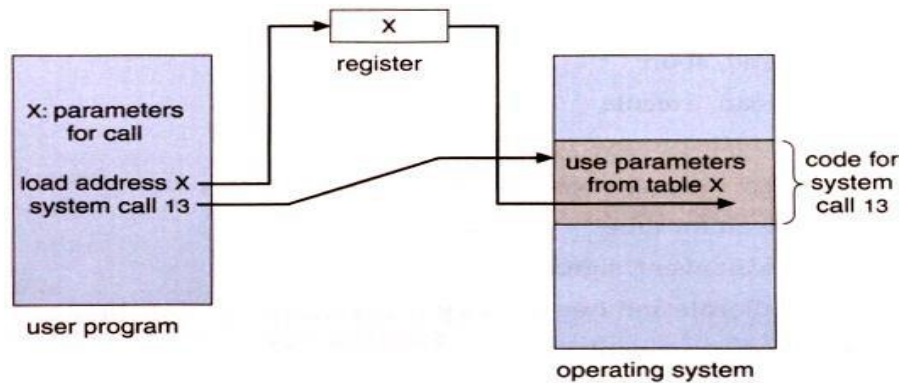
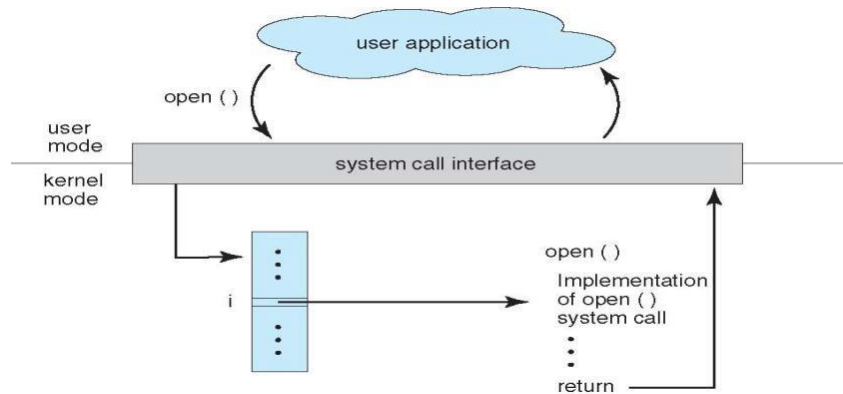
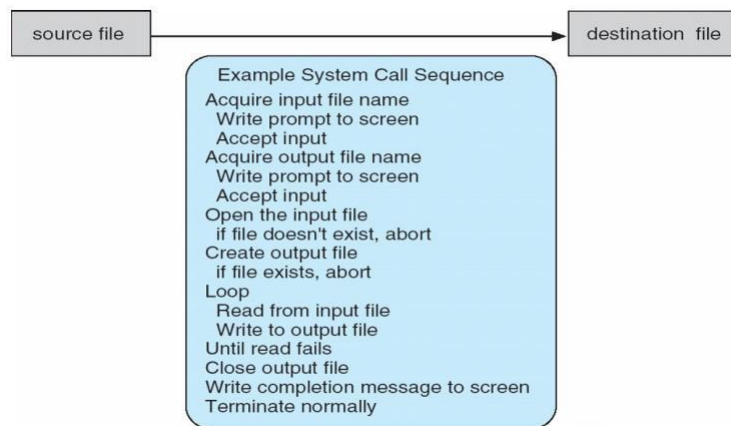


Figure: Passing of Parameters as a table



Example of system calls

- System call sequence to copy the contents of one file to another file



Types of System Calls

- System calls can be grouped roughly into five major categories:
 1. **Process control,**
 2. **File Management,**

3. **Device Management,**
4. **Information Maintenance,** and
5. **Communication.**

1. Process Control

- A running program needs to be able to stop execution either normally or abnormally.
- When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.
- If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.
- The dump is written to disk and may be examined by a **debugger** to determine the cause of the problem.
- Under either normal or abnormal circumstances, the operating system must transfer control to the command interpreter.

- **Process control**

- ✚ end, abort
- ✚ load, execute
- ✚ create process, terminate process
- ✚ get process attributes, set process attributes
- ✚ wait for time
- ✚ wait event, signal event
- ✚ allocate and free memory

- **File management**

- ✚ create file, delete file
- ✚ open, close
- ✚ read, write, reposition
- ✚ get file attributes, set file attributes

- **Device management**

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

- **Information maintenance**

- ✚ get time or date, set time or date
- ✚ get system data, set system data
- ✚ get process, file, or device attributes
- ✚ set process, file, or device attributes

- **Communications**

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

Figure : Types of system calls.

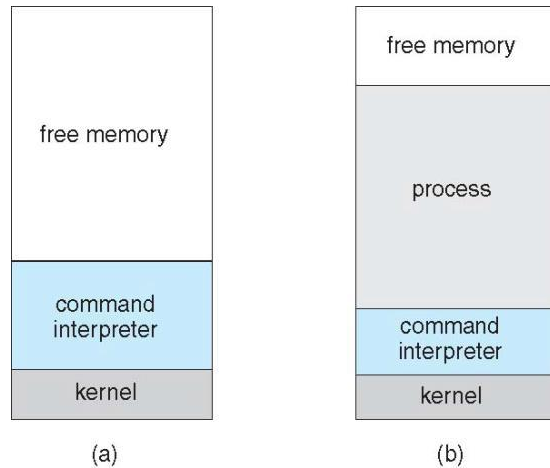


Figure: MS-DOS execution (a) At system startup (b) Running a program

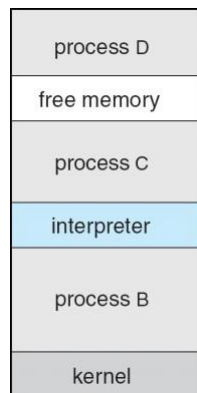


Figure: UNIX running multiple program

2.File Management

- We first need to be able to **create and delete files**. Either system call requires the name of the file and perhaps some of the file's attributes.
- Once the file is created, we need to open it and to use it. We may also **read, write, or reposition**.
- Some common system calls are *create, delete, read, write, reposition, or close*.
- File attributes include the file name, a file type, protection codes, accounting information.
- At least two system calls, get file attribute and set file attribute, are required for this function.
- Some operating system provide many more system calls.

3. Device Management

- A program, as it is running, may need additional resources to proceed. Additional resources may be more memory, tape drives, access to files.
- Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user program; otherwise, the program will have to wait until sufficient resources are available.
- These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

- User programs request the device, and when finished they release the device. Similar to files, we can read, write, and reposition the device.

4. Information Management

- Some system calls exist purely for transferring information between the user program and the operating system.
- For example, most system have a system call to return the current **time and date**.
- System calls may return information about the system such as:
 - The number of current users,
 - The version number of the operating system,
 - The amount of free memory or disk space
- The OS also keeps information about all its processes and provides system calls to report this information.

5. Communication

- There are two models of inter process communication, the **message-passing model** and the **shared memory model**.
- In the message-passing model, the communicating processes exchange messages with one another to transfer information.
- Messages can be exchanged between the processes either directly or indirectly through a common mailbox.
- Before communication can take place, a connection must be opened.
- The name of the other communicator must be known, be it another process on the same system or a process on another computer connected by a communications network.
- Each computer in a network has a *host name* by which it is commonly known. A host also has a network identifier, such as an IP address.
- The get host id and get processed system calls do this translation.
- The identifiers are then passed to the general purpose **open and close** calls provided by the file system or to specific **open connection and close connection** system calls, depending on the system's model of communication.

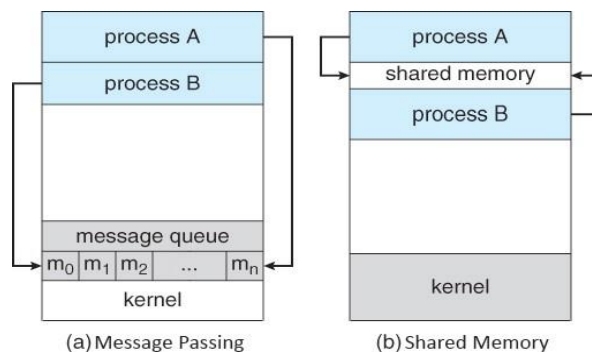


Figure: Communications models. (a) Msg passing. (b) Shared memory.

- Message-passing uses a common mailbox to pass messages between processes.

- Both of the models just discussed are common in operating systems, and most systems implement both.
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- It is also easier to implement than is shared memory for inter computer communication.

4. System Programs:

- Another aspect of a modern system is the collection of system programs.

At the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs.

- System programs provide a convenient environment for program development and execution.
- Some of them are simply user interfaces to system calls; others are considerably more complex.
- System programs provide a convenient environment for program development and execution.

The can be divided into:

- **File manipulation**
- **Status information**
- **File modification**
- **Programming language support**
- **Program loading and execution**
- **Communications**

- **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
- Others are more complex, providing detailed performance, logging, and debugging information.
- Typically, these programs format and print the output to the terminal or other output devices or files.
- Some systems also support a registry, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed.
- The system may provide absolute loaders, relocate able loaders, linkage editors, and overlay loaders.
- Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.

Types of System Calls	Windows	Linux
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Management	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Management	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
	GetCurrentProcessID()	getpid()
Information Maintenance	SetTimer()	alarm()
	Sleep()	sleep()
	CreatePipe()	pipe()
Communication	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()

- They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.
- In addition to systems programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations.
- Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games. These programs are known as **system utilities** or **application programs**.

5. SYSTEM STRUCTURE:

- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.
- A common approach is to partition the task into small components rather than have one monolithic system.
- Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.
- MS-DOS –written to provide the most functionality in the least space
 - Not divided into modules
 - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.

1. Simple structure:

- Many commercial systems do not have well-defined structures.
- Frequently, such operating systems started **as small, simple, and limited systems** and then grew beyond their original scope.
- **MS-DOS is an example of such a system.**
- It was written to provide the most functionality in the least space, so it was not divided into modules carefully. Figure shows in its **structure**.
- Another example of limited structuring is the original UNIX operating system. **UNIX** is another system that initially was limited by hardware functionality.
- It consists of two separable parts: **the kernel** and **the system programs**.
- The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.
- We can view the traditional UNIX operating system as being layered, as shown in Figure.
- Everything below the system call interface and above the physical hardware is the kernel.
- The kernel provides the **file system, CPU scheduling, memory management**, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level.
- This monolithic structure was difficult to implement and maintain.
- With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS or UNIX systems.
- The operating system can then retain much greater control over the computer and over the applications that make use of that computer.
- Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under the top-down approach, the overall functionality and features are determined and are separated into components.

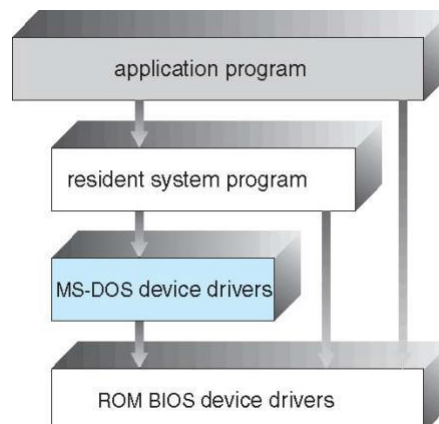


FIGURE: MS-DOS LAYERED STRUCTURE

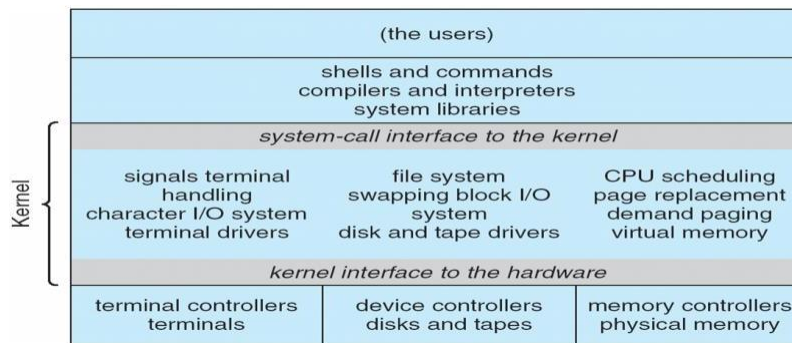


Figure: UNIX system structure.

2. Layered Approach:

- A system can be made modular in many ways.
- One method is the **layered approach**, in which the operating system is broken up into a number of layers (levels).
- The **bottom layer (layer 0)** is the hardware; the **highest (layer N)** is the user interface. This layering structure is depicted in Figure.
- An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.
- A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.
- The **main advantage of the layered approach is simplicity of construction and debugging.**
- The layers are selected so that each uses functions (operations) and services of only lower-level layers.
- This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, definition, it uses only the basic hardware (which is assumed correct) to implement its functions.

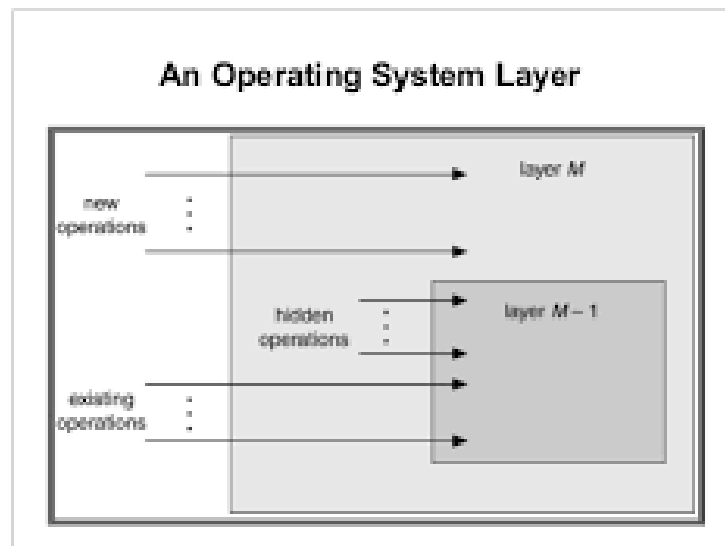


Figure : An Operating-System layer

- Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged.
- Thus, the design and implementation of the system is simplified. Each layer is implemented with only those operations provided by lower-level layers.
- A layer does not need to know how these operations are implemented; it needs to know only what these operations do.
- Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.

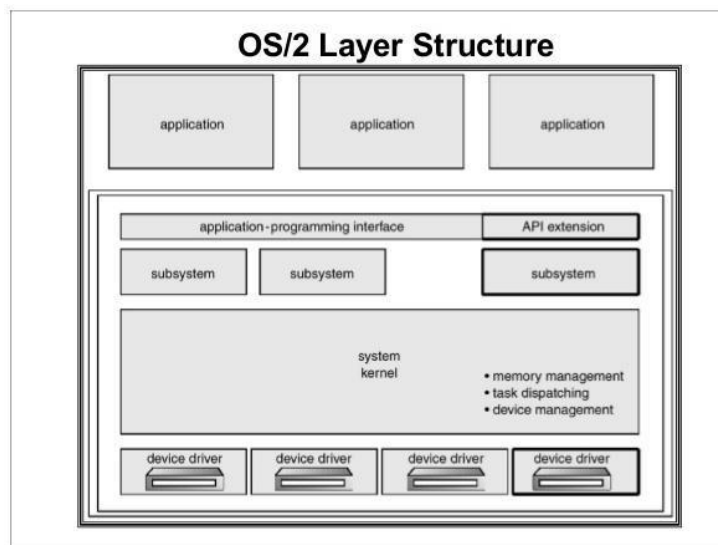


Figure : OS/2 layer structure.

- For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than the memory-management.
- The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time.
- A final problem with layered implementations is that they tend to be less efficient than other types.
- For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware.
- At each layer, the parameters may be modified, data may need to be passed, and so on.
- Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a non layered system.
- These limitations have caused a small backlash against layering in recent years.
- Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction.

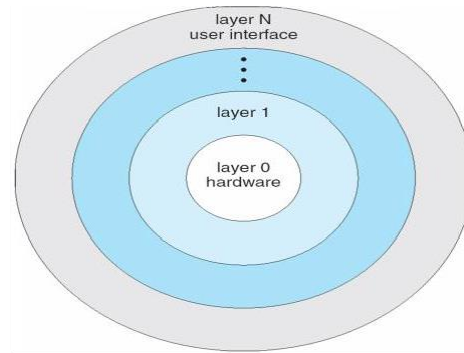


Figure: A layer operating system

3. Microkernel:

- As the UNIX operating system expanded, the kernel became large and difficult to manage.
- In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach.
- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.
- The result is a smaller kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, micro kernels provide minimal process and memory management, in addition to a communication facility.
- The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.
- Communication is provided by *message passing*.
- For example, if the client program wishes to access a file, it must interact with the file server.
- The client program and the service never interact directly.
- Rather, they communicate indirectly by exchanging messages with the microkernel.
- The benefit of the microkernel approach is ease of extending the operating system.
- The resulting operating system is easier to port from one hardware design to another. The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes.
- If a service fails, the rest of the operating system remains untouched.
- QNX is a real-time operating system that is also based on the microkernel design.
- The QNX microkernel provides services for message passing and process scheduling.
- It also handles low-level network communication and hardware interrupts.
- All other services in QNX are provided by standard processes that run outside the kernel in user mode.
- Unfortunately, micro kernels can suffer from performance decreases due to increased system function overhead.
- Consider the history of Windows NT.
- The first release had a layered microkernel organization.
- However, this version delivered low performance compared with that of Windows 95.
- Windows NT 4.0 partially redressed the performance problem by moving layers from user space to kernel space and integrating them more closely.
- By the time Windows XP was designed, its architecture was more monolithic than microkernel.

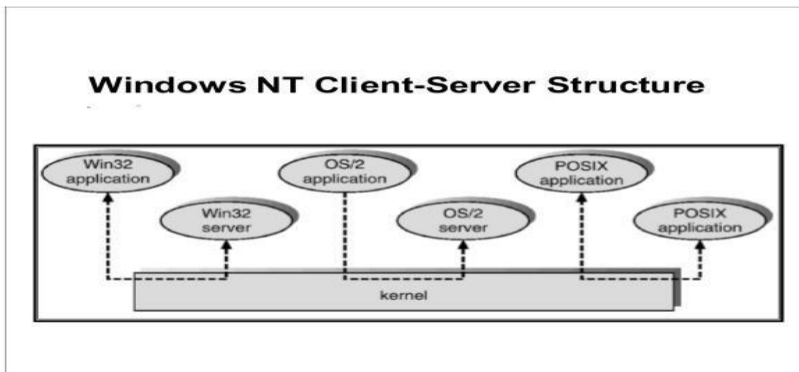


Figure: Windows NT client-server structure.

Microkernel System Structure

- Moves as much from the kernel into “user” space
- Communication takes place between user modules using message passing

Benefits:

- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

6. VIRTUAL MACHINES:

- A **virtual machine** takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- A virtual machine provides an interface identical to the underlying bare hardware.
- The operating system **host** creates the illusion that a process has its own processor and (virtual memory).
- Each **guest** provided with a (virtual) copy of underlying computer.
- The layered approach described is taken to its logical conclusion in the concept of a **virtual machine**.
- The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.
- A major difficulty with the virtual-machine approach involves disk systems.
- Suppose that the physical machine has three disk drives but wants to support seven virtual machines.
- Clearly, it cannot allocate a disk drive to each virtual machine, because the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling.
- The solution is to provide virtual disks—termed *minidisks* in IBM's VM operating system—that are identical in all respects except size.
- The system implements each minidisk by allocating as many tracks on the physical disks as the minidisk needs.
- Obviously, the sum of the sizes of all minidisks must be smaller than the size of the physical disk space available.

Virtual Machines History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware,

- Protect from each other,
- Some sharing of file can be permitted, controlled, Commutate with each other, other physical systems via networking Useful for development,
- Testing **Consolidation** of many low-resource use systems onto fewer busier systems.
- “Open Virtual Machine Format”, standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms.

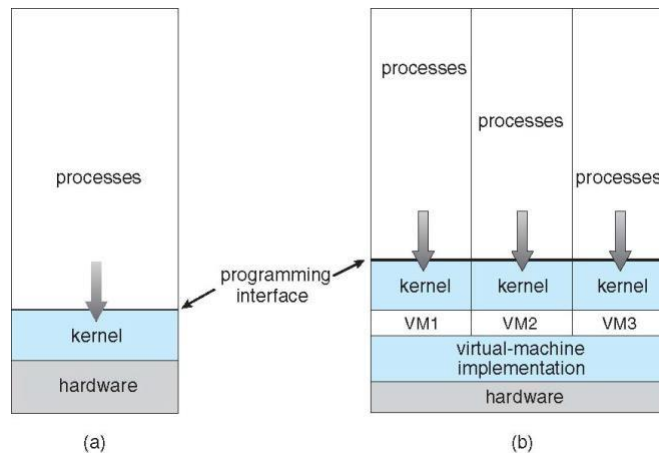


FIGURE: System models (a) Non virtual machine (b) virtual machine.

1. Implementation:

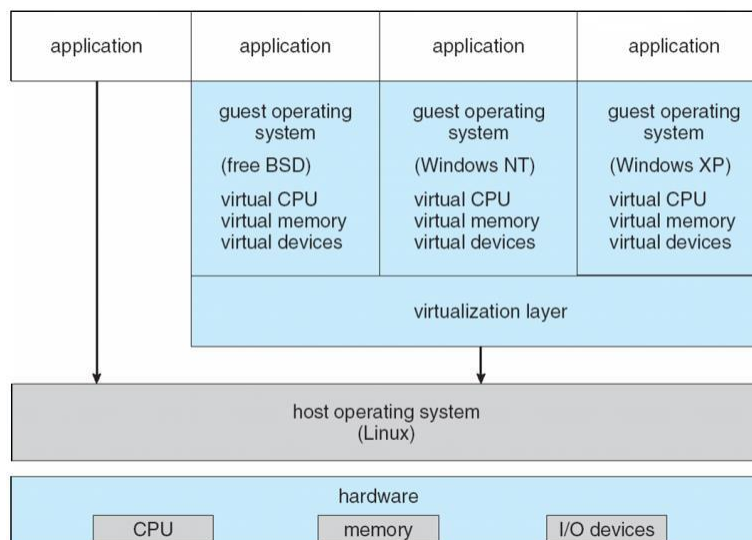
- Although the virtual-machine concept is useful, it is difficult to implement.
- Much work is required to provide an *exact* duplicate of the underlying machine.
- Remember that the underlying machine has two modes: **user mode and kernel mode**.
- The virtual-machine software can run in kernel mode, since it is the operating system. The virtual machine itself can execute in only user mode.
- Just as the physical machine has two modes, however, so must the virtual machine.
- Consequently, we must have a virtual user mode and a virtual kernel mode, both of which run in a physical user mode.
- Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.
- It can then restart the virtual machine, noting that it is now in virtual kernel mode.
- The major difference, of course, is time.
- Whereas the real I/O might have taken 100 milliseconds, the virtual I/O might take less time (because it is spooled) or more time (because it is interpreted).
- In addition, the CPU is being multi programmed among many virtual machines, further slowing down the virtual machines in unpredictable ways.
- In the extreme case, it may be necessary to simulate all instructions to provide a true virtual machine.
- VM works for IBM machines because normal instructions for the virtual machines can execute directly on the hardware.
- Only the privileged instructions (needed mainly for I/O) must be simulated and hence execute more slowly.

2. Benefits:

- There are two primary advantages to using virtual machines.
- First, by completely protecting system resources, the virtual machine provides a robust level of security.
- Second, the virtual machine allows system development to be done without troublesome normal system operation.
- Each virtual machine is completely isolated from all other virtual machines, so we have no security problem as the various system resources are completely protected.
- At the same time, however, there is no direct sharing of resources. Two approaches to provide sharing have been implemented
- First, it is possible to share a minidisk and thus to share files. This scheme is modeled after a physical shared disk but is implemented by software.
- Second, it is possible to define a network of virtual machines, each of which can send information over the virtual communications network.
- Again, the network is modeled after physical communication networks but is implemented in software.
- Operating systems are large and complex programs, and it is difficult to be sure that a change in one part will not cause obscure bugs in some other part.
- The operating system, however, runs on and controls the entire machine.
- Therefore, the current system must be stopped and taken out of use while changes are made and tested.
- This period is commonly called **system development time**. Since it makes the system unavailable to users, system development time is often scheduled late at night or on weekends, when system load is low.
- A virtual-machine system can eliminate much of this problem. System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine.
- Normal system operation seldom needs to be disrupted for system development.

Examples

- Despite the advantages of virtual machines, they received little attention for a number of years after they were first developed.



3. The Java Virtual Machine:

- Java is a very popular object-oriented programming language introduced by Sun Microsystems in 1995.
- In addition to a language specification and a large API library, Java also provides a specification for a **Java virtual machine—or JVM**.
- Java objects are specified with the **class** construct; a Java program consists of one or more classes.
- For each Java class, the compiler produces an architecture-neutral **byte code** output (.class) file that will run on any implementation of the JVM.
- The JVM is a specification for an abstract computer. It consists of a **class loader**, a **class verifier**, and a Java interpreter that executes the architecture-neutral byte codes, as diagrammed shown in Figure.
- The class loader loads the compiled **.class** files from both the Java program and the Java API for execution by the Java interpreter.
- After a class is loaded, the verifier checks that the **.class** file is valid Java byte code and does not overflow or underflow the stack.
- It also ensures that the byte code does not perform pointer arithmetic, which could provide illegal memory access.
- If the class passes verification, it is run by the Java interpreter.
- The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system.
- Much research focuses on garbage collection algorithms for increasing the performance of Java programs in the virtual machine.
- The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a web browser.
- Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs. If the JVM is implemented in software, the Java interpreter interprets the byte code operations one at a time.
- A faster software technique is to use a **just-in-time (JIT)** compiler.
- Here, the first time a Java method is invoked, the byte codes for the method are turned into native machine language for the host system.
- These operations are then cached so that subsequent invocations of a method are performed using the native machine instructions and the byte code operations need not be interpreted all over again.
- A technique that is potentially even faster is to run the JVM in hardware on a special Java chip that executes the Java byte code operations as native code, thus bypassing the need for either a software interpreter or a just-in-time compiler.

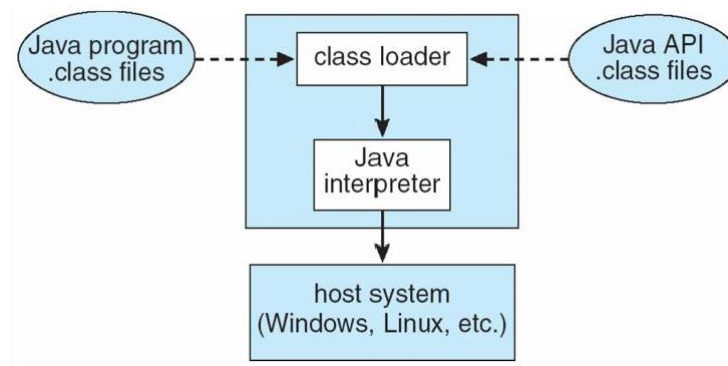


Figure: The Java Virtual Machine

- Java takes advantages of the complete environment that a virtual machine implements.
- Its virtual-machine design provides a secure, efficient, object-orient, portable, and architecture-neutral platform on which to run java programs.

7. SYSTEM DESIGN AND IMPLEMENTATION:

1. Design Goals:

- The first problem in designing a system is to define goals and specifications. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: It is quite complicated to define all the goals and specifications of the operating system while designing it.
- The design changes depending on the type of the operating system ie, if it is **batch system, time shared system, single user system, multi user system, distributed system etc.**
- At the highest level, system design is dominated by the choice of hardware and system type.
- Beyond this level, the requirements can be divided into two groups: **user goals, and system goals.**
- **User goals:**
- **User goals** include convenience, reliability, security, and speed.
- The operating system should be convenient, easy to use, reliable, safe and fast according to the users.
- However, these specifications are not very useful as there is no set method to achieve these goals.
- **System goals:**
- **System goals** include ease of design, implementation, maintenance, flexibility, and efficiency.
- The operating system should be easy to design, implement and maintain. These are specifications required by those who create, maintain and operate the operating system.
- But there is not specific method to achieve these goals as well.

2. Mechanisms and Policies

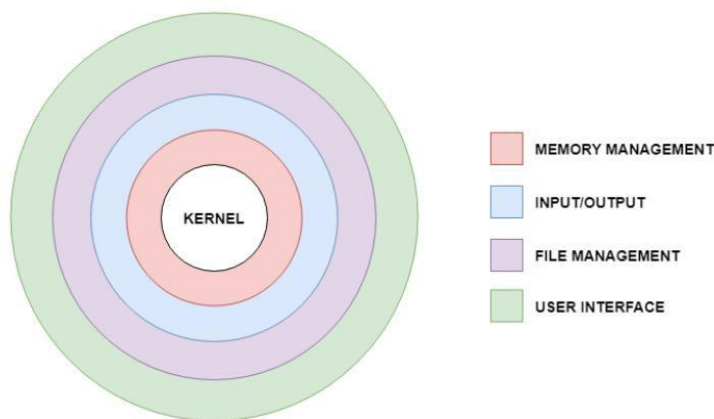
- There is no specific way to design an operating system as it is a highly creative task.
- However, there are general software principles that are applicable to all operating systems.
- A subtle difference between mechanism and policy is that mechanism shows how to do something and policy shows what to do.
- Policies may change over time and this would lead to changes in mechanism.
- So, it is better to have a general mechanism that would require few changes even when a policy change occurs.
- For example - If the mechanism and policy are independent, then few changes are required in mechanism if policy changes.

- If a policy favors I/O intensive processes over CPU intensive processes, then a policy change to preference of CPU intensive processes will not change the mechanism.

3. Implementation

- At first, operating systems were written in assembly, but now C/C++ is the language commonly used
- Small blocks of assembly code are still needed, especially related to some low level I/O functions in device drivers, turning interrupts on and off and the **Test and Set Instruction for Synchronization Facilities**.
- Using higher level languages allows the code to be written faster. It also makes the OS much easier to port to different hardware platforms.
- An operating system is a construct that allows the user application programs to interact with the system hardware.
- Operating system by itself does not provide any function but it provides an atmosphere in which different applications and programs can do useful work.
- There are many problems that can occur while designing and implementing an operating system.
- These are covered in operating system design and implementation.

Result This program has been successfully executed.



Layered Operating System Design

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

➤ **Advantages of Higher Level Language**

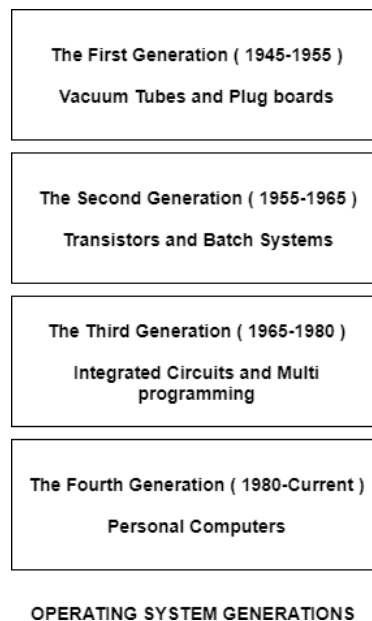
- There are multiple advantages to implementing an operating system using a higher level language such as: the code is written more fast, it is compact and also easier to debug and understand.
- Also, the operating system can be easily moved from one hardware to another if it is written in a high level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

➤ **Disadvantages of Higher Level Language**

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

8. System Generation:

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



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

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

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

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

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

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

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

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

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

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

The Fourth Generation (1980 - Present): Personal Computers

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

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

Chapter-3

PROCESSES

Topics:

- **Process Concepts**
- **Process Scheduling**
- **Operation on Processes**
- **Inter process Communication**
- **Communication in client server System**

❖ Introduction to Processes:

- Early computer systems allowed only one program to be executed at a time.
- Current-day computer systems allow multiple programs to be loaded into memory and executed concurrently.
- A process is the unit of work in a modern time-sharing system.
- A system therefore consists of a collection of processes: **operating system processes executing system code** and **user processes executing user code**.

Differences between Process and Program:

Process	Program
The process is basically an instance of the computer program that is being executed.	A Program is basically a collection of instructions that mainly performs a specific task when executed by the computer.
A process has a shorter lifetime .	A Program has a longer lifetime .
A Process requires resources such as memory,	A Program is stored by hard-disk and

Process	Program
CPU, Input-Output devices.	does not require any resources.
A process has a dynamic instance of code and data	A Program has static code and static data.
Basically, a process is the running instance of the code.	On the other hand, the program is the executable code .

❖ Process Concepts:

- A batch system executes jobs, whereas a time-shared system has user programs, or tasks.
- Even on a single-user system such as Microsoft Windows, a user may be able to **run several programs** at one time: **a word processor, a web browser, and an e-mail package.**
- Even if the user can execute only one program at a time, the operating system may need to support its own internal programmed activities, such as memory management.
- In many respects, all these activities are similar, so we call all of them **processes.**
 1. The process
 2. Process state
 3. Process Control Block
 4. Thread

1.The Process:

- A process is a program in execution. A process is more than the program code, which is sometimes known as the **text section.**

- It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers.
- A process generally includes the process **stack**, which contains temporary data (such as method parameters return addresses, and local variables), and a **data section**, which contains global variables.

2.Process State:

- As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.
- Each process may be in one of the following states:
 - ❖ **New**: The process is being created.
 - ❖ **Running**: Instructions are being executed.
 - ❖ **Waiting**: The process is waiting for some event to occur.
 - ❖ **Ready**: The process is waiting to be assigned to a processor.
 - ❖ **Terminated**: The process has finished execution.

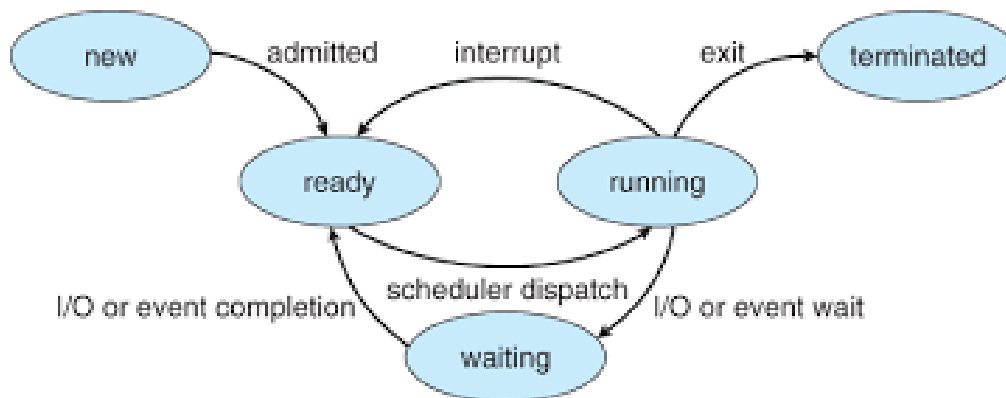


Figure: Diagram of process state.

- These state names are arbitrary, and they vary across operating system.
- Only one process can be running on any processor at any instant, although many processes may be **ready and waiting**. The state diagram corresponding to these is presented in above figure.

3.Process Control Block

- Each process is represented in the operating system by a **Process Control Block(PCB)** – also called a task control block. A PCB is shown in figure.
- It contains many pieces of information associated with a specific process, including these:
- **Process state**: The state may be new, ready, running, waiting, terminated and so on.

- **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- **CPU register:** The register vary in number and type, depending on the computer architecture. They include **accumulators, index registers, stack pointers, and general-purpose registers**, plus any condition-code information.
- **CPU- Scheduling Information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-Management Information:** This information may include such information as the value of the base and limit registers, the page tables or the segment tables, depending on the memory system used by the operating system.

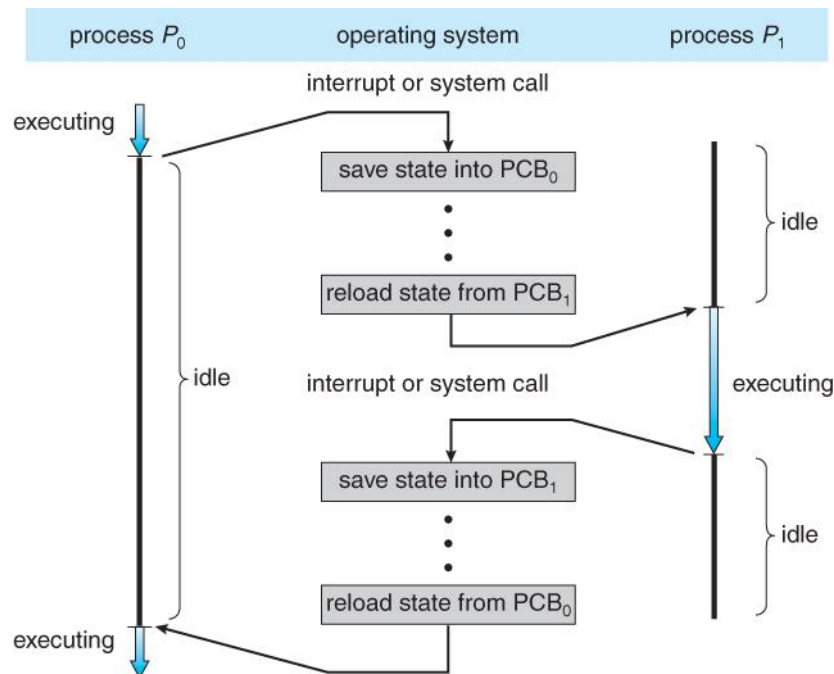


Diagram showing CPU switch from process to process

- **Accounting information:** This information includes the amount of **CPU and real time used, time limits, account numbers, job or process numbers**, and so on.
- **I/O status information:** The information include the list of I/O devices allocated to this process, a list of open files, and so on.

Threads:

- 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** is also called a lightweight process.

❖ Process Scheduling:

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 consists of all the processes in the system.
- **Ready queue** – The processes that are residing in main memory, ready and waiting to execute are kept on a list called **ready queue**. A new process is always put in this queue.
- **Device queues** – The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O **device queue**.

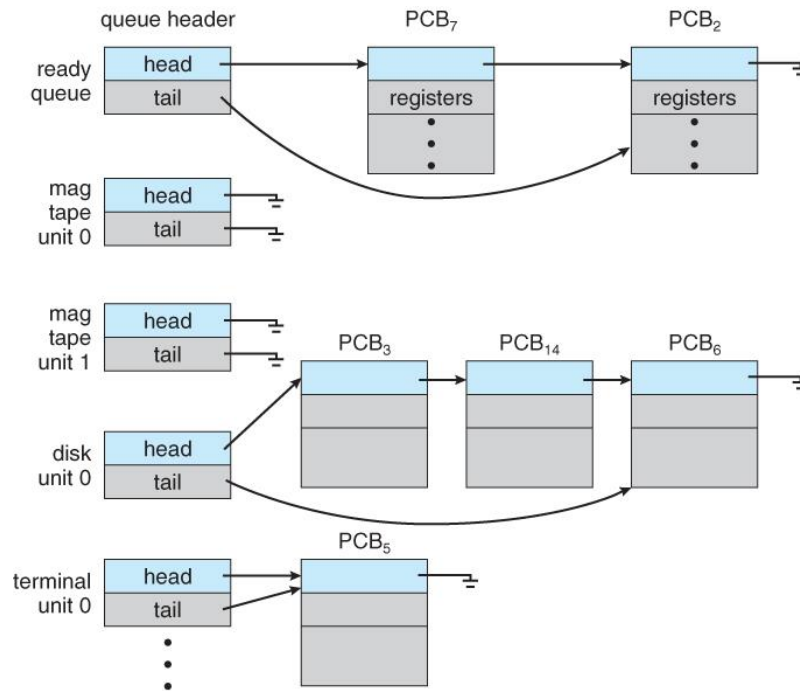
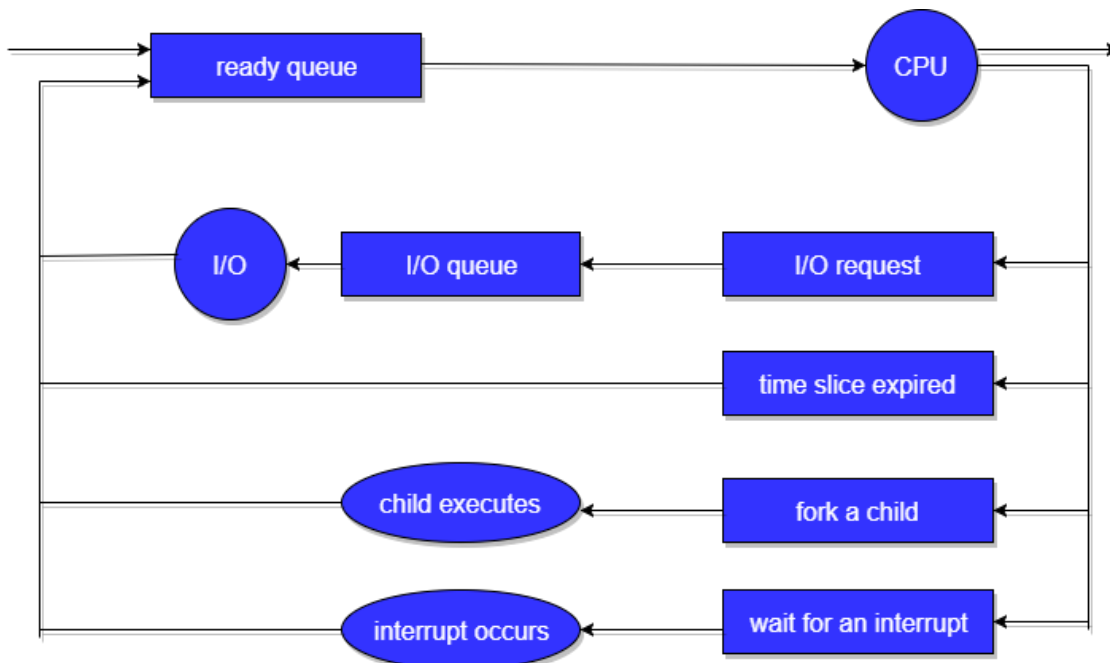


Figure: The ready queue and various I/O device queue

A common representation of process scheduling is a queueing diagram, such as that in figure. Each rectangular box represents a queue.



Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

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

- The process could issue an I/O request, and then be placed in an I/O queue.
- The process could create a new process and wait for its termination.
- The process could be removed from the CPU, as a result of an interrupt, and be back in the ready queue.

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

Long Term Scheduler

- It is also called a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Processes load into the memory for CPU scheduling.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

- It is also called as **CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

- Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

- Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in-charge of handling the swapped out-processes.
- A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion.
- In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage.
- This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

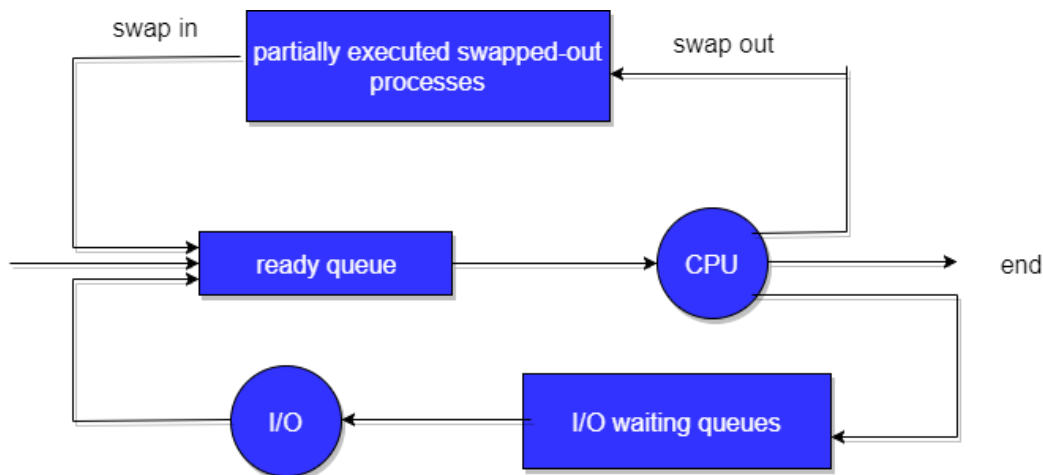


Figure: Addition of medium term scheduling to the queueing diagram

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 sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Context Switch

1. Switching the CPU to another process requires **saving** the state of the old process and **loading** the saved state for the new process. This task is known as a **Context Switch**.
2. The **context** of a process is represented in the **Process Control Block(PCB)** of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
3. Context switch time is **pure overhead**, because the **system does no useful work while switching**. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions(such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds.
4. Context Switching has become such a performance **bottleneck** that programmers are using new structures(threads) to avoid it whenever and wherever possible.

❖ Operations in Processes:

The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism for **process creation and termination**.

Process Creation:

- A process may create several new processes, via a create-process system call, during the course of execution.
- The creating process is called a parent process, whereas the new processes are called the children of that process.
- Each of these new processes may in turn create other processes, forming a tree of processes in **figure**.

When the process creates a new process, two possibilities exist in terms of execution.

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

There are also two possibilities in terms of the address space of the new process.

- The child process is a duplicate of the parent process.
- The child process has a program loaded into it.

Process Termination:

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

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

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

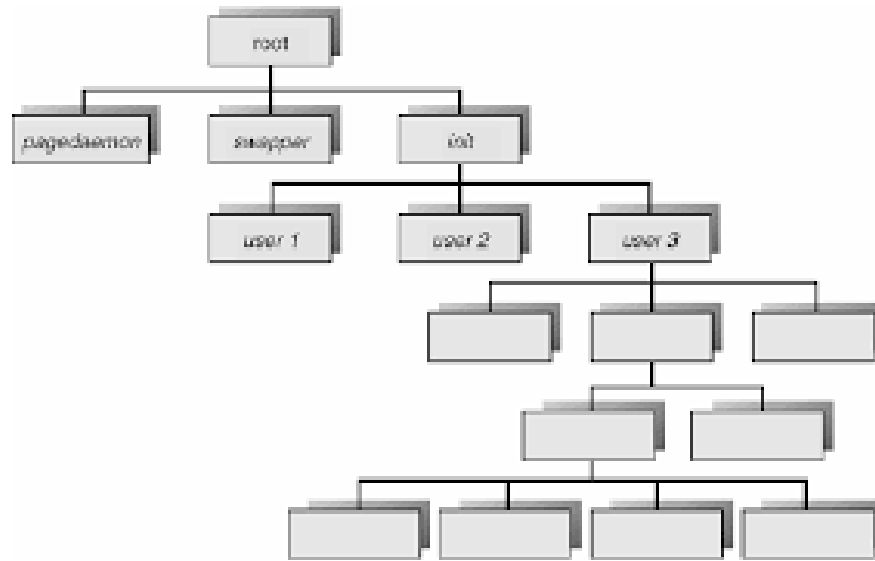


Figure: A tree of processes on a typical UNIX system

❖ INTERPROCESS COMMUNICATION:

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the **transferring of data from one process to another**.



A diagram that illustrates interprocess communication is as following Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other.

Message Passing System:

In this article, you will learn another popular method of interprocess communication called message passing. You will learn different techniques and characteristics of message passing.

The cooperating processes can communicate with the help of a message passing facility. There are different ways communication happens:

- Between same threads of a process.

- Between processes on same node or computer.
- Between two processes on different nodes or computers.
Example: chat system over Internet

A message-passing system has atleast two primitive operations:

1. send(message)
2. receive(message)

The messages could be of a fixed size or a variable size. If system uses fixed size messages, then system level implementation is easy. If chosen variable size messages, the system level implementation is difficult, but the programming level task is easy.

If P and Q process wants to communicate, then a communication link must exist between them. There are several ways to implement physical link(shared memory, hardware, or network), but we are interested in logical implementation of a link. Here are different ways to communicate using message passing:

1. Direct or Indirect communication
2. Synchronous or asynchronous communication
3. Automatic or explicit buffering.

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

Direct Communication:

Under direct communication, processes must explicitly name the sender or receiver in the communication.

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

send(P, message) – send a message to process P

receive(Q, message) – receive a message from process Q

The communication link in direct communication has the following properties:

- The link is established automatically. Processes need each other's identity to send message.
- A link is associated with exactly two processes.
- Between two processes, there is only one link.

There is a symmetric in addressing. In asymmetrical addressing, the sending process needs to address the receiver, but the recipient does not need to address the sender.

Direct Messaging:

send(p, message)- send message to process P

received(id, message) – receive message from any process. Id is replaced with name of the sender process.

- The disadvantages in symmetric and asymmetric schemes is in changing the identifier of a process.
- We need to change all the other process definition and references to the old identifier and replace with the new one.

Indirect Communication:

- In indirect communication, message are sent and received from **mailboxes or ports**. The processes can place messages into a mailbox or remove messages from them. The mailbox has a unique identification.

Two process can communicate only if they have a shared mailbox.

send(A, message) – send a message to mailbox A

receive (A, message) – receive a message from mailbox A

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

1. A link is established between a pair of processes only if both have a same shared mailbox.
2. A link may be associated with more than two processes.
3. There may be different links, with each link corresponding to one mailbox, between pair of communicating processes.

Example: P1, P2 and P3 all share mailbox A. P1 sends message to A, while P2 and P3 execute a receive() from A. Which process will receive the message? The answer depends on one of the chosen method given below.

- Allow link to associated with at – most 2 processes.
- Allow at most 1 process to execute receive() operation.
- Choose randomly at- most one process to receive message or choose an algorithm to receive message such as round – robin(each process take turn to receive message).

The mailbox may be owned either by a process or by the OS.

Indirect Messaging:

If process owns (mailbox part of process address space) then:

Distinguish between owner(can only receive through own mailbox).

The user (can only send message to the mailbox).

When process with mailbox terminates, mailbox disappears, and all other processes should be notified.

If OS owns the mailbox, then

- It is independence process, not attached to a process.
- Then the OS must allows the process to:
 1. Create a mailbox
 2. Send and receive messages through mailbox.

3. Delete the mailbox.
- Process becomes the owner of new mailbox by default.
- Owner process can only receive messages through this mailbox.
- Ownership and receiving privileges can be passed using system calls to other processes. This will result in multiple receivers for each mailbox.

Synchronous and Asynchronous Communication:

Communication happens using send() and receive(). There are many options for these two primitives. Message passing may be blocking or non blocking also known as synchronous and asynchronous.

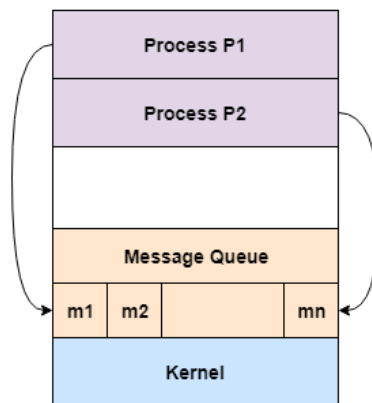
- **Blocking send**- sending is blocked, until a message is received by receiving process or mailbox.
- **Non-blocking send** – sending process sends the message and resumes operation.
- **Blocking receive** – receiver blocks until a message is available.
- **Non –blocking receive** – the receiver retrieves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

Automatic and Explicit Buffering

The messages exchanged between communicating processes resides in a temporary queue. The queue can be implemented in three ways:

1. Zero capacity
 2. Bounded capacity
 3. Unbounded capacity
- **Zero capacity** – with zero capacity the link cannot have messages waiting. Blocking send is the correct option.
 - **Bounded capacity** – the queue is of finite length n. If queue not full, sender can continue to send messages. If full, blocking send.
 - **Unbounded capacity** – Infinite queue length; any number of messages wait and sender never block

A diagram that demonstrates message passing model of process communication is given as follows –



Message Passing Model

In the above diagram, both the processes P1 and P2 can access the message queue and store and retrieve data.

Advantages of Message Passing Model

Some of the advantages of message passing model are given as follows –

- The message passing model is much easier to implement than the shared memory model.
- It is easier to build parallel hardware using message passing model as it is quite tolerant of higher communication latencies.

Disadvantage of Message Passing Model

The message passing model has slower communication than the shared memory model because the connection setup takes time.

❖ Communication in client server system:

Sockets:

A **socket** is one endpoint of a two way communication link between two programs running on the network. The **socket** mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication take place.

- A socket is made up of an IP address concatenated with a port number. In general, socket use a client-server architecture.
- The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection.

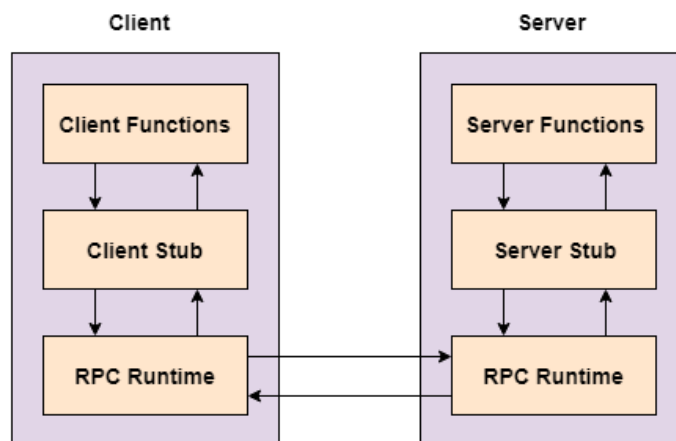
REMOTE PROCEDURE CALL (RPC):

- A remote procedure call is an interprocess communication technique that are used for client-server based applications. It is also known as a subroutine call or a function call.
- A client has a request message that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server.
- When the server receives the request, it sends the required response back to the client. The client is blocked while the server is processing the call and only resumed execution after the server is finished.

The sequence of events in a remote procedure call are given as follows –

- The client stub is called by the client.
- The client stub makes a system call to send the message to the server and puts the parameters in the message.
- The message is sent from the client to the server by the client's operating system.
- The message is passed to the server stub by the server operating system.
- The parameters are removed from the message by the server stub.
- Then, the server procedure is called by the server stub.

A diagram that demonstrates this is as follows –



Advantages of Remote Procedure Call

Some of the advantages of RPC are as follows –

- Remote procedure calls support process oriented and thread oriented models.
- The internal message passing mechanism of RPC is hidden from the user.
- The effort to re-write and re-develop the code is minimum in remote procedure calls.
- Remote procedure calls can be used in distributed environment as well as the local environment.
- Many of the protocol layers are omitted by RPC to improve performance.

Disadvantages of Remote Procedure Call

Some of the disadvantages of RPC are as follows

- The remote procedure call is a concept that can be implemented in different ways. It is not a standard.
 - There is no flexibility in RPC for hardware architecture. It is only interaction based.
 - There is an increase in costs because of remote procedure call.
- A diagram that illustrates remote procedure calls is given as follows:

Remote Method Invocation(RMI)

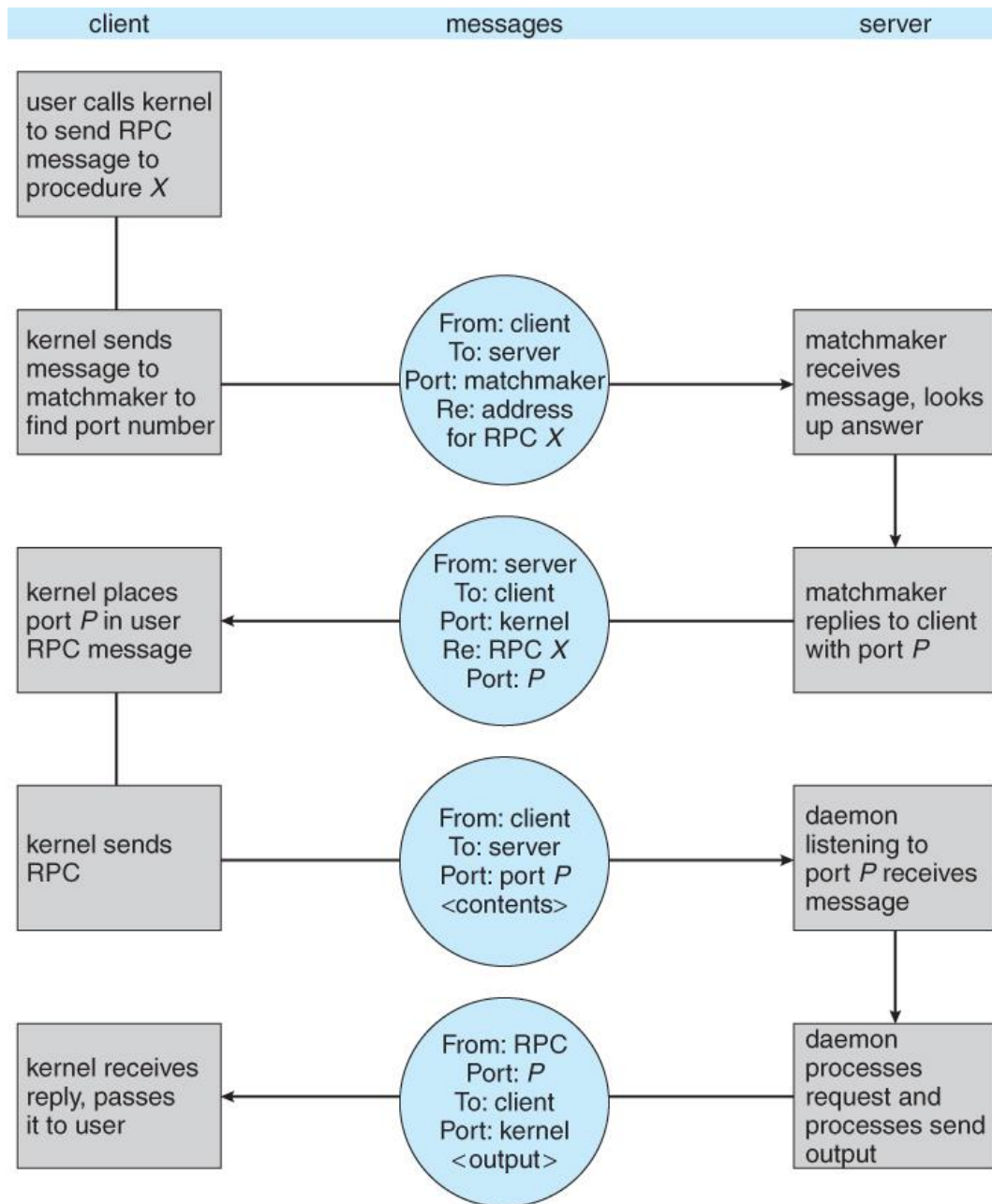
RMI stands for **Remote Method Invocation**. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package **java.rmi**.

Architecture of an RMI Application

In an RMI application, we write two programs, a **server program** (resides on the server) and a **client program** (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.



Let us now discuss the components of this architecture.

- **Transport Layer** – This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.

- **Skeleton** – This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** – It is the layer which manages the references made by the client to the remote object.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

Stub

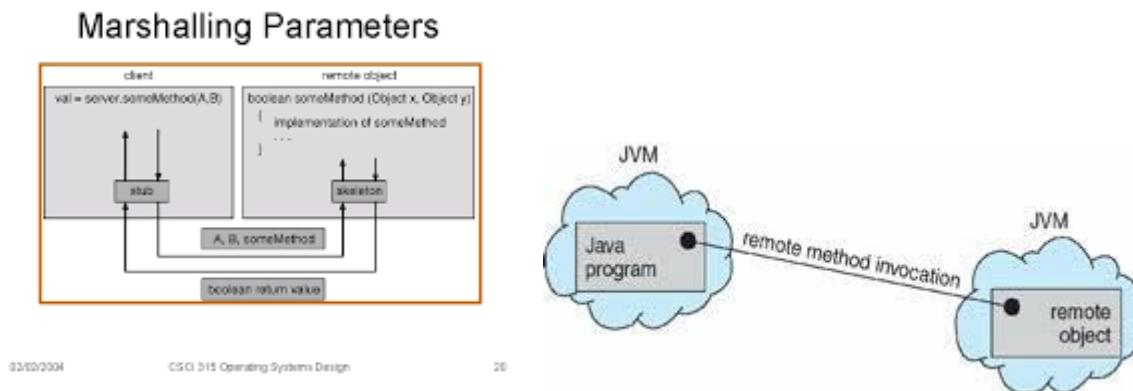
The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

1. It initiates a connection with remote Virtual Machine (JVM),
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. It finally, returns the value to the caller.

Skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

1. It reads the parameter for the remote method
2. It invokes the method on the actual remote object, and
3. It writes and transmits (marshals) the result to the caller.



Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

Goals of RMI

Following are the goals of RMI –

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects

Chapter-4 THREADS

TOPICS:

- ❖ Thread
- ❖ Multithreading Models
- ❖ Threading Issues
- ❖ Pthread

THREADS:

- The implementation of **threads** and processes differs between **operating systems**, but in most cases a **thread** is a component of a process.
- Multiple **threads** can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.
- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.
- A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism.
- Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.
- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.
- Threads have been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

The following figure shows the working of a single-threaded and a multithreaded process.

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.

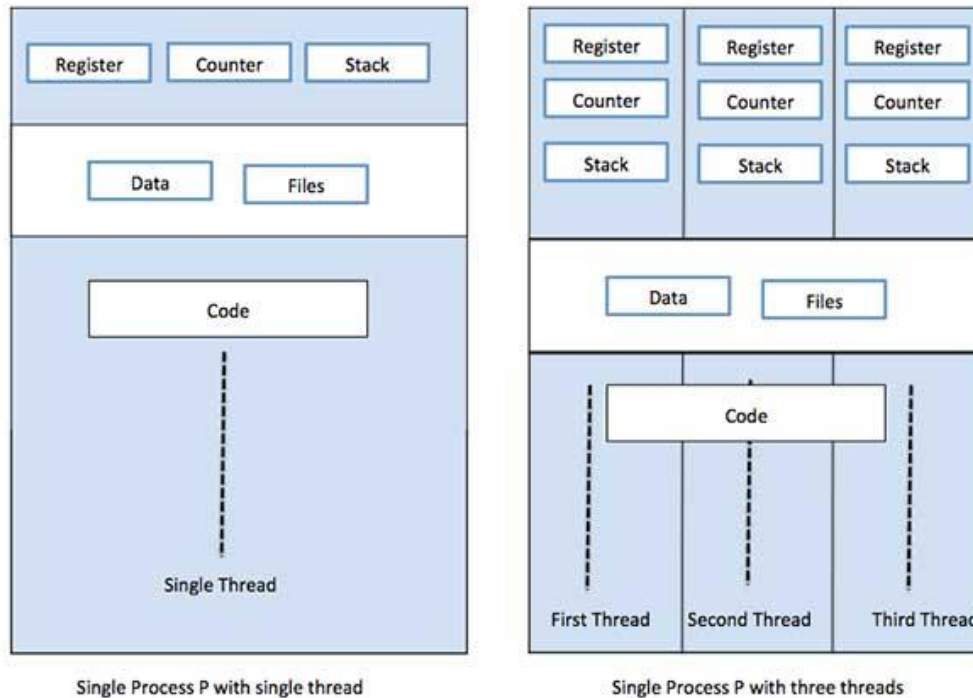


Figure: Single and Multithreaded processes

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.

Benefits:

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

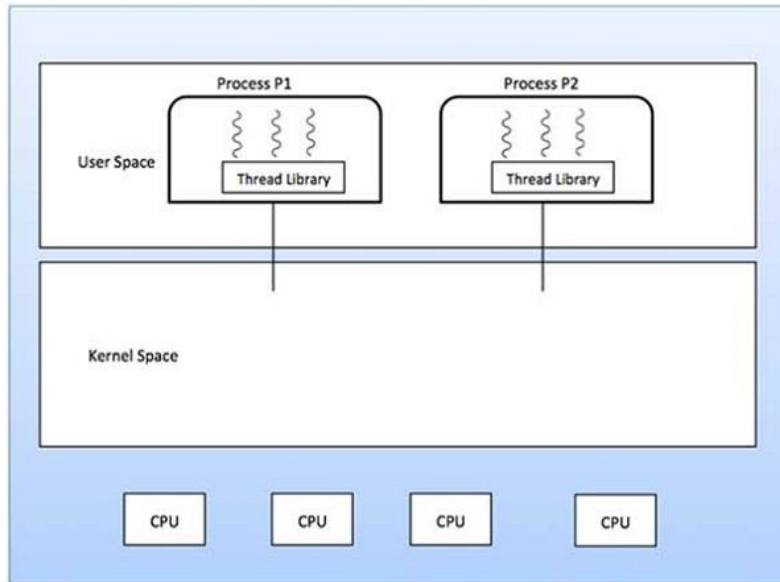
Types of Thread

Threads are implemented in following two ways –

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

User Level Threads

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



Advantages

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

Disadvantages

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

Kernel Level Threads

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

Advantages

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

Disadvantages

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

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.

❖ 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.

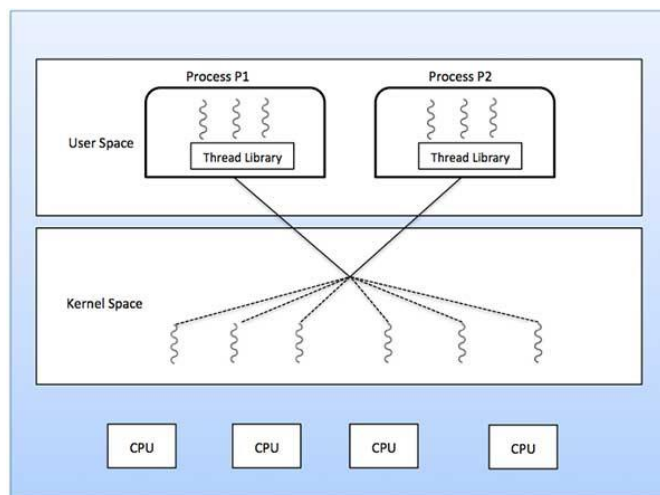
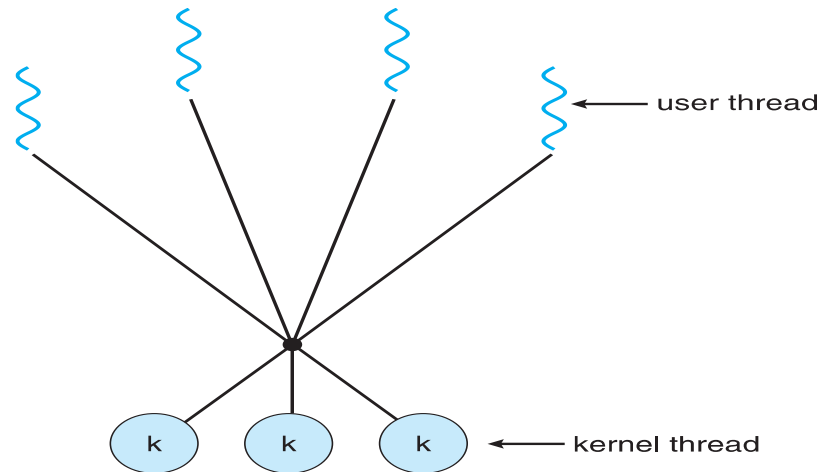


Figure: Many to Many model



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.

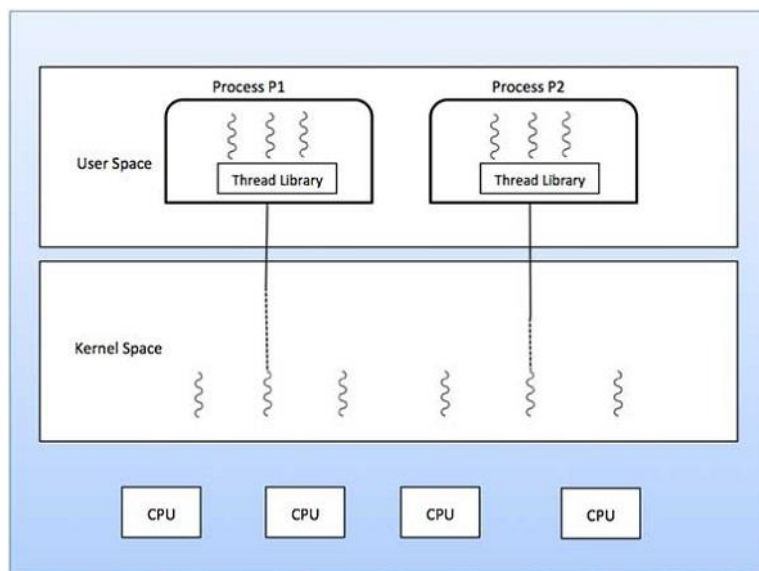
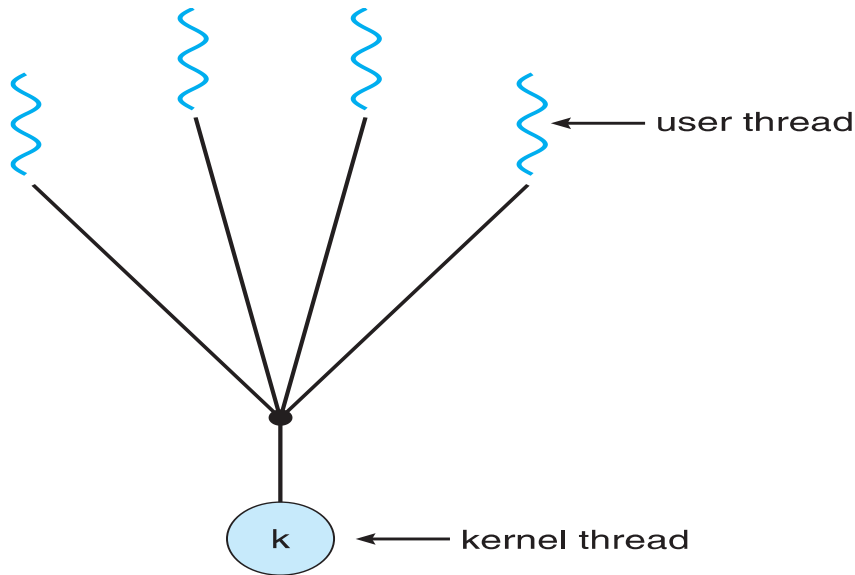


Figure: Many to One model



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.

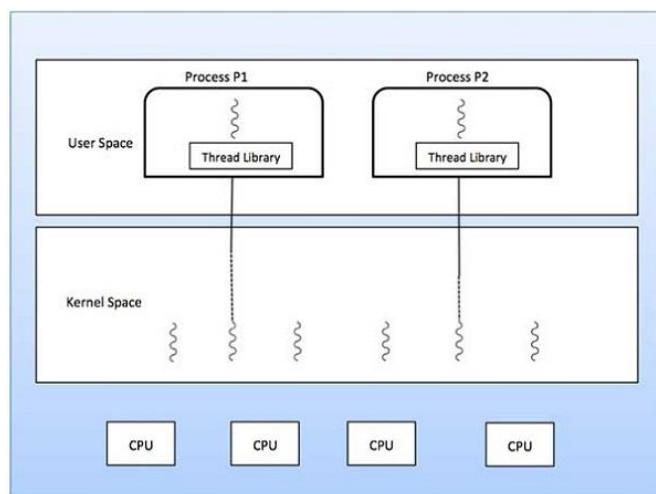
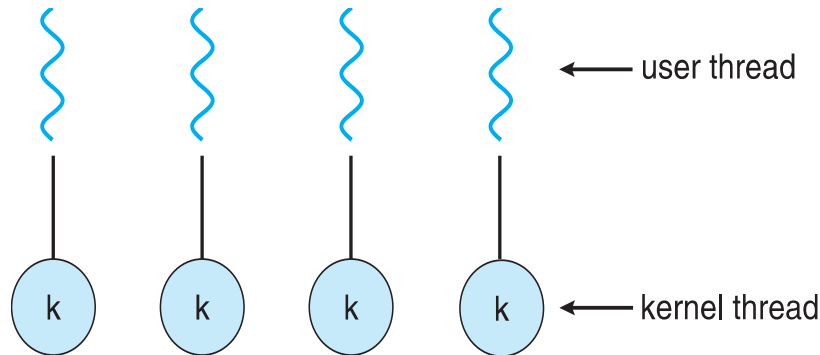


Figure: One to One model



❖ Threading Issues:

- ❖ The fork and exec system call
- ❖ Cancellation
- ❖ Signal Handling
- ❖ Thread pools
- ❖ Thread-specific data

The fork and exec system call:

- The fork system call is used to create a separate, duplicate process.
- In a multithreaded program, the semantics of the fork and exec system calls changes.
- Some UNIX system have chosen to have two versions of fork, **one that duplicates all threads** and **another that duplicates only the thread that invoked the fork system call**.
- The exec system call typically works in the same way.
- The two versions of **fork** depends upon the application. If **exec** is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec will replace the process.
- The separate process does not call exec after forking, the separate process should duplicate all threads.

Cancellations:

- **Thread cancellation** is the task of terminating a thread before it has completed.
- **For example:** if multiple threads are concurrently searching through a database and one thread return thr result, the remaining threads might be cancelled.
- **A thread** that is to be cancelled is often referred to as the **target thread**.
- **Two general approaches:**
 - **Asynchronous cancellation** - terminates the target thread immediately
 - **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.
- A signal handler is used to process signals
 - Signal is generated by particular event
 - Signal is delivered to a process
 - Signal is handled by one of **two signal handlers**:
 - **Default signal handler**
 - **user-defined signal handler**

Every signal has default handler that kernel runs when handling signal

1. User-defined signal handler can override default
2. For single-threaded, signal delivered to process

Where should a signal be delivered for multi-threaded?

- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process.

Thread pools:

- Create a number of threads in a pool where they await work

Advantages:

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
- Separating task to be performed from mechanics of creating task allows different strategies for running task
- i.e. Tasks could be scheduled to run periodically

Thread-Specific Data:

- Threads belonging to a process share the data of the process.
- Each thread might need its own copy of certain data in some circumstances. We will call such data thread-specific data.
For Example: in a transaction-processing system, we might service each transaction in a separate thread.

❖ **Pthreads:**

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- *Specification, not implementation*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
}
```



```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

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

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```


1. 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.

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer, the CPU would then sit idle; all this waiting time is wasted.

With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time when one process has to wait, the operating system takes the CPU away from that process and given the CPU to another process.

Scheduling is a fundamental operating-system function.

1. CPU-I/O Burst Cycle:

- The success of CPU scheduling depends on an observed property of processes:
- Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst.
- That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on.
- Eventually, the final CPU burst ends with a system request to terminate execution (Figure).
- The durations of CPU bursts have been measured extensively.
- Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

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

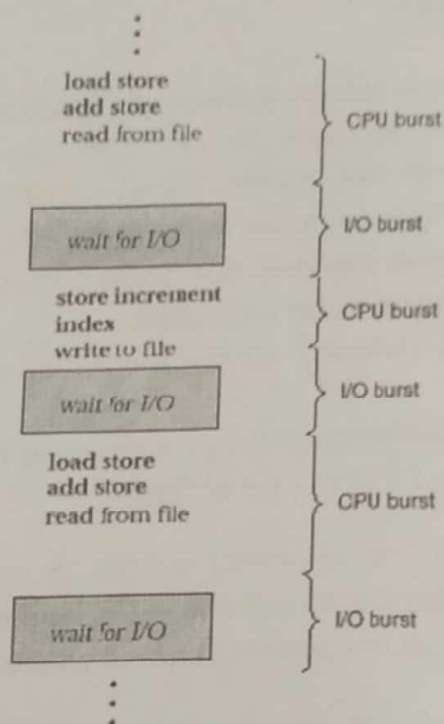


Figure : Alternating sequence of CPU and I/O bursts.

2. CPU Scheduler:

- Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
- The selection process is carried out by the short-term scheduler (or CPU scheduler).
- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue.
- As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.
- Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU.
- The records in the queues are generally process control blocks (PCBs) of the processes.

3. Preemptive Scheduling:

- CPU-scheduling decisions may take place under the following four circumstances:
 1. When a process switches from the **running state** to the **waiting state** (for example, as the result of an I/O request or an invocation of wait for the termination of one of the child processes)
 2. When a process switches from the **running state** to the **ready state** (for example, when an interrupt occurs)
 3. When a process switches from the **waiting state** to the **ready state** (for example, at completion of I/O)
 4. When a process **terminates**
- 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

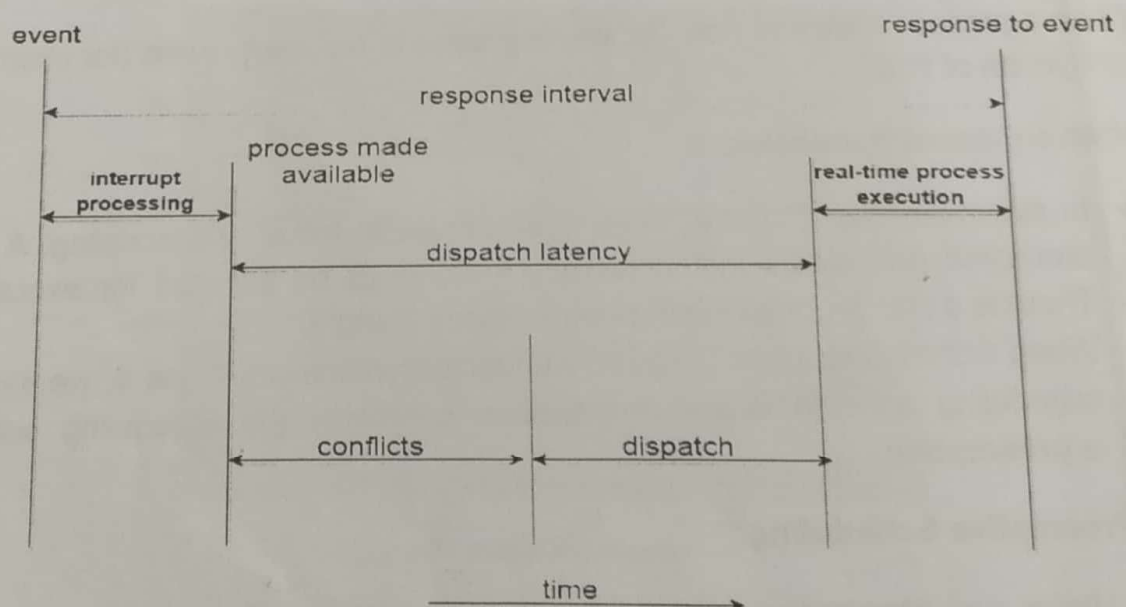
In this type of Scheduling, the tasks are usually assigned with priorities. At times it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished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

4. Dispatcher:

- Another component involved in the CPU scheduling function is the **dispatcher**.
- The dispatcher is the module that given control of the CPU to the process selected by the **short-term scheduler**.
 - Switching context
 - Switching to user mode
 - Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Dispatch Latency can be explained using the below figure:



2. Scheduling Criteria:

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

1. CPU Utilization
2. Throughput
3. Turnaround Time
4. Waiting Time
5. Response Time

CPU Utilization: We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called *throughput*. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

Turnaround Time: Turnaround time. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the **turnaround time**. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time. The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. **Waiting time** is the sum of the periods spent waiting in the ready queue.

Response time. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user.

- Thus, another measure is the time from the submission of a request until the first response is produced.
- This measure, called *response time*, is the time it takes to start responding, not the time it takes to output the response.
- The turnaround time is generally limited by the speed of the output device.
- It is desirable to maximize CPU utilization and throughput and to minimize Turn around time, waiting time, and response time.
- For example, to guarantee that all users get good service, we may want to
- example, to guarantee that all users get good service, we may want to minimize
- the maximum response time.minimize the maximum response time.

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

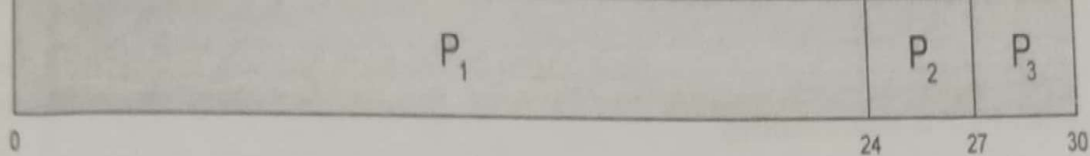
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

1. First Come First Serve Scheduling:

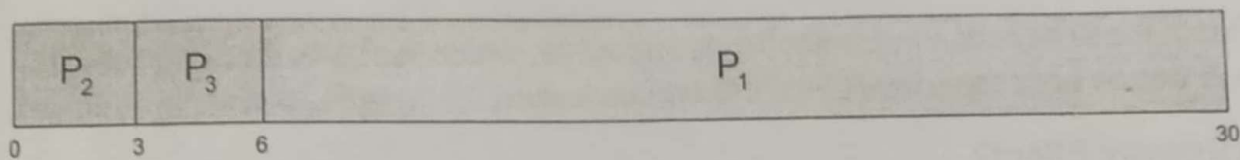
- By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS) scheduling algorithm**.
- With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.
- The code for FCFS scheduling is simple to write and understand.
- The average waiting time under the FCFS policy, however, is often quite long.
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

<u>Process</u>	<u>Burst Time</u>
P1	24
P2	3
P3	3

- If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart**:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
 - Average waiting time: $(0 + 24 + 27)/3 = 17$
- The waiting time is 0 milliseconds for process P₁, 24 milliseconds for process P₂, and 27 milliseconds for process P₃.
- Thus, the average waiting time is $(0 + 24 + 27)/3 = 17$ milliseconds.
- If the processes arrive in the order P₂, P₃, P₁, however, the results will be as shown in the following Gantt chart:
- Suppose that the processes arrive in the order:
 P_2, P_3, P_1
- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
 - Average waiting time: $(6 + 0 + 3)/3 = 3$
- The average waiting time is now $(6 + 0 + 3)/3 = 3$ milliseconds. This reduction is substantial.
- Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.
- 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 **head** of the queue.



- 8
- A perfect real life example of FCFS scheduling is **buying tickets at ticket counter**.

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.
- The FCFS scheduling algorithm is **non preemptive**.
- Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS algorithm is thus particularly troublesome for time-sharing systems where it is important that each user get a share of the CPU at regular intervals.
- It would be disastrous to allow one process to keep the CPU for an extended period.

Program for FCFS Scheduling:

- Here we have a simple C++ program for processes with **arrival time** as 0.
- In the program, we will be calculating the **Average waiting time** and **Average turn around time** for a given array of **Burst times** for the list of processes.

/* Simple C++ program for implementation
of FCFS scheduling */

(9)

```
#include<iostream>
```

```
using namespace std;
```

```
// function to find the waiting time for all processes
```

```
void findWaitingTime(int processes[], int n, int bt[], int wt[])
```

```
{
    // waiting time for first process will be 0
    wt[0] = 0;
```

```
    // calculating waiting time
```

```
    for (int i = 1; i < n ; i++)
```

```
    {
        wt[i] = bt[i-1] + wt[i-1];
    }
```

```
}
```

```
// function to calculate turn around time
```

```
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
```

```
{
    // calculating turnaround time by adding
```

```
    // bt[i] + wt[i]
```

```
    for (int i = 0; i < n ; i++)
```

```
    {
        tat[i] = bt[i] + wt[i];
    }
```

```
}
```

```
// function to calculate average time
```

```
void findAverageTime( int processes[], int n, int bt[])
```

```
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
```

```
    // function to find waiting time of all processes
    findWaitingTime(processes, n, bt, wt);
```

```
    // function to find turn around time for all processes
    findTurnAroundTime(processes, n, bt, wt, tat);
```

```
    // display processes along with all details
```

```
    cout << "Processes " << " Burst time " << " Waiting time " << " Turn around
time\n";
```

```
    // calculate total waiting time and total turn around time
```

```
    for (int i = 0; i < n; i++)
```

```
    {
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << i+1 << "\t\t" << bt[i] << "\t\t" << wt[i] << "\t\t" << tat[i] << endl;
```

```
}
```

(5)

```

    cout << "Average waiting time = "<< (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "<< (float)total_tat / (float)n;
}

// main function
int main()
{
    // process ids
    int processes[] = { 1, 2, 3, 4};
    int n = sizeof processes / sizeof processes[0];

    // burst time of all processes
    int burst_time[] = {21, 3, 6, 2};

    findAverageTime(processes, n, burst_time);

    return 0;
}

```

Processes	Burst time	Waiting time	Turn around time
1	21	0	21
2	3	21	24
3	6	24	30
4	2	30	32

Average waiting time = 18.75
 Average turn around time = 26.75

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.

For the program above, we have considered the **arrival time** to be 0 for all the processes, try to implement a program with variable arrival times.

First Come First Serve (FCFS)

Advantages:

- FCFS algorithm doesn't include any complex logic, it just puts the process requests in a queue and executes it one by one.
- Hence, FCFS is pretty simple and easy to implement.
- Eventually, every process will get a chance to run, so starvation doesn't occur.

Disadvantages:

- There is no option for pre-emption of a process. If a process is started, then CPU executes the process until it ends.
- Because there is no pre-emption, if a process executes for a long time, the processes in the back of the queue will have to wait for a long time before they get a chance to be executed.

2. Shortest Job First(SJF) Scheduling:

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

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

<u>Process</u>	<u>Burst Time</u>
P ₁	6
P ₂	8
P ₃	7
P ₄	3

$$\text{Average waiting time} = (3 + 16 + 9 + 0) / 4 = 7$$

- Using SJF scheduling, we would schedule these processes according to the following Gantt chart:
- The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4.
- Thus, the average waiting time is $(3 + 16 + 9 + 0)/4 = 7$ milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.
- The SJF scheduling algorithm is provably *optimal*, in that it gives the minimum average waiting time for a given set of processes.
- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process.
- Consequently, the *average* waiting time decreases.
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request.
- The SJF algorithm can be either **preemptive or non preemptive**.
- The choice arises when a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

average waiting time = $[(10-1)+(1-1)+(17-2)+(5-3))/4 = 26/4 = 6.5$ milliseconds.
 Process $P1$ is started at time 0, since it is the only process in the system.
 Process $P2$ arrives at time 1.

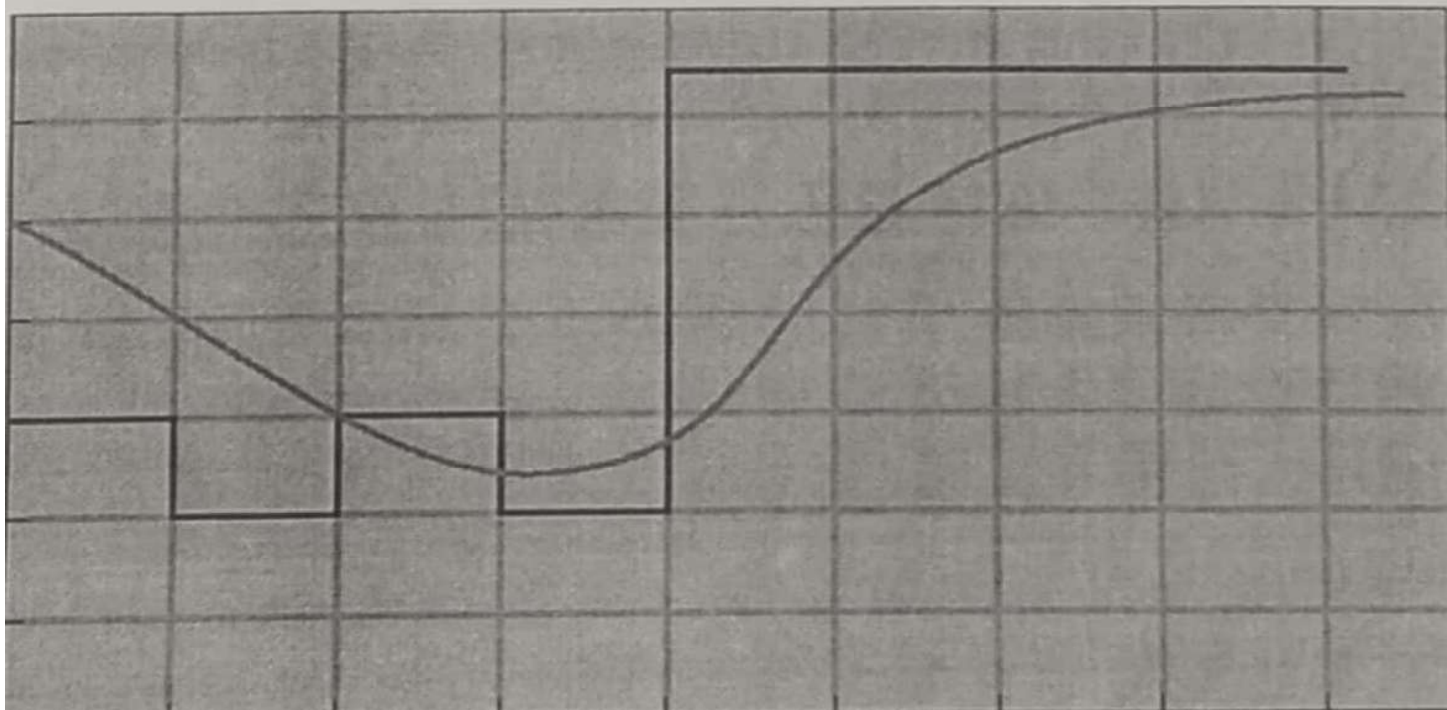
The remaining time for process $P1$ (7 milliseconds) is larger than the required time for process $P2$ (4 milliseconds), so process $P1$ is preempted and process $P2$ is scheduled.

The average waiting time for this example is $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5$ milliseconds.

Preemptive SJF scheduling would result in an average waiting time of 6.5 milliseconds.

Can only estimate the length

Can be done by using the length of previous CPU bursts, using exponential averaging



- (14)
- $\tau_{n+1} = \tau_n$
 - Recent history does not count.
 - $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts.
 - If we expand the formula, we get:
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$
 - Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

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 **Shortest Job First (SJF)**

Starting with the **Advantages:** of Shortest Job First scheduling algorithm.

- According to the definition, short processes are executed first and then followed by longer processes.
- The throughput is increased because more processes can be executed in less amount of time.

And the **Disadvantages:**

- The time taken by a process must be known by the CPU beforehand, which is not possible.
- Longer processes will have more waiting time, eventually they'll suffer starvation.

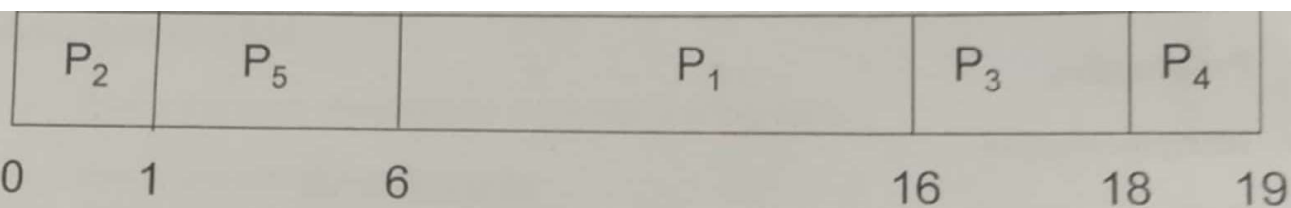
3. Priority Scheduling

- 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.
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the **highest priority** (smallest integer $\equiv$ highest priority)
- **Preemptive**
- **Nonpreemptive**
- SJF is priority scheduling where priority is the next CPU burst time
- Problem $\equiv$ **Starvation** – low priority processes may never execute
- Solution $\equiv$ **Aging** – as time progresses increase the priority of the process

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

- A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.
- Note that we discuss scheduling in terms of **high priority** and **low priority**.
- Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority.
- Some systems use low numbers to represent low priority; others use low numbers for high priority. As an example, consider the following set of processes, assumed to have arrived at time 0, in the order $P_1, P_2, \dots, P_5$, with the length of the CPU burst given in milliseconds:

Process Burst Time Priority



Average waiting time = 8.2 msec

Priority based Scheduling

Advantages of Priority Scheduling:

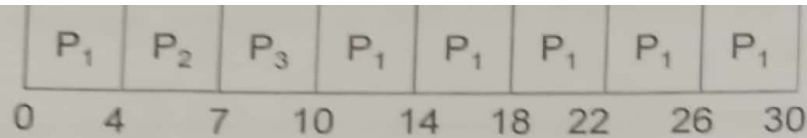
- The priority of a process can be changed.

4. Round Robin Scheduling:

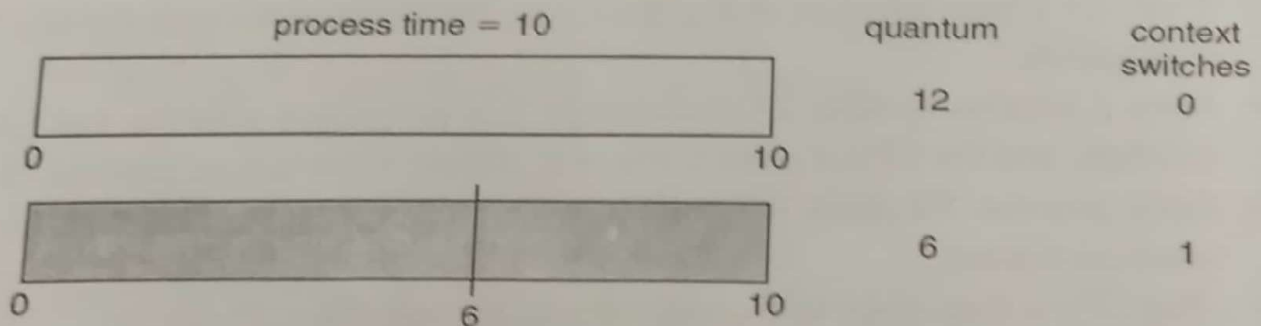
- The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum or time slice, is defined.
- A time quantum is generally from 10 to 100 milliseconds.
- The ready queue is treated as a circular queue.
- The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- 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.
- The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

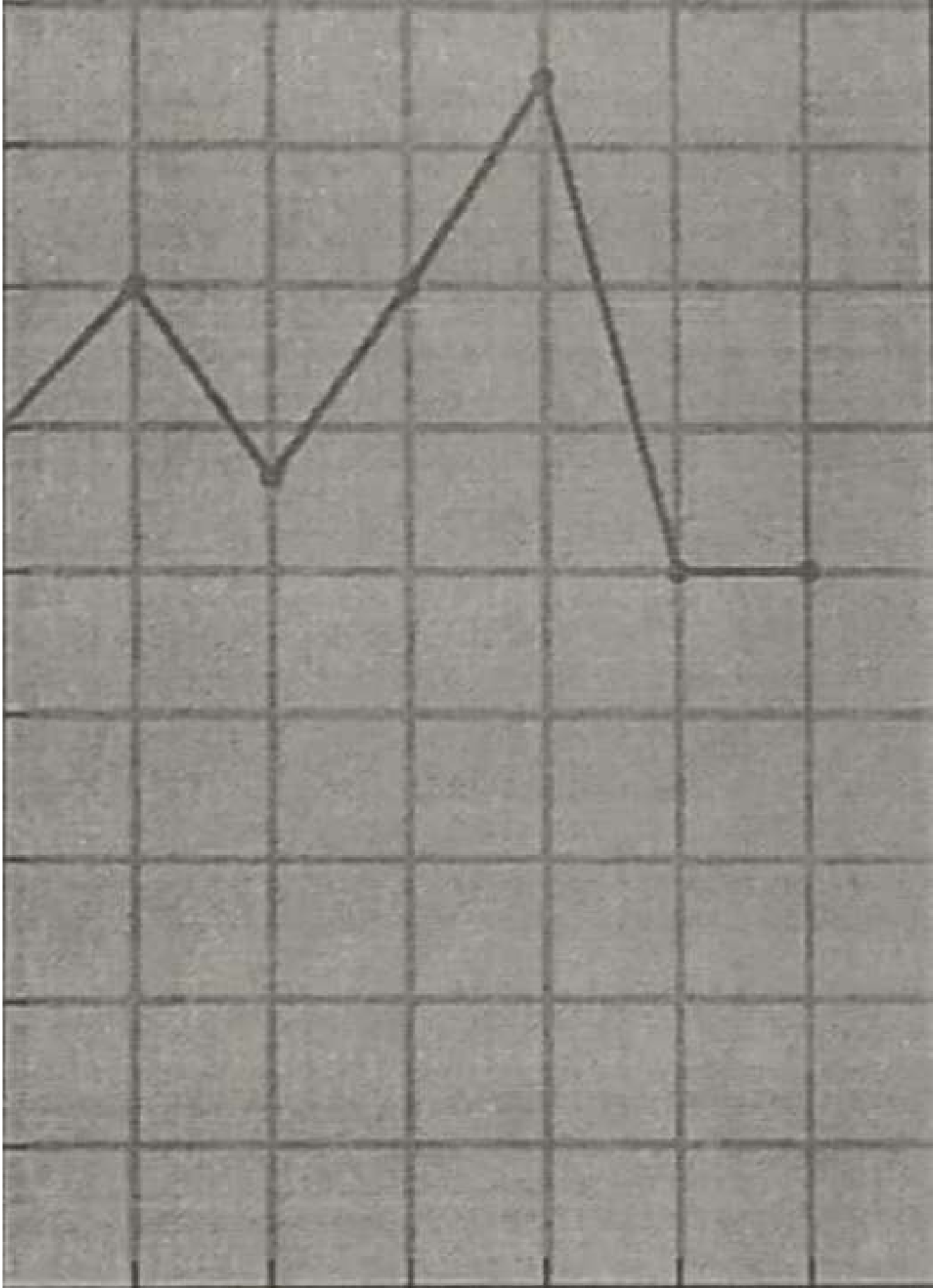
Process	Burst Time
P1	24
P2	3
P3	3

- If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds.
- Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2.
- Since process P2 does not need 4 milliseconds, it quits before its time quantum expires.
- The CPU is then given to the next process, process P3.
- Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is



- The average waiting time is $17/3 = 5.66$ milliseconds.
- In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row.
- If a process's CPU burst exceeds 1 time quantum, that process is **preempted** and is put back in the ready queue.
- The RR scheduling algorithm is thus preemptive.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units.
- Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum.
- **For example**, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.







Round Robin (RR)

Advantages: of using the Round Robin Scheduling:

- Each process is served by the CPU for a fixed time quantum, so all processes are given the same priority.
- Starvation doesn't occur because for each round robin cycle, every process is given a fixed time to execute. No process is left behind.

Disadvantages:

- The throughput in RR largely depends on the choice of the length of the time quantum. If time quantum is longer than needed, it tends to exhibit the same behavior as FCFS.
- If time quantum is shorter than needed, the number of times that CPU switches from one process to another process, increases. This leads to decrease in CPU efficiency.

4. 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.

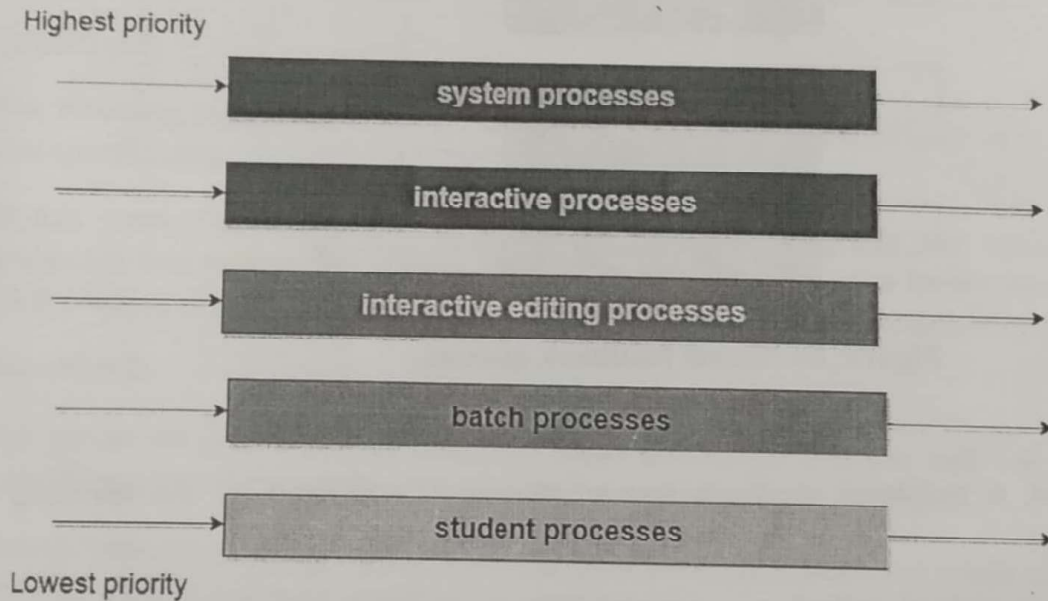
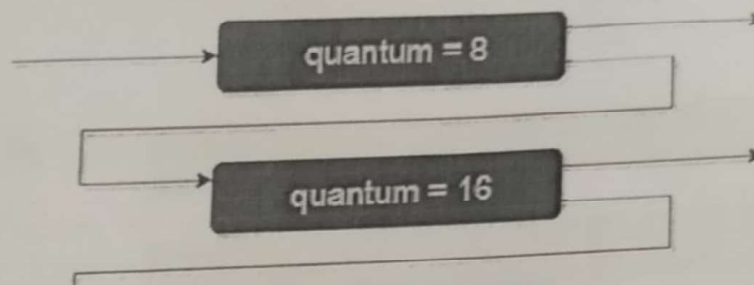


Figure: Multilevel queue scheduling.

- Another possibility is to time-slice among the queues.
- Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

- 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.



- 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**.

Comparison of Scheduling Algorithms

By now, you must have understood how CPU can apply different scheduling algorithms to schedule processes. Now, let us examine the advantages and disadvantages of each scheduling algorithms that we have studied so far.

Usage of Scheduling Algorithms in Different Situations

Every scheduling algorithm has a type of a situation where it is the best choice. Let's look at different such situations:

Situation 1:

The incoming processes are short and there is no need for the processes to execute in a specific order.

In this case, FCFS works best when compared to SJF and RR because the processes are short which means that no process will wait for a longer time. When each process is executed one by one, every process will be executed eventually.

Situation 2:

The processes are a mix of long and short processes and the task will only be completed if all the processes are executed successfully in a given time.

Round Robin scheduling works efficiently here because it does not cause starvation and also gives equal time quantum for each process.

Situation 3:

The processes are a mix of user based and kernel based processes.

Priority based scheduling works efficiently in this case because generally kernel based processes have higher priority when compared to user based processes.

- If multiple CPUs are available, load sharing among them becomes possible.
- The scheduling problem becomes more complex.
- We concentrate in this discussion on systems in which the processors are identical (homogeneous) in terms of their functionality.
- We can use any available processor to run any process in the queue.

1.Approaches to Multiple-Processor Scheduling:

- One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor-the master server.
- The other processors execute only user code.
- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Two approaches: **Asymmetric** processing and **symmetric** processing.
- **Asymmetric multiprocessing**
- **Symmetric multiprocessing (SMP)** Processor affinity – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**

Asymmetric multiprocessing (ASMP)

- One processor handles all scheduling decisions, I/O processing, and other system activities
- The other processors execute only user code
- Because only one processor accesses the system data structures, the need for data sharing is reduced

Symmetric multiprocessing (SMP)

- Each processor schedules itself
- All processes may be in a common ready queue or each processor may have its own ready queue
- Either way, each processor examines the ready queue and selects a process to execute
- Efficient use of the CPUs requires load balancing to keep the workload evenly distributed
- In a **Push** migration approach, a specific task regularly checks the processor loads and redistributes the waiting processes as needed
- In a **Pull** migration approach, an idle processor pulls a waiting job from the queue of a busy processor

- 25
- Virtually all modern operating systems support SMP, including Windows XP, Solaris, Linux, and Mac OS X.

2. Processor Affinity:

- Processor affinity – process has affinity for processor on which it is currently running
- **soft affinity**
- **hard affinity**
- The high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor.
- This is known as processor affinity, meaning that a process has an affinity for the processor on which it is currently running.
- Processor affinity takes several forms.
- When an operating system has a policy of attempting to keep a process running on the same processor-but not guaranteeing that it will do so- we have a situation known as **soft affinity**.
- Here, it is possible for a process to migrate between processors. Some systems -such as Linux-also provide system calls that support **hard affinity**, thereby allowing a process to specify that it is not to migrate to other processors.

5. Real-Time CPU Scheduling:

Real-time system:

- "A real-time system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced".
- "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment".
- Real-time computing is divided into two types.
- **Hard real time**
- **Soft real time**

- **Hard real time:** Hard real time systems are required to complete a critical task within a guaranteed amount of time. Generally, a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O.
- The scheduler then either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible. This is known as **resource reservation**.
- **Soft real-time:** Soft real-time computing is less restrictive. It requires that critical processes receive priority over less fortunate ones. Although adding soft real-time functionality to a time-sharing system may cause an unfair allocation of resources and unfair

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:

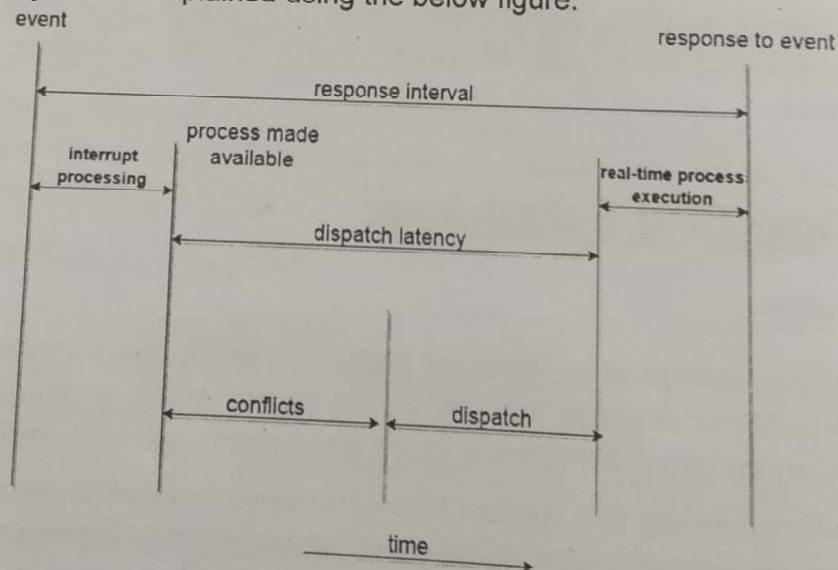


Figure: Dispatch Latency

CHAPTER – 6

Process Synchronization

TOPICS:

- The critical – section problem
- Synchronization Hardware
- Semaphores
- Classic problems of synchronization
- Critical Regions
- Monitors

Introduction to process synchronization:

Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- Independent Process
- Cooperative Process

Independent Processes

Two processes are said to be independent if the execution of one process does not affect the execution of another process.

Cooperative Processes

Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

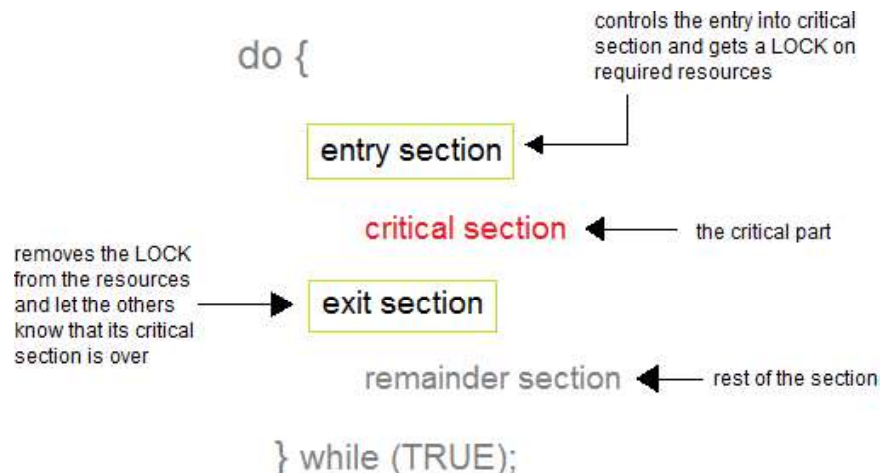
It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources.

- It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes.
- In order to synchronize the processes, there are various synchronization mechanisms.
- Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time.

- Process synchronization is the technique to overcome the problem of concurrent access to shared data which can result in data inconsistency. A cooperating process is the one which can affect or be affected by other process which will lead to inconsistency in processes data therefore Process synchronization is required for consistency of data.

❖ The Critical-Section Problem:

- Every process has a reserved segment of code which is known as **Critical Section**. In this section, process can change common variables, update tables, write files, etc.
- The key point to note about critical section is that when one process is executing in its critical section, no other process can execute in its critical section.
- Each process must request for permission before entering into its critical section and the section of a code implementing this request is the **Entry Section**, the end of the code is the **Exit Section** and the remaining code is the **remainder section**.
- 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.
- The entry to the critical section is mainly handled by **wait()** function while the exit from the critical section is controlled by the **signal()** function.



Entry Section

- In this section mainly the process requests for its entry in the critical section.

Exit Section

- This section is followed by the critical section.

The solution to the 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, the system must grant the process permission to get into its critical section.

Synchronization Hardware:

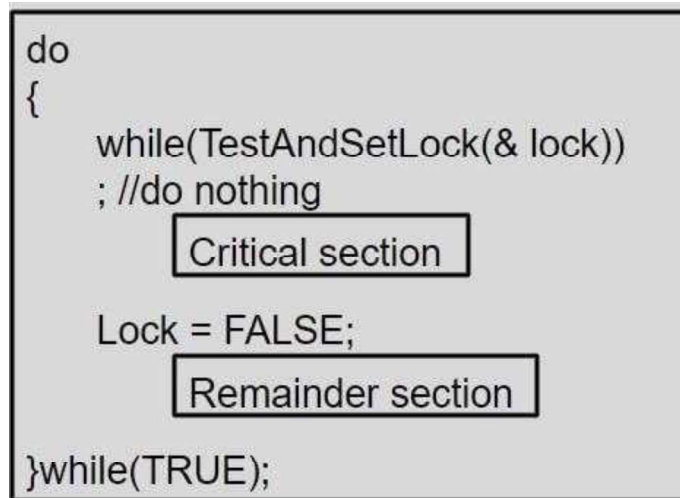
It is implemented using two types of instructions –

- Test and Set()
- swap()

Test and Set () is a hardware solution to solve the problem of synchronization. In this, there is a shared variable which is shared by multiple processes known as Lock which can have one value from 0 and 1 where 1 represents Lock gained and 0 represents Lock released.

Whenever the process is trying to enter their critical sections they need to enquire about the value of lock. If the value of lock is 1 then they have to wait until the value of lock won't get changed to 0.

Given below is the mutual-exclusion implementation with TestAndSet()



- In Synchronization hardware, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software based APIs available to application programmers.
- These solutions are based on the premise of locking; however, the design of such locks can be quite sophisticated.
- These Hardware features can make any programming task easier and improve system efficiency. Here, we present some simple hardware instructions that are available on many systems and show how they can be used effectively in solving the critical-section problem.
- If we could prevent interrupts from occurring while a shared variable was being modified. The critical-section problem could be solved simply in a uniprocessor environment.
- In this manner, we would be assuring that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.
- This is the approach taken by non-preemptive kernels. But unfortunately, this solution is not as feasible in a multiprocessor environment. Since the message is passed to all the processors, disabling interrupts on a multiprocessor can be time consuming.
- We may use these special instructions to solve the critical-section problem in a relatively simple manner. Now we abstract the main concepts behind these types of instructions. The TestAndSet() instruction can be defined as shown in below code.

```

boolean test and set(boolean *target){
    boolean rv = *target;
  
```



```

    *target = true;

    return rv;
}

```

Definition of the test and set() instruction.

The essential characteristic is that this instruction is executed atomically. So, if two TestAndSet C) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false, if the machine supports the TestAndSet () instruction.

Semaphores:

Semaphores

Semaphore is a synchronization tool that is used to overcome the problems generated by TestAndSet() and Swap() instructions. A semaphore S is an integer variable that can be accessed through two standard atomic operations that are wait() and signal()

Function for wait():

```

wait(S) {
    While S <= 0
    ; // no operation
    S--;
}

```

Function for Signal():

```

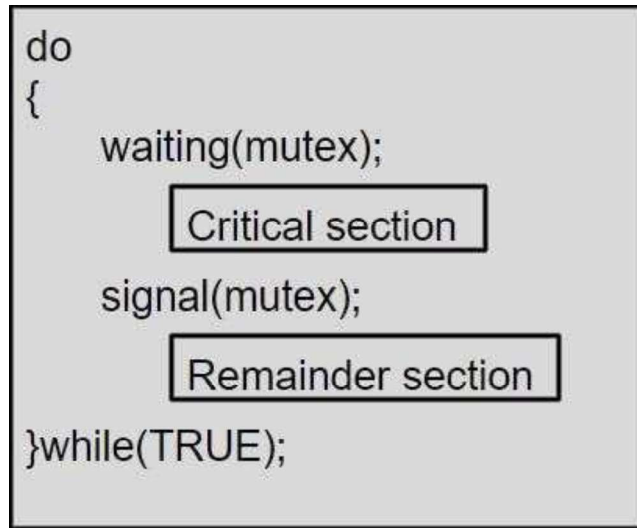
signal(S) {
    S++;
}

```

When one process is modifying the value of semaphore then no other process can simultaneously manipulate the same semaphore value.

Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual exclusion. We can use binary semaphores to deal with the critical-section problem for multiple processes



Implementation:

The main disadvantage of the semaphore definition given here is that it requires busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.

This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

Properties of Semaphores

1. It's simple and always have a non-negative integer value.
2. Works with many processes.
3. Can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphores.
5. Can permit multiple processes into the critical section at once, if desirable.

Types of Semaphores

Semaphores are mainly of two types in Operating system:

1. Binary Semaphore:

It is a special form of semaphore used for implementing mutual exclusion, hence it is often called a **Mutex**. A binary semaphore is initialized to **1** and only takes the values **0** and **1** during the execution of a program. In Binary Semaphore, the wait operation works only if the value of semaphore = 1, and the signal operation

succeeds when the semaphore = 0. Binary Semaphores are easier to implement than counting semaphores.

2. Counting Semaphores:

These are used to implement **bounded concurrency**. The Counting semaphores can range over an **unrestricted domain**. These can be used to control access to a given resource that consists of a finite number of Instances. Here the semaphore count is used to indicate the number of available resources. If the resources are added then the semaphore count automatically gets incremented and if the resources are removed, the count is decremented. Counting Semaphore has no mutual exclusion.

Advantages of Semaphores

- With the help of semaphores, there is a flexible management of resources.
- Semaphores are machine-independent and they should be run in the machine-independent code of the microkernel.
- Semaphores do not allow multiple processes to enter in the critical section.
- They allow more than one thread to access the critical section.
- As semaphores follow the mutual exclusion principle strictly and these are much more efficient than some other methods of synchronization.

Disadvantages of Semaphores

- One of the biggest limitations is that semaphores may lead to priority inversion; where low priority processes may access the critical section first and high priority processes may access the critical section later.
- To avoid deadlocks in the semaphore, the Wait and Signal operations are required to be executed in the correct order.
- Using semaphores at a large scale is impractical; as their use leads to loss of modularity and this happens because the wait() and signal() operations prevent the creation of the structured layout for the system.
- Their use is not enforced but is by convention only.
- With improper use, a process may block indefinitely. Such a situation is called **Deadlock**.

Classical Problems of Synchronization:

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

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

We will discuss the following three problems:

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

Bounded Buffer Problem:

Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.

- 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.
- The solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- This Producers mainly produce a product and consumers consume the product, but both can use one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

```
do {
    // produce an item in nextp

    wait (empty) ;

    wait (mutex) i

    // add nextp to buffer

    signal (mutex) ;

    signal (full) ;

}while (TRUE) i
```

Figure :The structure of the producer process.

```
do {

    wait(full) ;

    wait (mutex) ;

    // remove an item from buffer to nextc
```

```

signal (mutex) ;

signal (empty) ;

// consume the item in nextc

}while (TRUE);

```

Figure : The structure of the consumer process.

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization. Let's start by understanding the problem here, before moving on to the solution and program code.

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.

Dining philosopher's problem:

- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the center. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

The dining philosophers problem is another classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes.

Consider there are five philosophers sitting around a circular dining table. The dining table has five chopsticks and a bowl of rice in the middle as shown in the below figure.

```

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.

At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick. Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.

But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever



Figure : The situation of the dining philosophers

The possible solutions for this are:

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.
- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

The Readers Writers Problem

- In this problem, there are some processes (called **readers**) that only read the shared data, and never change it, and there are other processes (called **writers**) that may change the data in addition to reading, or instead of reading it.
- There is various type of readers-writers problems, most centered on relative priorities of readers and writers.
- The main complexity of this problem occurs from allowing more than one reader to access the data at the same time. Readers writer problem is another example of a classic synchronization problem.
- There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**.
- Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource.
- When a **writer** is writing data to the resource, no other process can access the resource.
- A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last

```
do {
    wait (wrt) ;

    // writing is performed

    signal (wrt) ;
}while (TRUE);
```

Figure :The structure of a writer process.

```

do {
wait (mutex) ;
readcount++;
if (readcount ==1)
wait (wrt);
signal (mutex) ;
// reading is performed
wait (mutex) ;
readcount - -;
if (readcount ==0)
signal (wrt) ;
signal (mutex) ;
}while (TRUE) ;

```

Figure : The structure of a reader process.

reader that enters or exits the critical section. It is not used by readers.

Monitors:

- Another high-level synchronization construct is the monitor type.
 - The monitor is one of the ways to achieve Process synchronization.
 - The monitor is supported by programming languages to achieve mutual exclusion between processes.
 - **For example** Java Synchronized methods. Java provides wait() and notify() constructs.
1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
 2. The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.
 3. Only one process at a time can execute code inside monitors.

Syntax:

```

Monitor Demo //Name of Monitor
{
  variables;
  condition variables;

  procedure p1 {...}
  prodecure p2 {...}

}

Syntax of Monitor
    
```

Condition Variables:

Two different operations are performed on the condition variables of the monitor.

Wait.

signal.

condition x, y;

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended.

The suspended processes are placed in block queue of that condition variable.

Note: Each condition variable has its unique block queue.

Signal operation

x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

If (x block queue empty)

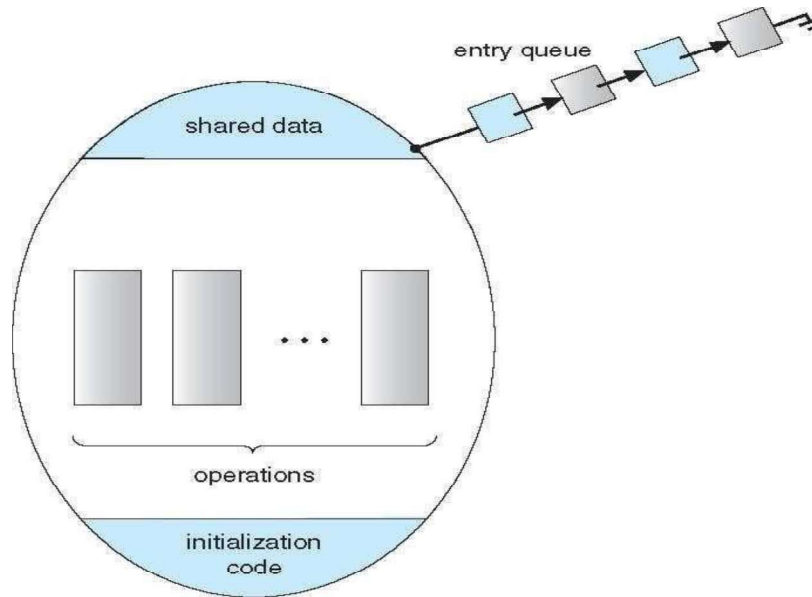


Figure: Schematic view of a monitor

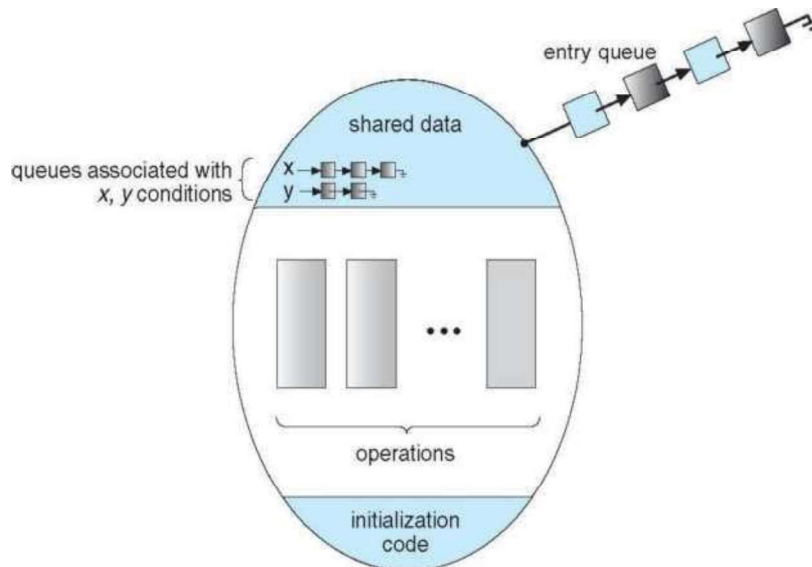


Figure: Monitor with condition variable

Characteristics of Monitors.

- Inside the monitors, we can only execute one process at a time.
- Monitors are the group of procedures, and condition variables that are merged together in a special type of module.

- If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.
- Monitors offer high-level of synchronization
- Monitors were derived to simplify the complexity of synchronization problems.
- There is only one process that can be active at a time inside the monitor.

Components of Monitor

There are four main components of the monitor:

- Initialization
- Private data
- Monitor procedure
- Monitor entry queue

Initialization: – Initialization comprises the code, and when the monitors are created, we use this code exactly once.

Private Data: – Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.

Monitor Procedure: – Monitors Procedures are those procedures that can be called from outside the monitor.

Monitor Entry Queue: – Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.

Advantages of Monitor:

Monitors have the advantage of making parallel programming easier and less error prone than using techniques such as semaphore.

Disadvantages of Monitor:

- Monitors have to be implemented as part of the programming language.

The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java, C#, Visual Basic, Ada .

CHAPTER -7

DEADLOCKS

TOPICS:

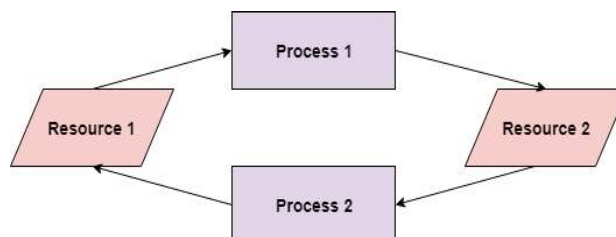
- **Deadlock Characterization**
- **Deadlock Handling**
- **Deadlock Prevention**
- **Deadlock Avoidance**
- **Deadlock Detection**
- **Deadlock Recovery**

Deadlock definition:

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

A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.



Deadlock in Operating System

- In the above diagram, the process 1 has resource 1 and needs to acquire resource 2. Similarly process 2 has resource 2 and needs to acquire resource 1.

- Process 1 and process 2 are in deadlock as each of them needs the other's resource to complete their execution but neither of them is willing to relinquish their resources.

Deadlock Characterization:

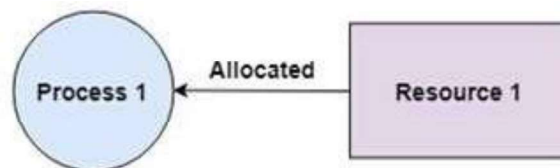
Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in the system.

The four conditions are given as follows –

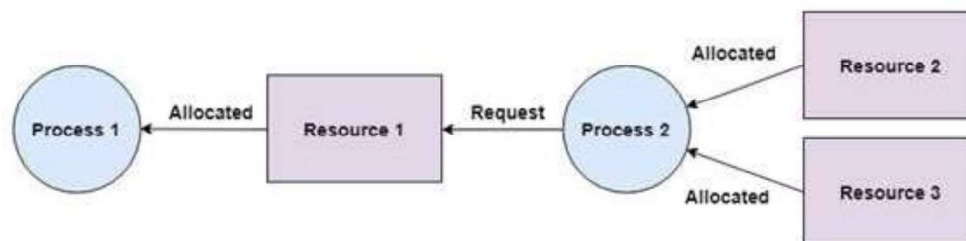
- **Mutual Exclusion**

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



- **Hold and Wait**

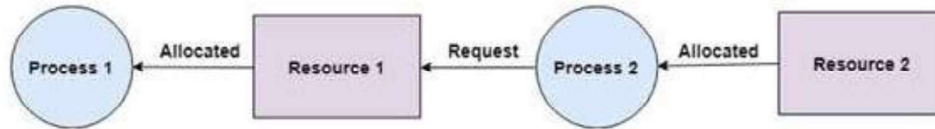
A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



- **No Preemption**

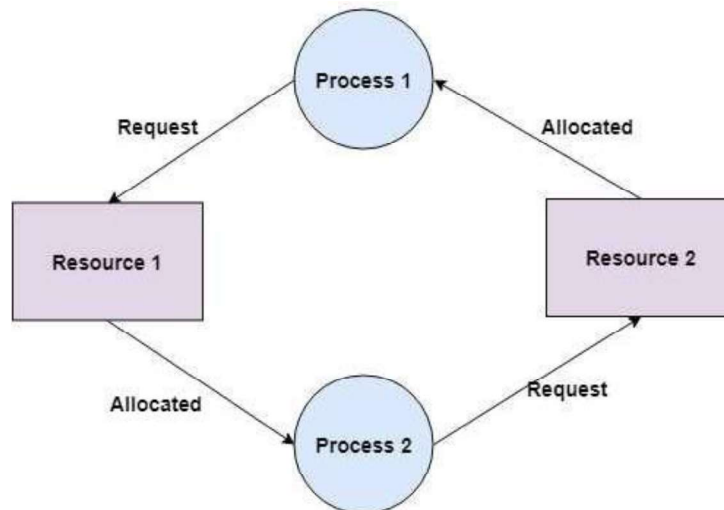
A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt

Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



- **Circular Wait**

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Resource – Allocation Graph :

Deadlocks can be described more precisely in terms of a directed graphs called a **system Resource Allocation Graph**.

- This Graph acts as the pictorial representation of the state of the system.
- The Resource Allocation graph mainly consists of a **set of vertices V** and a **set of Edges E**.

- This graph mainly contains all the information related to the processes that are holding some resources and also contains the information of the processes that are waiting for some more resources in the system.
- Also, this graph contains all the information that is related to all the instances of the resources which means the information about available resources and the resources which are being used by the process
- In this graph, the **circle is used to represent the process**, and the **rectangle is used to represent the resource**.

Components of Resource Allocation Graph

Given below are the components of RAG:

1. Vertices
2. Edges

1.Vertices

There is two kinds of vertices used in the resource allocation graph and these are:

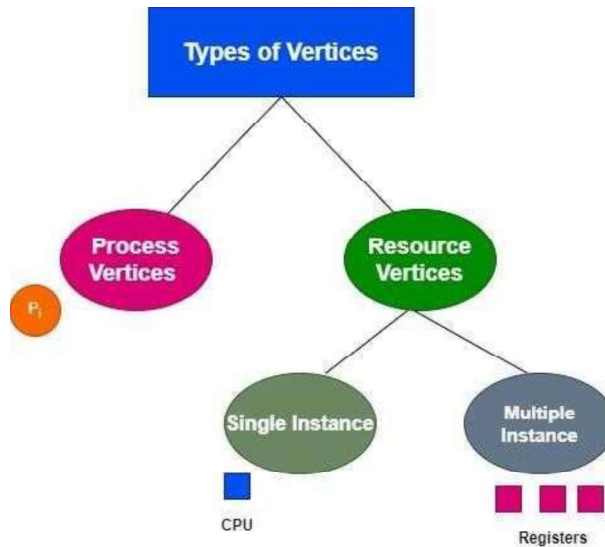
- Process Vertices
- Resource Vertices

Process Vertices

These vertices are used in order to represent process vertices. The circle is used in order to draw the process vertices and the name of the process is mentioned inside the circle.

Resource Vertices

These vertices are used in order to represent resource vertices. The rectangle is used in order to draw the resource vertices and we use dots inside the circle to mention the number of instances of that resource.



In the system, there may exist a number of instances and according to them, there are two types of resource vertices and these are single instances and multiple instances.

Single Instance

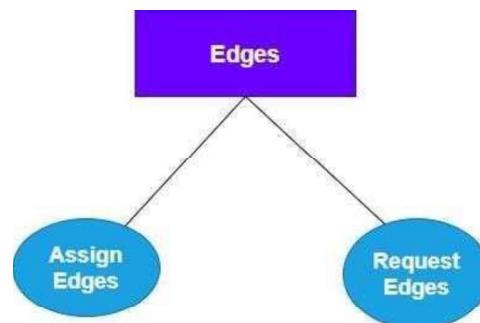
In a single instance resource type, there is a single dot inside the box. The single dot mainly indicates that there is one instance of the resource.

Multiple Instance

In multiple instance resource types, there are multiple dots inside the box, and these Multiple dots indicate that there are multiple instances of the resources.

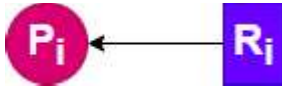
2. Edges

In the Resource Allocation Graph, Edges are further categorized into two:



1. Assign Edges

Assign Edges are mainly used to represent the allocation of resources to the process. We can draw assign edges with the help of an arrow in which mainly the arrowhead points to the process, and the process mainly tail points to the instance of the resource.



In the above Figure, the resource is assigned to the process

2. Request Edges

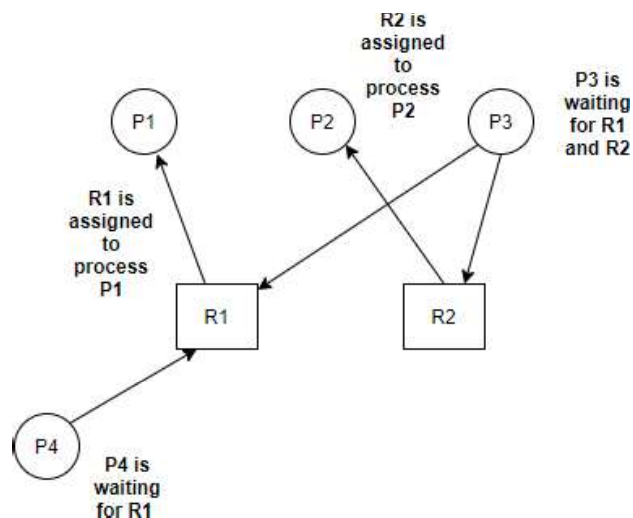
Request Edge is mainly used to signify the waiting state of the process. Likewise in assigned edge, an arrow is used to draw an arrow edge. But Here the arrowhead points to the instance of a resource, and the tail of the process points to the process.



In the above figure, the process is requesting a resource

Single Instance RAG Example

Suppose there are Four Processes P1, P2, P3, P4, and two resources R1 and R2, where P1 is holding R1 and P2 is holding R2, P3 is waiting for R1 and R2 while P4 is waiting for resource R1.

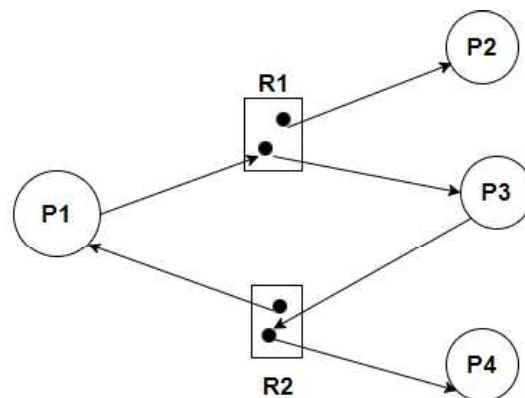


In the above example, there is no circular dependency so there are no chances for the occurrence of deadlock.

Thus having cycled in single-instance resource type must be the sufficient condition for deadlock.

Multiple Instance RAG Example

Suppose there are four processes P1, P2, P3, P4 and there are two instances of resource R1 and two instances of resource R2:



Multiple Instance Resource Allocation graph with a cycle but no deadlock

One instance of R2 is assigned to process P1 and another instance of R2 is assigned to process P4, Process P1 is waiting for resource R1.

One instance of R1 is assigned to Process P2 while another instance of R2 is assigned to process P3, Process P3 is waiting for resource R2.

- **For example:**

Resource -allocation graph shown in below figure the following situation.

- ❖ The set P, R and E:
 - $P = \{P1, P2, P3\}$
 - $R = \{R1, R2, R3, R4\}$
 - $E = \{P1 \rightarrow R1, P2 \rightarrow R3, R1 \rightarrow P2, R2 \rightarrow P2, R2 \rightarrow P1, R3 \rightarrow P3\}$
- ❖ Resource instances:
 - One instance of resource type R1
 - Two instance of resource type R2
 - One instance of resource type R3
 - Three instance of resource type R4

❖ Process state:

- Process P1 is holding an instance of resource type R2, and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.

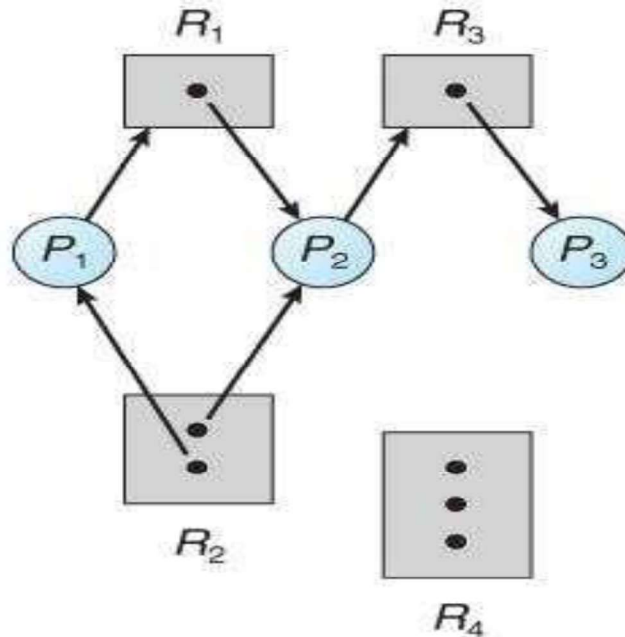


Figure - Resource allocation graph

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

$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$

$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$

- ❖ Processes P1,P2,and P3 are deadlocked.
- ❖ Process P2 is waiting for the resource R3,which is held by process P3.

- ❖ Process P3, on the other hand, is waiting for either process P1 or Process P2 to release resource R2.
- ❖ Process P1 is waiting for process P2 to release resource R1.
- ❖ The resource –allocation graph in figure also have a cycle.
- ❖ $P1 \rightarrow R1 \rightarrow P3 \rightarrow R2 \rightarrow P1$

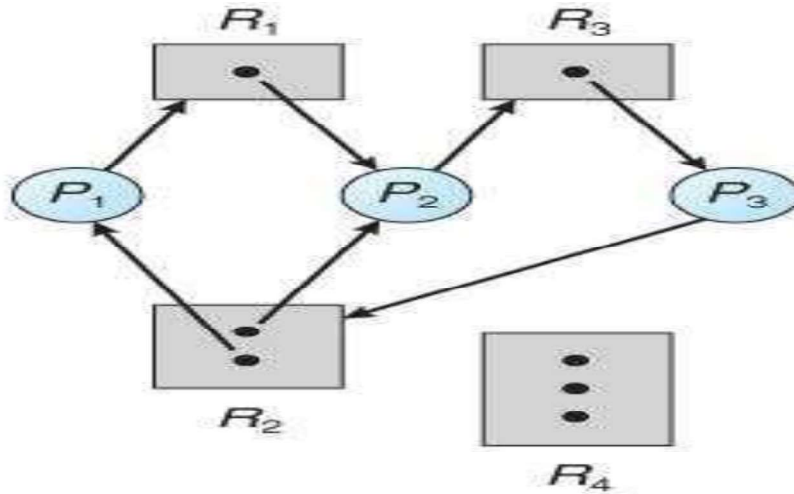


Figure - Resource allocation graph with a deadlock.

Methods for Handling Deadlocks:

- Generally speaking there are three ways of handling deadlocks:
- Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
- Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
- 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 the constant overhead and system performance associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

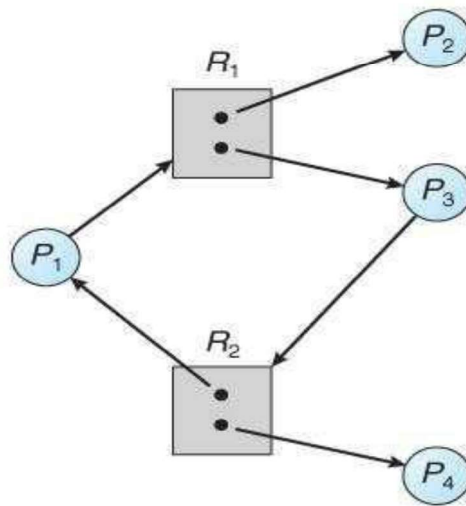


Figure - Resource allocation graph with a cycle but no deadlock

- 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 straight forward, but deadlock recovery requires either aborting processes.
- 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.

Deadlock Prevention

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

Mutual Exclusion

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

Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.

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

No Preemption

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

Circular Wait

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

Deadlock Avoidance:

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization.

- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation **state** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

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

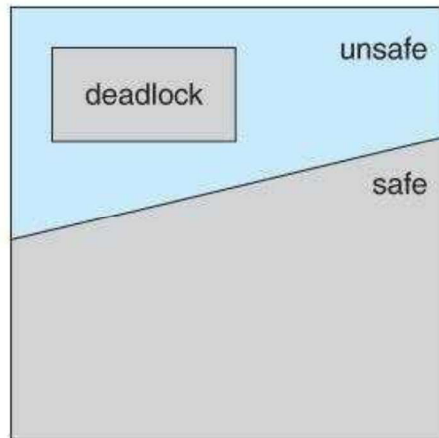


Figure: Safe, unsafe, and deadlocked state spaces.

- 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.

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 $P_i \rightarrow R_j$ 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:

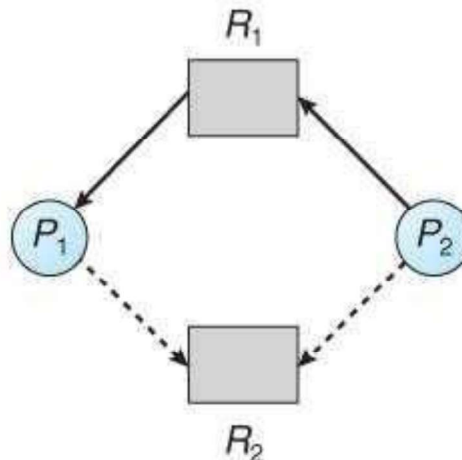


Figure : Resource allocation graph for deadlock avoidance

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

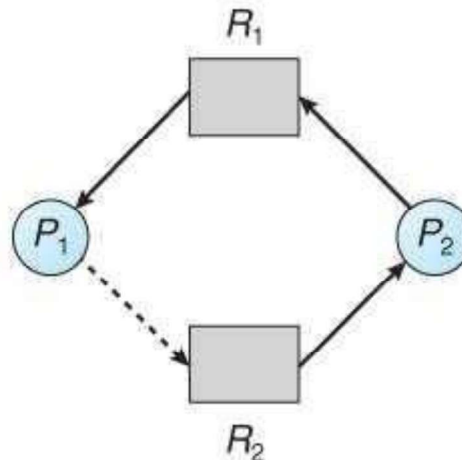


Figure - An unsafe state in a resource allocation graph

Banker's Algorithm

- ❖ The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the **banker's algorithm**.
- **The 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 $\leq$ a vector Y if $X[i] \leq Y[i]$ for all i.

Safety Algorithm

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

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

1. Let Work and Finish be vectors of length m and n respectively. Initialize Work := Available and Finish[i] = false for $i = 1, 2, \dots, n$.

2. Find an i such that both

a. Finish[i] == false

b. Need i < or = Work

If no such i exists, go to step 4.

3. Work := Work + Allocation;

Finish[i] = true

Go to step 2.

4. If Finish[i] == true for all i, then the system is in a safe state.

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

7.5.3.2 Resource-Request Algorithm

We now describe the algorithm which determines if requests can be safely granted.

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

1. If Request[i] < or = Need i, go to step 2. Otherwise/ raise an error condition, since the process has exceeded its maximum claim.
2. If Request[i] < or = Available, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

Available := Available – Request i;

Allocation $i = \text{Allocation } i + \text{Request } i$;

Need $i = \text{Need } i - \text{Request } i$;

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

An Illustrative Example:

- Consider a system with five processes P_0 through P_4 and three resource types A,B,C. Resource type A has 10 instance, resource type B has 5 instances, and resource type C has 7 instances. Suppose that, at a time T_0 , the following snapshot of the system has been taken:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- The consider of the matrix Need is defined to be Max-Allocation and is follow:
- We claim that the system is currently in a safe state. Indeed, the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria.
- Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C, so Request $_1 = (1,0,2)$. To decide whether this request can be immediately granted, we first check that Request $_1 < \text{or } = \text{Available}$ (that is, that $(1,0,2) < \text{or } = (3,3,2)$), which is true. We then pretend. that this request has been fulfilled, and we arrive at the following new state:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- We must determine whether this new system state is safe.
- To do so, we execute our safety algorithm and find that the sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies our safety requirements.
- Hence, we can immediately grant the request of process P_1 .

Deadlock Detection

- If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment, the system must provide:
 - **An algorithm that examines the state of the system to determine whether a deadlock has occurred**
 - **An algorithm to recover from the deadlock**
- In the following discussion, we elaborate on these two requirements as they pertain to systems with only a **single instance of each resource type**, as well as to systems with **several instances of each resource type**.
- **Single Instance of Each Resource Type:**
 - If each resource category has a single instance, then we can define a deadlock detection algorithm that uses 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 edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs.
 - An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q .

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

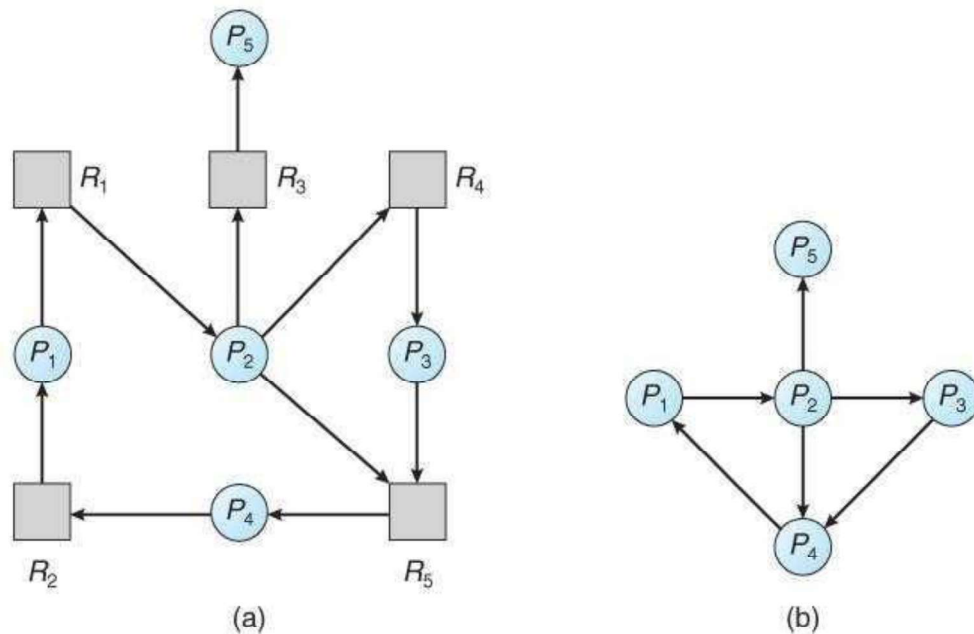


Figure - (a) Resource allocation graph. (b) Corresponding wait-for graph

Several Instances of a Resource Type:

- ❖ The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system. The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm.
- ❖ **Available:** A vector of length m indicates the number of available resources of each type.
- ❖ **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- ❖ **Request:** An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i, j] = k$, then process P_i is requesting k more instances of resource type R_j .

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work:=Available. For $i=1,2,\dots,n$ if Allocation $i \neq 0$, then Finish[i] := false otherwise, Finish[i] :=true.
 2. Find an index i such that both
 - a. Finish[i] = false.
 - b. Request $i \leq$ Work.
 If no such i exists, go to step 4.
 3. Work := Work + Allocation;
 Finish[i] := true
 Go to step 2.
 4. If Finish[i] = false, for some i, $1 \leq i \leq n$, then the system is in a deadlocked state. Moreover, if Finish[i] = false, then process P_i is deadlocked.
- ❖ This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.
 - ❖ To illustrate this algorithm, we consider a system with **five processes P_0 through P_4** and **three resource types A, B, and C**.
 - ❖ Resource type A has **seven** instances, resource type B has **two** instances, and resource type C has **six** instances.
 - ❖ Suppose that, at time T_0 , we have the following resource-allocation state:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

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

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

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

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

Detection-Algorithm Usage

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

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

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

Recovery From Deadlock

- There are three basic approaches to recovery from deadlock:
 1. Inform the system operator, and allow him/her to take manual intervention.
 2. Terminate one or more processes involved in the deadlock
 3. Preempt resources.

Process Termination

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

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

- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked
- 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 the process has used (for example, whether the resources are simple to preempt)
 4. How many more resources the process needs in order to complete.
 5. How many processes will need to be terminated
 6. Whether the process is interactive or batch.

Resource Preemption

- When preempting resources to relieve deadlock, there are three important issues to be addressed:
- **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
- **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.)
- **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.

UNIT-3
CHAPTER-8
MEMORY MANAGEMENT

TOPICS:

- **Swapping**
- **Contiguous Memory Allocation**
- **Paging**
- **Segmentation**
- **Segmentation with paging**

Memory Management

- Memory Management is the process of controlling and coordinating computer memory, assigning portions known as blocks to various running programs to optimize the overall performance of the system.
- It is the most important function of an operating system that manages primary memory. It helps processes to move back and forward between the main memory and execution disk. It helps OS to keep track of every memory location, irrespective of whether it is allocated to some process or it remains free.

Why Use Memory Management?

Here, are reasons for using memory management:

- It allows you to check how much memory needs to be allocated to processes that decide which processor should get memory at what time.
- Tracks whenever inventory gets freed or unallocated. According to it will update the status.
- It allocates the space to application routines.
- It also make sure that these applications do not interfere with each other.
- Helps protect different processes from each other
- It places the programs in memory so that memory is utilized to its full extent.

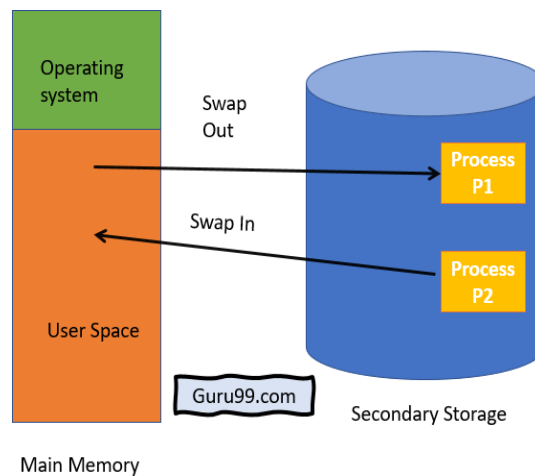
What is Swapping?

Swapping is a method in which the process should be swapped temporarily from the main memory to the backing store.

It will be later brought back into the memory for continue execution.

Backing store is a hard disk or some other secondary storage device that should be big enough inorder to accommodate copies of all memory images for all users.

It is also capable of offering direct access to these memory images.



Benefits of Swapping

Here, are major benefits/pros of swapping:

- ✓ It offers a higher degree of multiprogramming.
- ✓ Allows dynamic relocation. For example, if address binding at execution time is being used, then processes can be swap in different locations. Else in case of compile and load time bindings, processes should be moved to the same location.
- ✓ It helps to get better utilization of memory.
- ✓ Minimum wastage of CPU time on completion so it can easily be applied to a priority-based scheduling method to improve its performance.

Contiguous Memory Allocation

- In contiguous memory allocation, each process is contained in a single contiguous block of memory.
- Memory is divided into several fixed-size partitions. Each partition contains exactly one process.
- When a partition is free, a process is selected from the input queue and loaded into it.
- The free blocks of memory are known as holes.
- The set of holes is searched to determine which hole is best to allocate.

Memory Protection

Memory protection is a phenomenon by which we control memory access rights on a computer.

The main aim of it is to prevent a process from accessing memory that has not been allocated to it.

Hence prevents a bug within a process from affecting other processes, or the operating system itself, and instead results in a segmentation fault or storage violation exception being sent to the disturbing process, generally killing of process.

Memory Allocation in OS

Memory allocation is a process by which computer programs are assigned memory or space. It is of three types :

First Fit Allocation

The first hole that is big enough is allocated to the program.

Best Fit Allocation

The smallest hole that is big enough is allocated to the program.

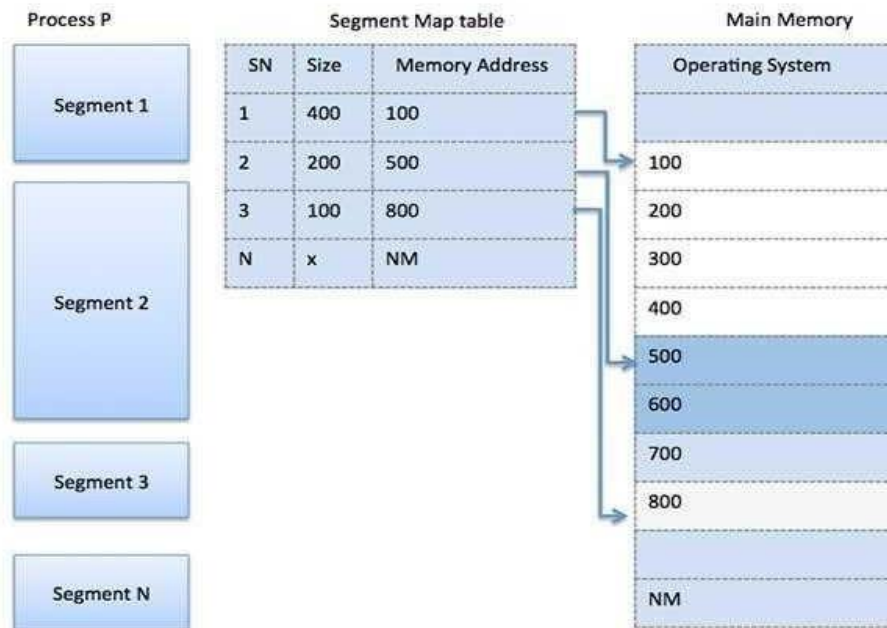
Worst Fit Allocation

The largest hole that is big enough is allocated to the program.

Segmentation

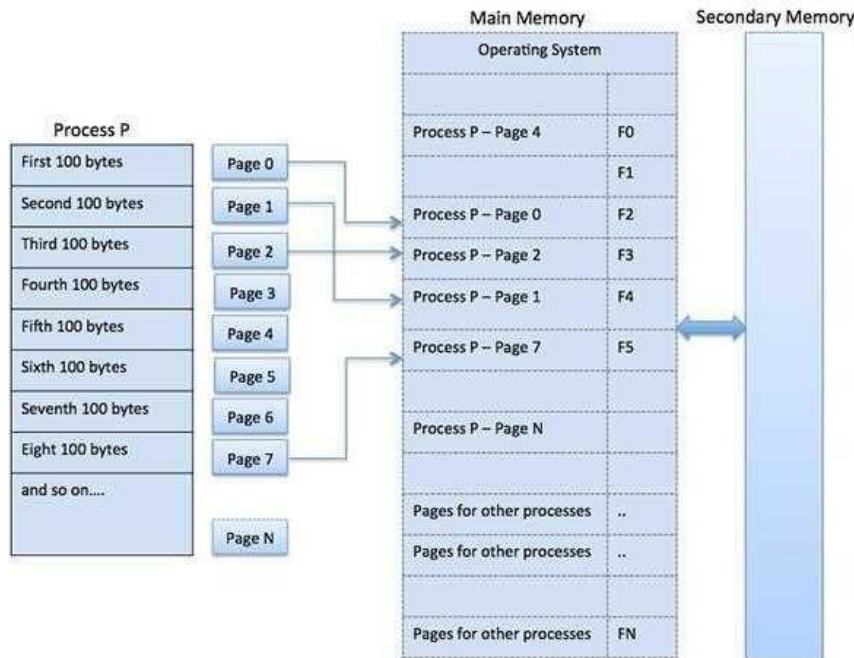
- Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions.
- Each segment is actually a different logical address space of the program.
- When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
- Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.
- A program segment contains the program's main function, utility functions, data structures, and so on.
- The operating system maintains a segment map table for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory.
- For each segment, the table stores the starting address of the segment and the length of the segment.

A reference to a memory location includes a value that identifies a segment and an offset.



Paging

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



Address Translation

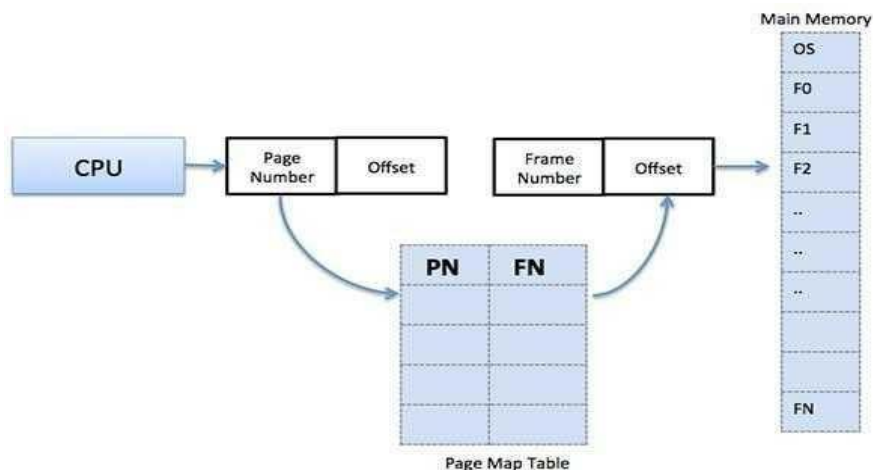
Page address is called logical address and represented by page number and the offset.

Logical Address = Page number + page offset

Frame address is called physical address and represented by a frame number and the offset.

Physical Address = Frame number + page offset

A data structure called page map table is used to keep track of the relation between a page of a process to a frame in physical memory.



- When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

- When a process is to be executed, its corresponding pages are loaded into any available memory frames.
- Suppose you have a program of 8Kb but your memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture.
- When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.
- This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Advantages and Disadvantages of Paging

Here is a list of advantages and disadvantages of paging –

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

Segmented Paging:_____

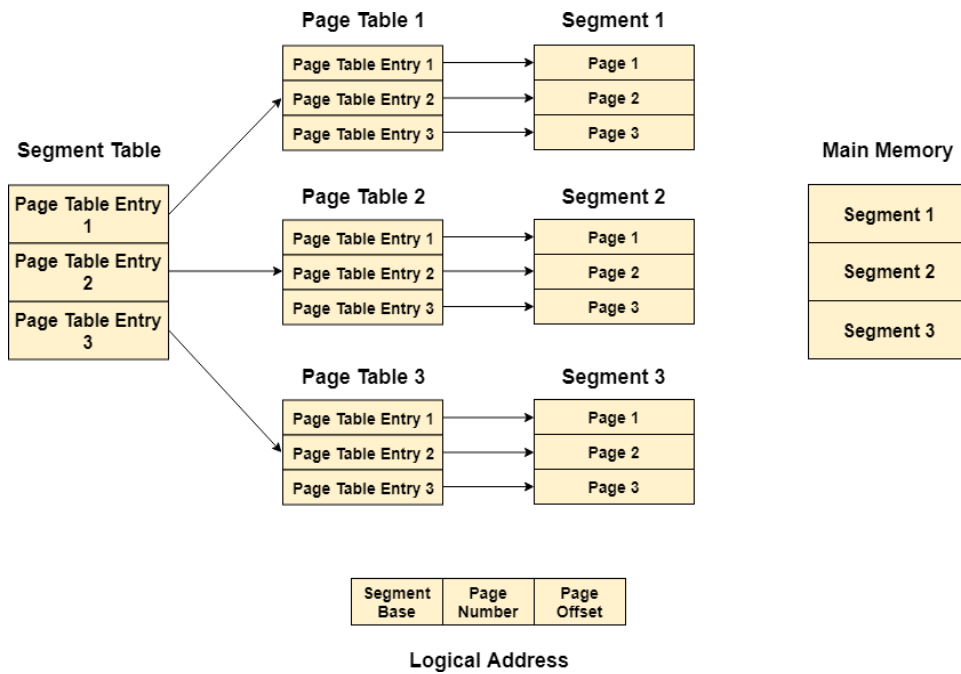
- Pure segmentation is not very popular and not being used in many of the operating systems.
- However, Segmentation can be combined with Paging to get the best features out of both the techniques.
- In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.
- Pages are smaller than segments.
- Each Segment has a page table which means every program has multiple page tables.
- The logical address is represented as Segment Number (base address), Page number and page offset.

Segment Number → It points to the appropriate Segment Number.

Page Number → It Points to the exact page within the segment

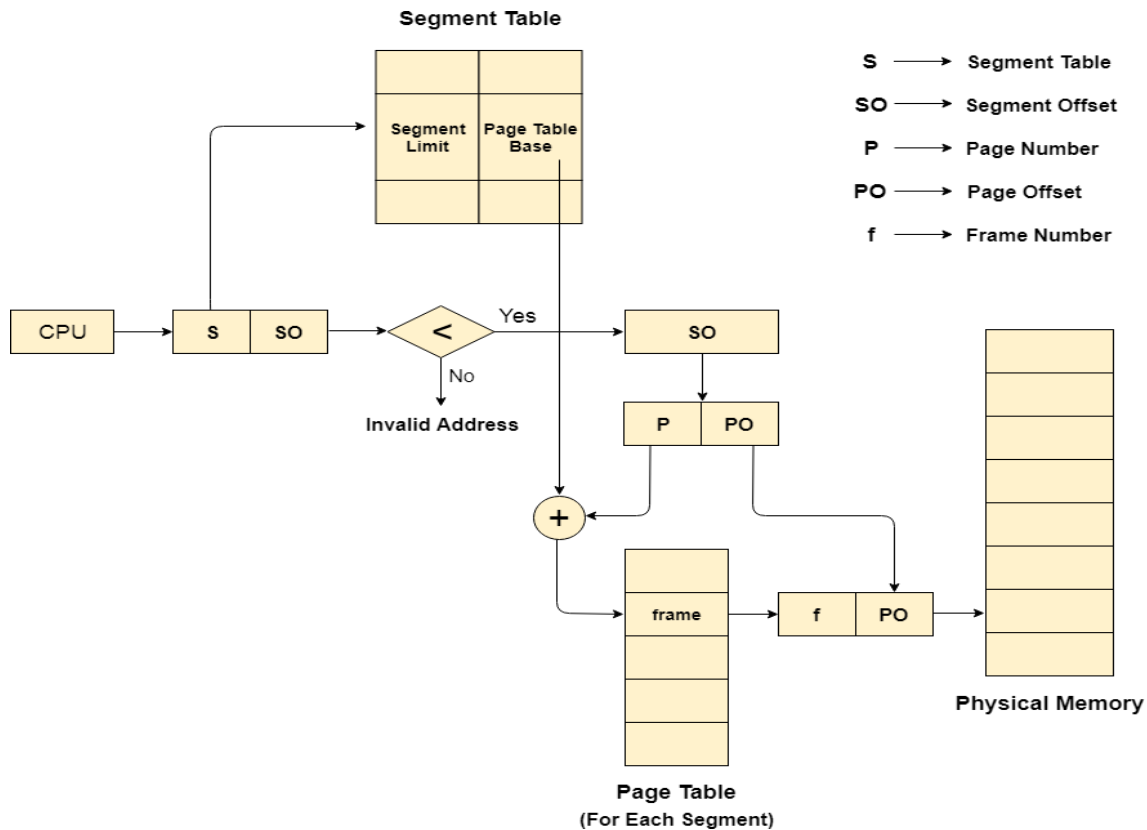
Page Offset → Used as an offset within the page frame

- Each Page table contains the various information about every page of the segment.
- The Segment Table contains the information about every segment.
- Each segment table entry points to a page table and every page table entry is mapped to one of the page within a segment.



Translation of logical address to physical address

- The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset.
- The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset.
- To map the exact page number in the page table, the page number is added into the page table base.
- The actual frame number with the page offset is mapped to the main memory to get the desired word in the page of the certain segment of the process.



Advantages of Segmented Paging

- It reduces memory usage.
- Page table size is limited by the segment size.
- Segment table has only one entry corresponding to one actual segment.
- External Fragmentation is not there.
- It simplifies memory allocation.

Disadvantages of Segmented Paging

- Internal Fragmentation will be there.
- The complexity level will be much higher as compare to paging.
- Page Tables need to be contiguously stored in the memory.

Chapter -9

VIRTUAL MEMORY

TOPICS:

- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing

Virtual Memory:

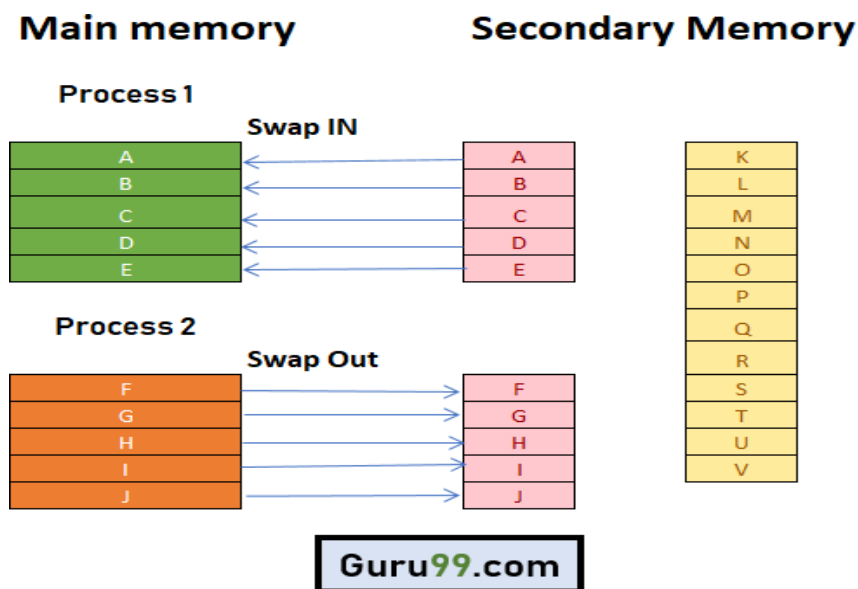
- Virtual Memory is a storage mechanism which offers user an illusion of having a very big main memory.
- It is done by treating a part of secondary memory as the main memory. In Virtual memory, the user can store processes with a bigger size than the available main memory.
- Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory. Virtual memory is mostly implemented with demand paging and demand segmentation.

Background

- Preceding sections talked about how to avoid memory fragmentation by breaking process memory requirements down into smaller bites (pages), and storing the pages non-contiguously in memory. However the entire process still had to be stored in memory somewhere.
- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
 1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
 2. Arrays are often over-sized for worst-case scenarios, and only a small fraction of the arrays are actually used in practice.
 3. Certain features of certain programs are rarely used, such as the routine to balance the federal budget. ☺

- The ability to load only the portions of processes that were actually needed (and only when they were needed) has several benefits:
 - Programs could be written for a much larger address space (virtual memory space) than physically exists on the computer.
 - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
 - Less I/O is needed for swapping processes in and out of RAM, speeding things up.

What is Demand Paging?



- A demand paging mechanism is very much similar to a paging system with swapping where processes stored in the secondary memory and pages are loaded only on demand, not in advance.
- So, when a context switch occurs, the OS never copy any of the old program's pages from the disk or any of the new program's pages into the main memory.

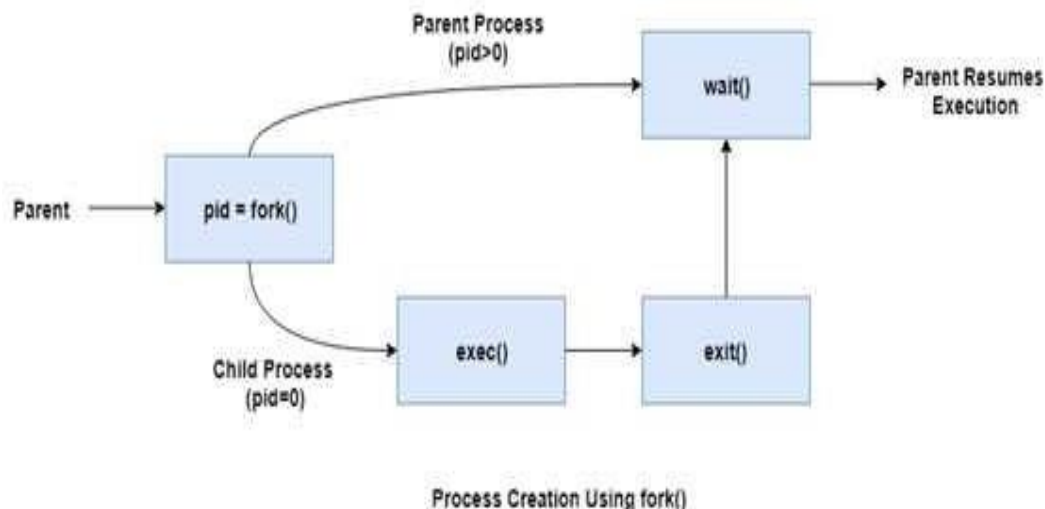
Instead, it will start executing the new program after loading the first page and fetches the program's pages, which are referenced.

- During the program execution, if the program references a page that may not be available in the main memory because it was swapped, then the processor considers it as an invalid memory reference.
- That's because the page fault and transfers send control back from the program to the OS, which demands to store page back into the memory.

Process Creation

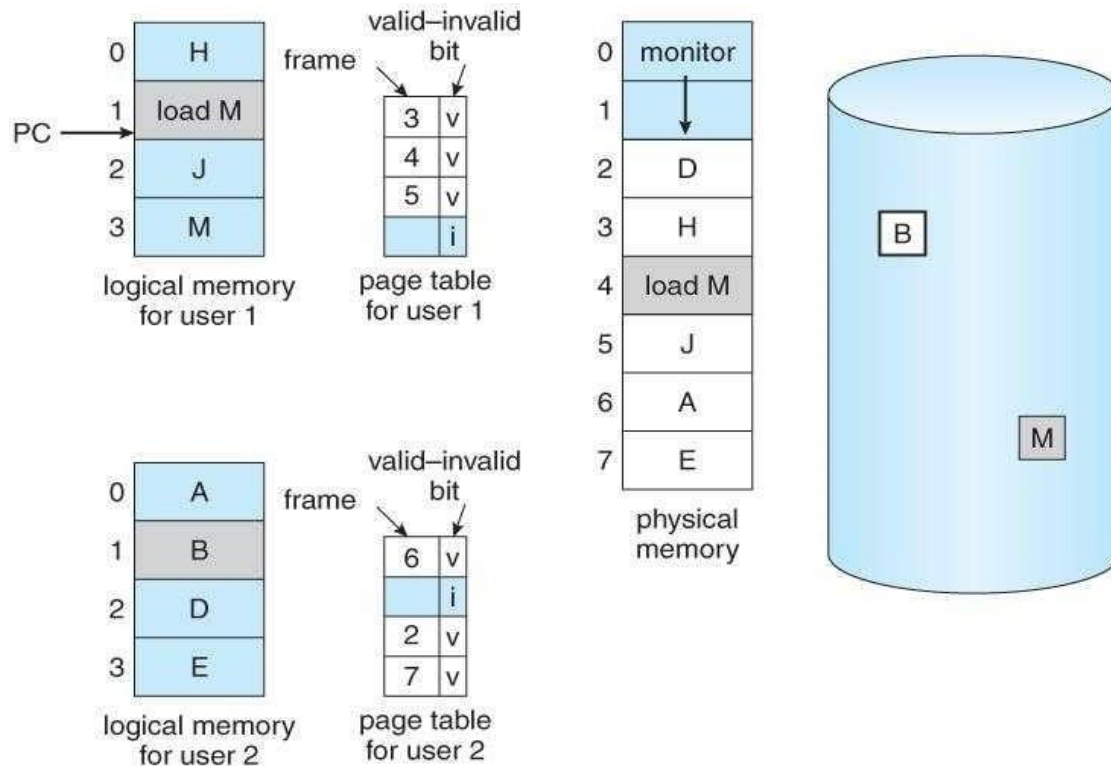
- A process may be created in the system for different operations. Some of the events that lead to process creation are as follows –
 - User request for process creation
 - System Initialization
 - Batch job initialization
 - Execution of a process creation system call by a running process
- A process may be created by another process using **fork()**. The creating process is called the parent process and the created process is the child process.
- A child process can have only one parent but a parent process may have many children.
- Both the parent and child processes have the same memory image, open files and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows –



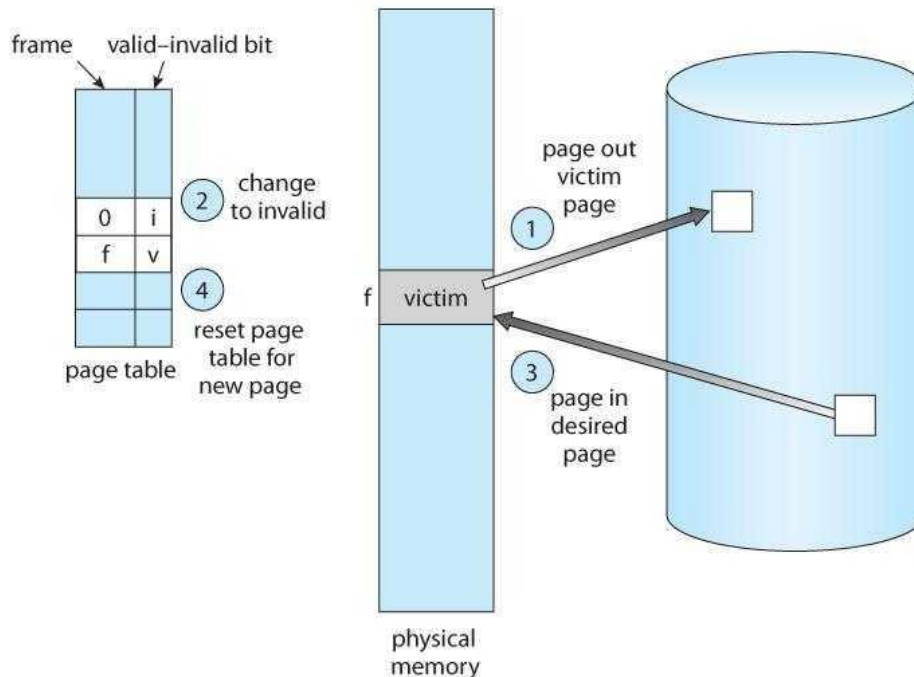
Page Replacement

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



Basic Page Replacement

- The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:
 1. Find the location of the desired page on the disk, either in swap space or in the file system.
 2. Find a free frame:
 - a. If there is a free frame, use it.
 - b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the victim frame.
 - c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
 3. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
 4. Restart the process that was waiting for this page.



- 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.

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:

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0		0	0				7	7	7
	0	0	0		3	3	3	2	2	2		1	1				1	0	0
		1	1		1	0	0	0	3	3		3	2				2	2	1

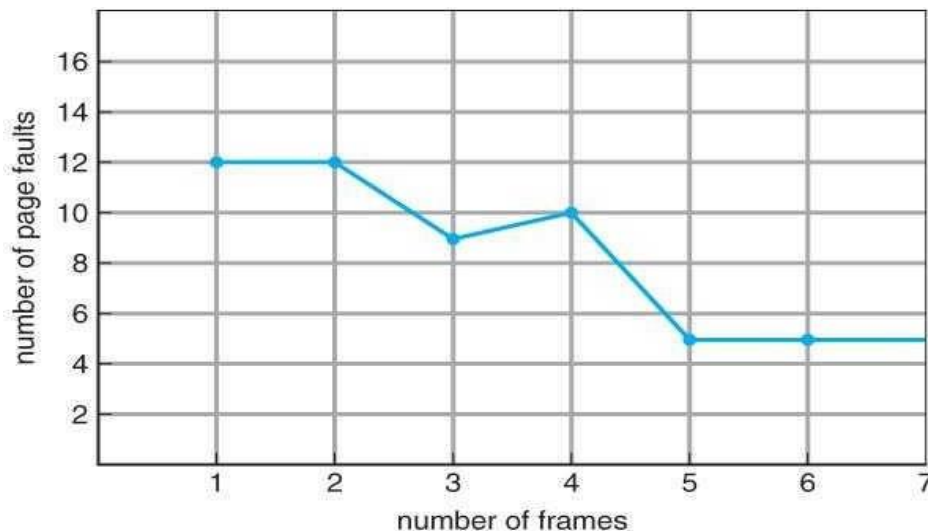
page frames

FIFO page-replacement algorithm.

- Although FIFO is simple and easy, it is not always optimal, or even efficient.

- An interesting effect that can occur with FIFO is Belady's anomaly, in which increasing the number of frames available can actually increase the number of page faults that occur! Consider, for example, the following chart based on the page sequence (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5) and a varying number of available frames.

Obviously the maximum number of faults is 12 (every request generates a fault), and the minimum number is 5 (each page loaded only once), but in between there are some interesting results:



Features:

Whenever a new page loaded, the page recently comes in the memory is removed. So, it is easy to decide which page requires to be removed as its identification number is always at the FIFO stack.

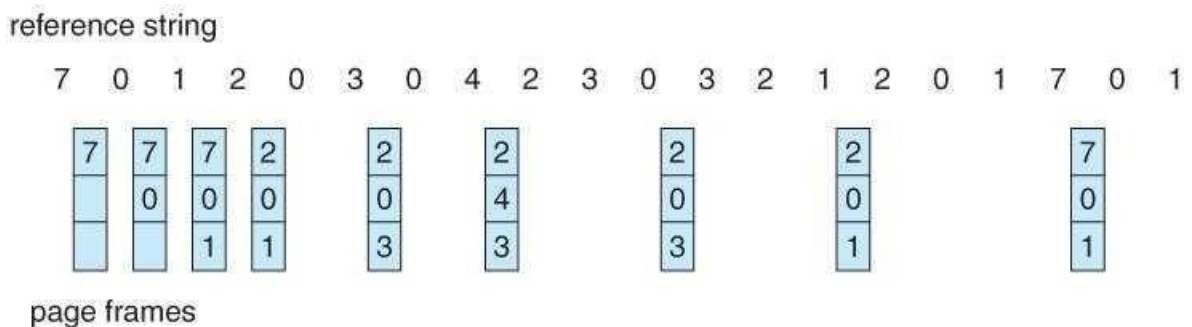
The oldest page in the main memory is one that should be selected for replacement first.

Optimal Page Replacement

- ✓ The discovery of Belady's anomaly lead to the search for an optimal page-replacement algorithm, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- ✓ Such an algorithm does exist, and is called OPT or MIN. This algorithm is simply "Replace the page that will not be used for the longest time in the future."
- ✓ For example, Figure shows that by applying OPT to the same reference string used for the FIFO example, the minimum number of possible page faults is 9.

Since 6 of the page-faults are unavoidable (the first reference to each new page), FIFO can be shown to require 3 times as many (extra) page faults as the optimal algorithm. (Note: The book claims that only the first three page faults are required by all algorithms, indicating that FIFO is only twice as bad as OPT.)

- ✓ Unfortunately OPT cannot be implemented in practice, because it requires foretelling the future, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.



Optimal page-replacement algorithm

Features:

- Optimal algorithm results in the fewest number of page faults. This algorithm is difficult to implement.
- An optimal page-replacement algorithm method has the lowest page-fault rate of all algorithms.
- This algorithm exists and which should be called MIN or OPT.

Replace the page which unlikely to use for a longer period of time. It only uses the time when a page needs to be used.

LRU Page Replacement

- The prediction behind LRU, the Least Recently Used, algorithm is that the page that has not been used in the longest time is the one that will not be used again in the near future. (Note the distinction between FIFO and LRU: The former looks at the oldest load time, and the latter looks at the oldest use time.)
- Some view LRU as analogous to OPT, except looking backwards in time instead of forwards. (OPT has the interesting property that for any reference string S and its reverse R, OPT will generate the same number of page faults for S and for R. It turns out that LRU has this same property.)

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

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		4	4	4	0		1		1		1			
	0	0	0		0		0	0	3	3		3		0		0			
		1	1		3		3	2	2	2		2		2		7			

page frames

LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it. There are two simple approaches commonly used:

1.Counters. Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value. Note that overflowing of the counter must be considered.

2.Stack. Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

Features:

- The LRU replacement method has the highest count.
- This counter is also called aging registers, which specify their age and how much their associated pages should also be referenced.
- The page which hasn't been used for the longest time in the main memory is the one that should be selected for replacement.
- It also keeps a list and replaces pages by looking back into time.

Allocation of frames in Operating System

An important aspect of operating systems, [virtual memory](#) is implemented using demand paging. Demand paging necessitates the development of a [page-replacement algorithm](#) and a **frame allocation algorithm**. Frame allocation algorithms are used if you have multiple processes; it helps decide how many frames to allocate to each process.

There are various constraints to the strategies for the allocation of frames:

- You cannot allocate more than the total number of available frames.
- At least a minimum number of frames should be allocated to each process. This constraint is supported by two reasons. The first reason is,

as less number of frames are allocated, there is an increase in the page fault ratio, decreasing the performance of the execution of the process. Secondly, there should be enough frames to hold all the different pages that any single instruction can reference.

Frame allocation algorithms –

The two algorithms commonly used to allocate frames to a process are:

Equal allocation: In a system with x frames and y processes, each process gets equal number of frames, i.e. x/y . For instance, if the system has 48 frames and 9 processes, each process will get 5 frames. The three frames which are not allocated to any process can be used as a free-frame buffer pool.

Disadvantage: In systems with processes of varying sizes, it does not make much sense to give each process equal frames. Allocation of a large number of frames to a small process will eventually lead to the wastage of a large number of allocated unused frames.

Proportional allocation: Frames are allocated to each process according to the process size.

For a process p_i of size s_i , the number of allocated frames is $a_i = (s_i/S)*m$, where S is the sum of the sizes of all the processes and m is the number of frames in the system. For instance, in a system with 62 frames, if there is a process of 10KB and another process of 127KB, then the first process will be allocated $(10/137)*62 = 4$ frames and the other process will get $(127/137)*62 = 57$ frames.

Advantage: All the processes share the available frames according to their needs, rather than equally.

Thrashing

If a process cannot maintain its minimum required number of frames, then it must be swapped out, freeing up frames for other processes.

This is an intermediate level of CPU scheduling.

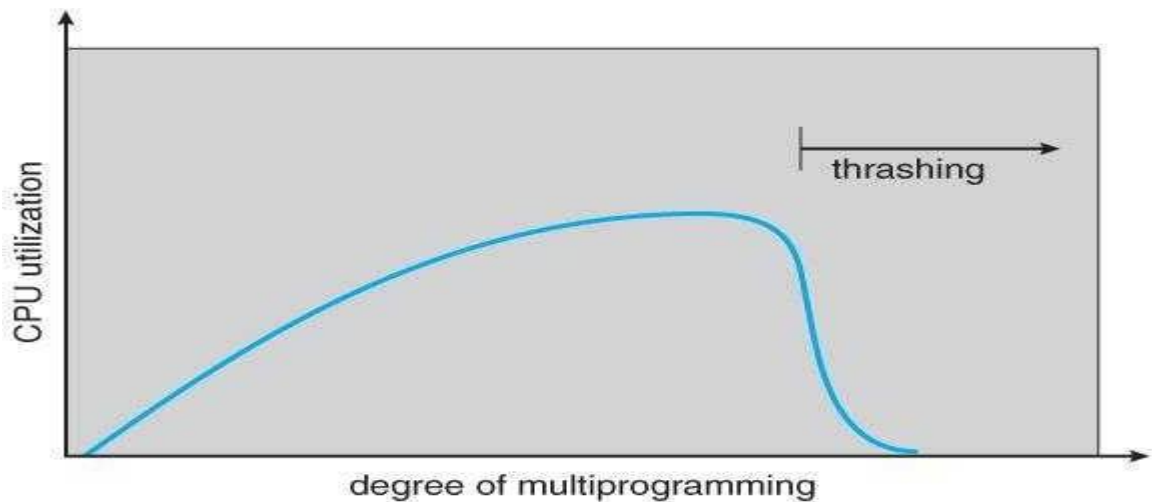
But what about a process that can keep its minimum, but cannot keep all of the frames that it is currently using on a regular basis? In this case it is forced to page out pages that it will need again in the very near future, leading to large numbers of page faults.

A process that is spending more time paging than executing is said to be thrashing.

Cause of Thrashing

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

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

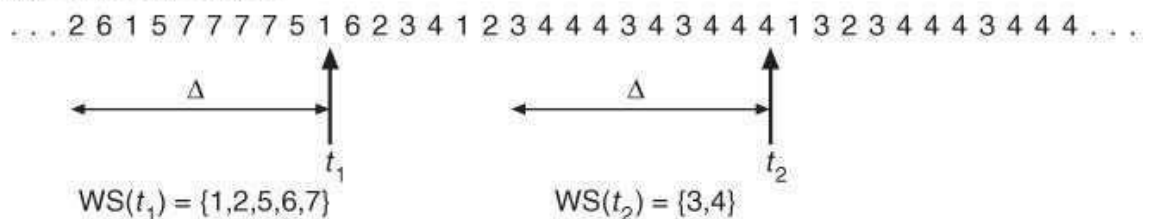


○

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.

page reference table



Chapter -10

FILE SYSTEM INTERFACE

TOPICS:

- **File Concepts**
- **Access Methods**
- **Directory Structure**
- **File System Mounting**
- **File Sharing**

File Concept

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its ...

File Attributes

Name – only information kept in human-readable form

- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring
- Information about files are kept in the directory structure, which is maintained on the disk..

File Operations

File is an abstract data type

- Create
- Write – at write pointer location
- Read – at read pointer location
- Reposition within file - seek
- Delete
- Truncate

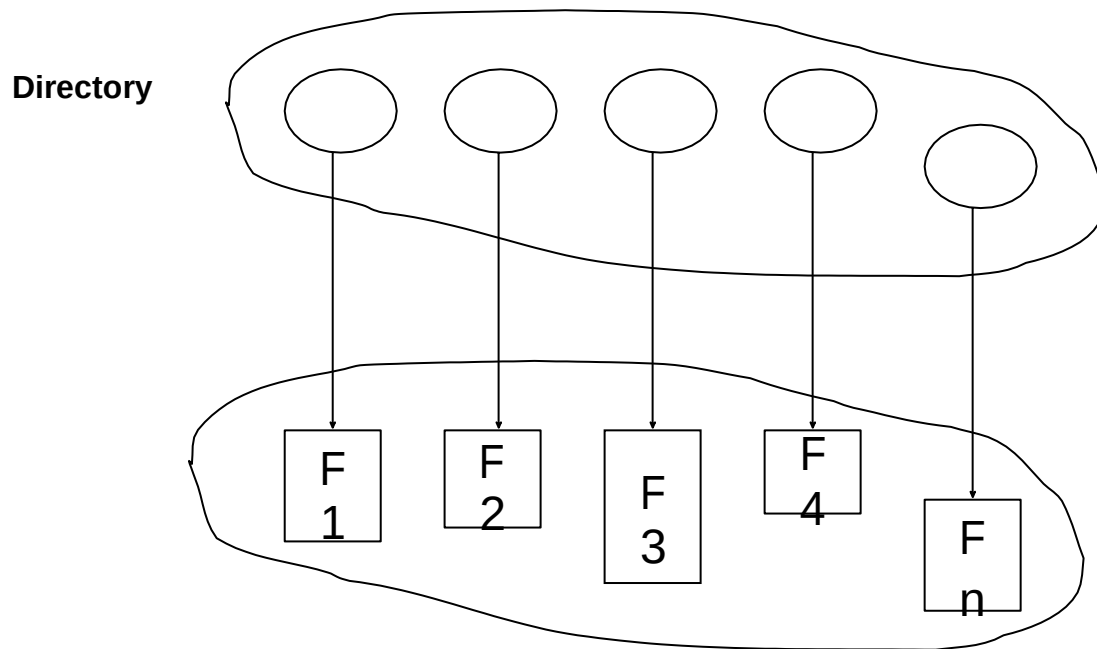
- Open(Fi) – search the directory structure on disk for entry Fi and move the content of entry to memory
- Close (Fi) – move the content of entry Fi in memory to directory structure on disk...

File Types – Name, Extension

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

Directory Structure

- A collection of nodes containing information about all files



Files

Operations Performed on Directory

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file

Directory Organization

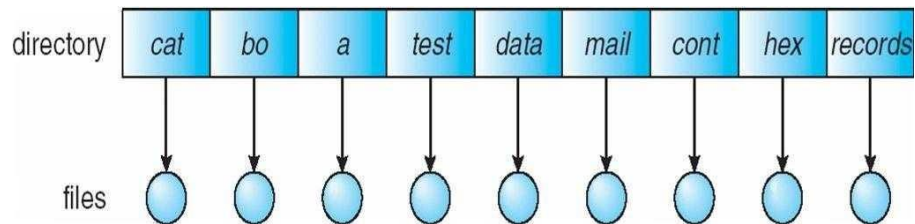
The directory is organized logically to obtain

- Efficiency – locating a file quickly
- Naming – convenient to users
 - Two users can have same name for different files
 - The same file can have several different names
- Grouping – logical grouping of files by properties, (e.g., all Java programs, all games, ...)

Single-Level Directory

A single directory for all users

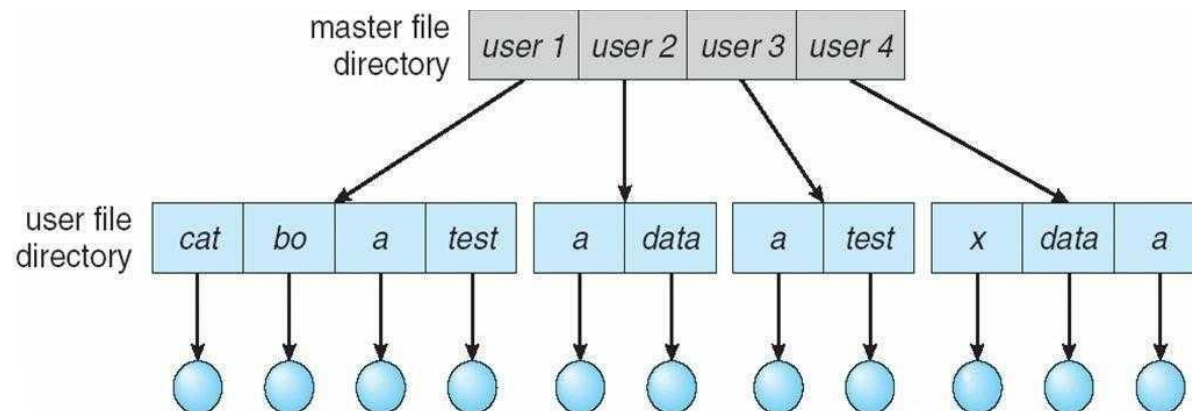
- Naming problem



- Grouping problem

Two-Level Directory

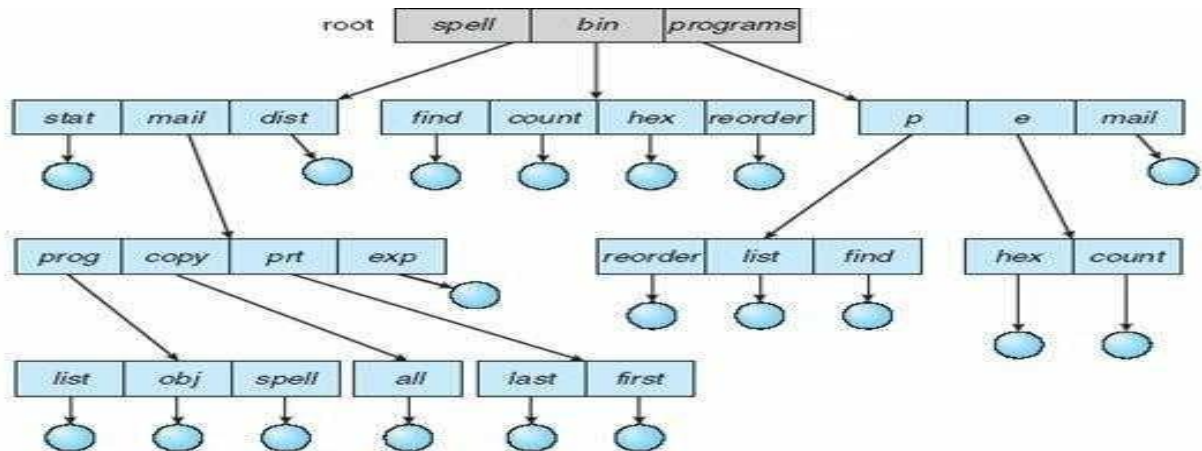
Separate directory for each user



- Pathname
- Can have the same file name for different user
- Efficient searching
- No grouping capability

Tree-Structured Directories

- Efficient searching



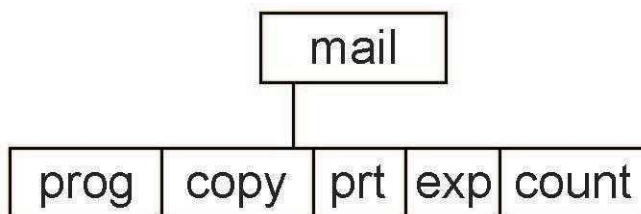
- Grouping Capability
- Current directory (working directory)
- `cd/spell/mail/prog`
 - typelist**
 - Absolute or relative path name
 - Creating a new file is done in current directory
 - Delete a file


```
rm <file-name>
```

Creating a new subdirectory is done in current directory

```
mkdir <dir-name>
```

Example: if in current directory `/mail` `mkdir count`.

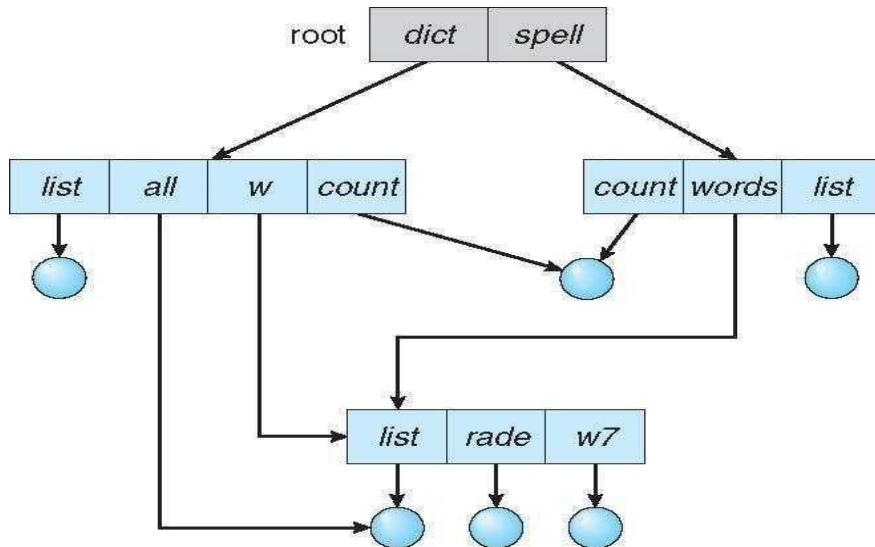


Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories

- Have shared subdirectories and files

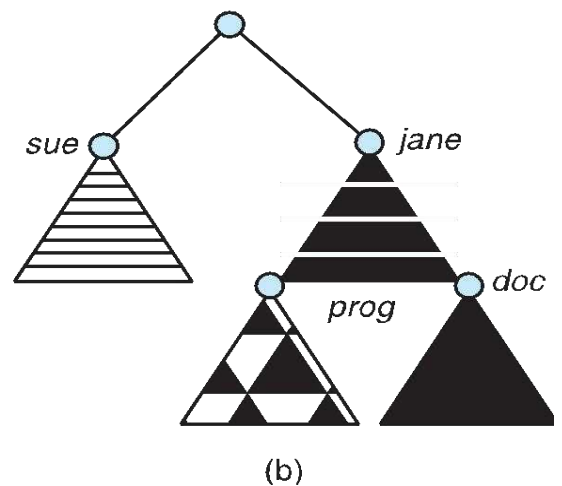
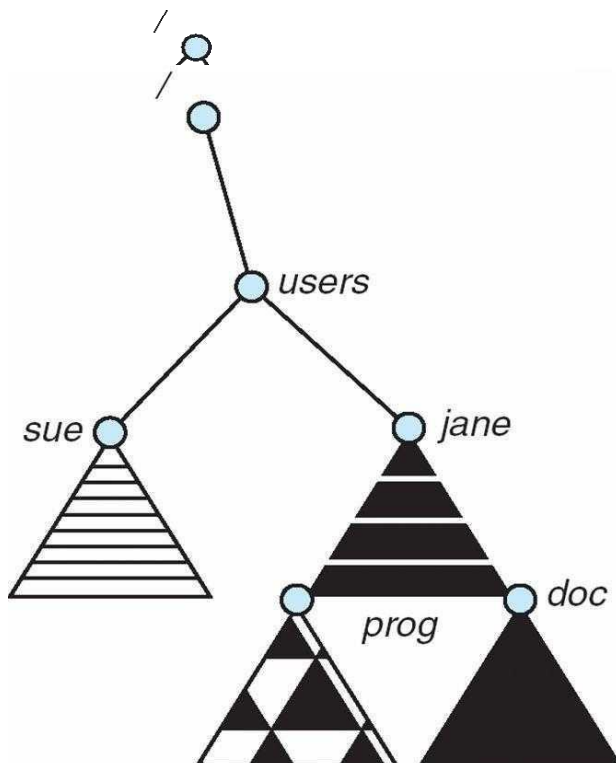
File System Mounting



File System Mounting

- A filesystem must be **mounted** before it can be accessed
- An unmounted filesystem (i.e., Fig. 11-11(b)) is mounted at a **mount point**

Mount Point



File Sharing

- Sharing of files on multi-user systems is desirable
- Sharing may be done through a protection scheme
- On distributed systems, files may be shared across a network
- Network File System (NFS) is a common distributed file-sharing method
- If multi-user system
- **User IDs** identify users, allowing permissions and protections to be per-user

Group IDs allow users to be in groups, permitting group access rights

- Owner of a file / directory
- Group of a file / directory

File Sharing – Remote File Systems

Uses networking to allow file system access between systems

- Manually via programs like FTP
- Automatically, seamlessly using distributed file systems
- Semi automatically via the world wide web

Client-server model allows clients to mount remote file systems from servers

- Server can serve multiple clients
- Client and user-on-client identification is insecure or complicated
- NFS is standard UNIX client-server file sharing protocol
- Standard operating system file calls are translated into remote calls..

Protection

- File owner/creator should be able to control:
- what can be done
- by whom
- Types of access
- **Read**
- **Write**
- **Execute**
- **Append**
- **Delete**
- **List**

Chapter -11

FILE SYSTEM IMPLEMENTATION

TOPICS:

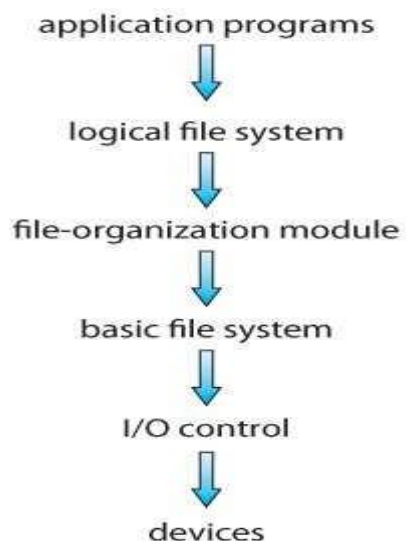
- **File System Structure**
- **File System Implementation**
- **Directory Implementation**
- **Allocation Methods**
- **Free space Management**
- **Efficiency and Performance**
- **Recovery**

File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems:
 - (1) Blocks of data can be rewritten in place, and
 - (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency.
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
- At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
- **I/O Control** consists of **device drivers**, special software programs (often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers.
- Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ports) that it listens to, and a unique set of command codes and results codes that it understands.

OPERATING SYSTEM (UNIT-3)

- **The basic file system** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block.
- Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
- The file organization module knows about files and their logical blocks, and how they map to physical blocks on the disk.
- In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
- The logical file system deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself.
- This level manages the directory structure and the mapping of file names to file control blocks, FCBs, which contain all of the meta data as well as block number information for finding the data on the disk.
- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific.
- Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 (among 40 others supported.)



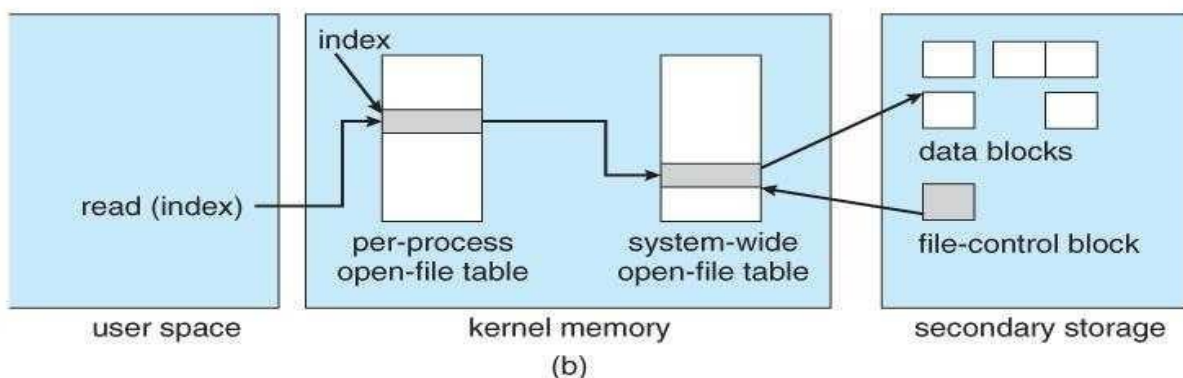
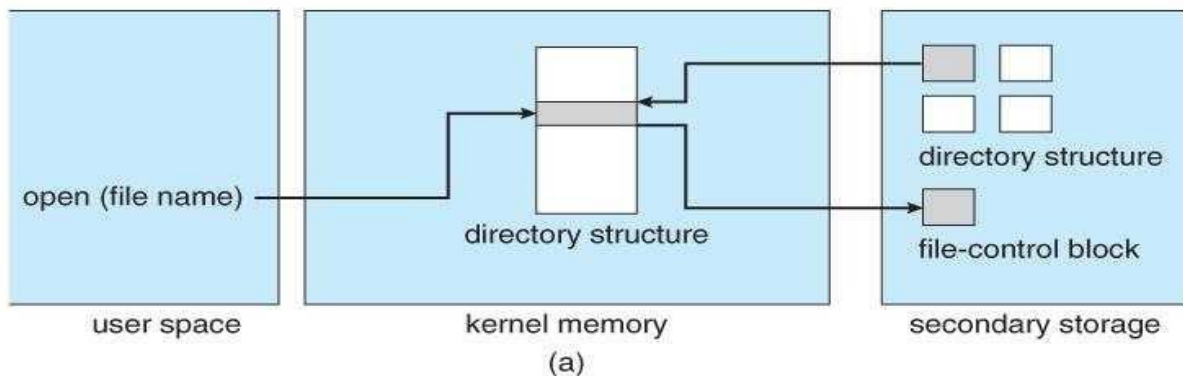
File-System Implementation

- File systems store several important data structures on the disk:
 - A **boot-control block**, (per volume) a.k.a. the **boot block** in UNIX or the **partition boot sector** in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
 - A **volume control block**, (per volume) a.k.a. the **master file table** in UNIX or the **superblock** in Windows, which contains information such as the partition table, number of blocks on each filesystem, and pointers to free blocks and free FCB blocks.
 - A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a **master file table**.
 - The **File Control Block, FCB**, (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

- There are also several key data structures stored in memory:
 - An in-memory mount table.
 - An in-memory directory cache of recently accessed directory information.
 - A **system-wide open file table**, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
 - A **per-process open file table**, containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)
- Figure 12.3 illustrates some of the interactions of file system components when files are created and/or used:

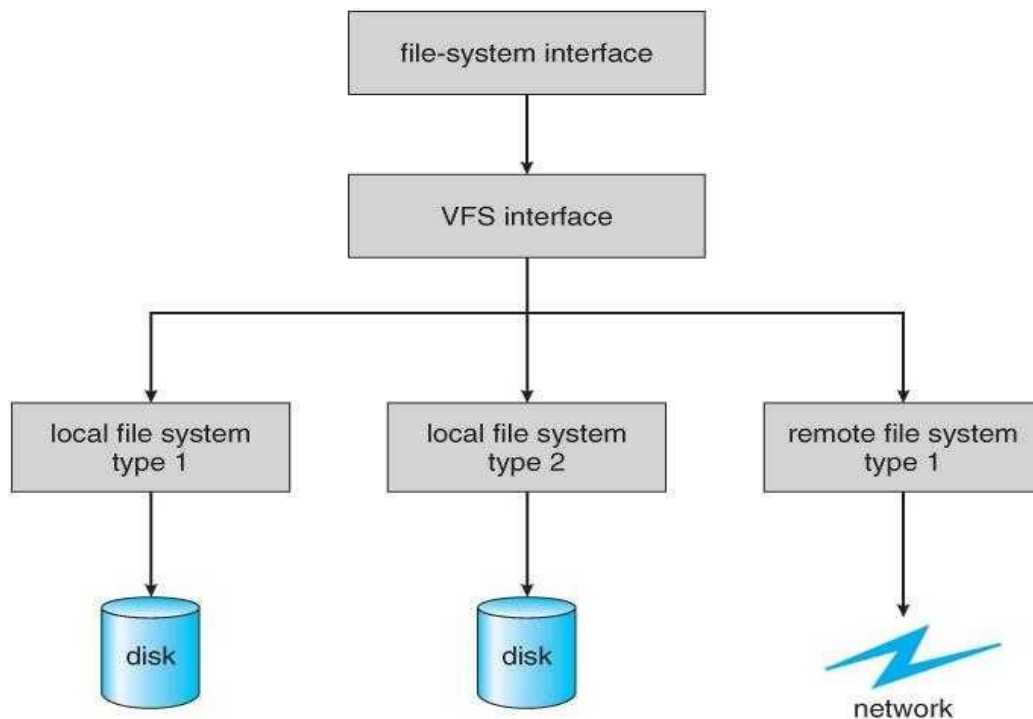
- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
- When a file is accessed during a program, the `open( )` system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the `open( )` system call. UNIX refers to this index as a **file descriptor**, and Windows refers to it as a **file handle**.
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.



Virtual File Systems

- **Virtual File Systems, VFS**, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier (`vnode`) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)

- The VFS in Linux is based upon four key object types:
 - The [inode](#) object, representing an individual file
 - The [file](#) object, representing an open file.
 - The [superblock](#) object, representing a filesystem.
 - The [dentry](#) object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. See `/usr/include/linux/fs.h` for full details. Common operations provided include `open()`, `read()`, `write()`, and `mmap()`.



Directory Implementation

- Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

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.

Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented *in addition to* a linear or other structure

Allocation Methods

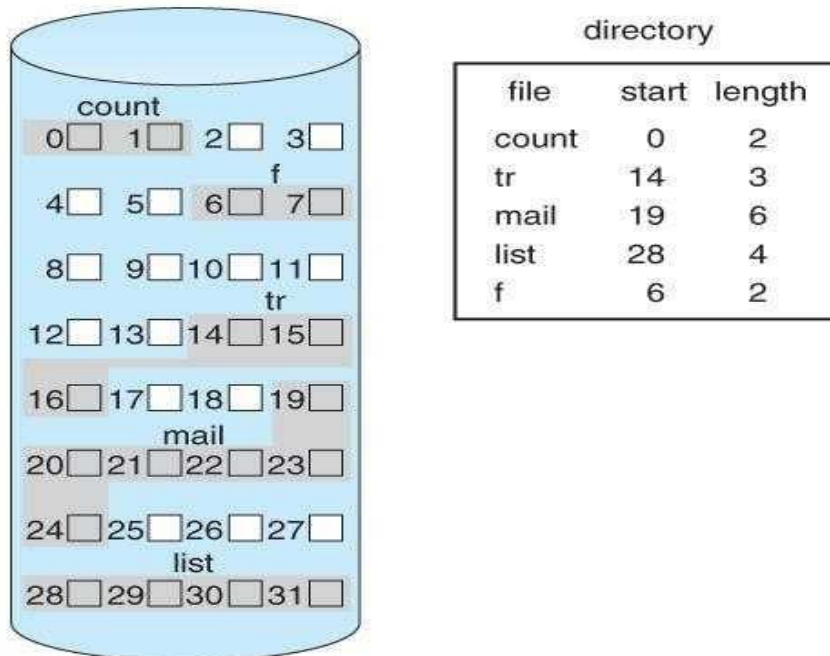
- There are three major methods of storing files on disks: contiguous, linked, and indexed.

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.)

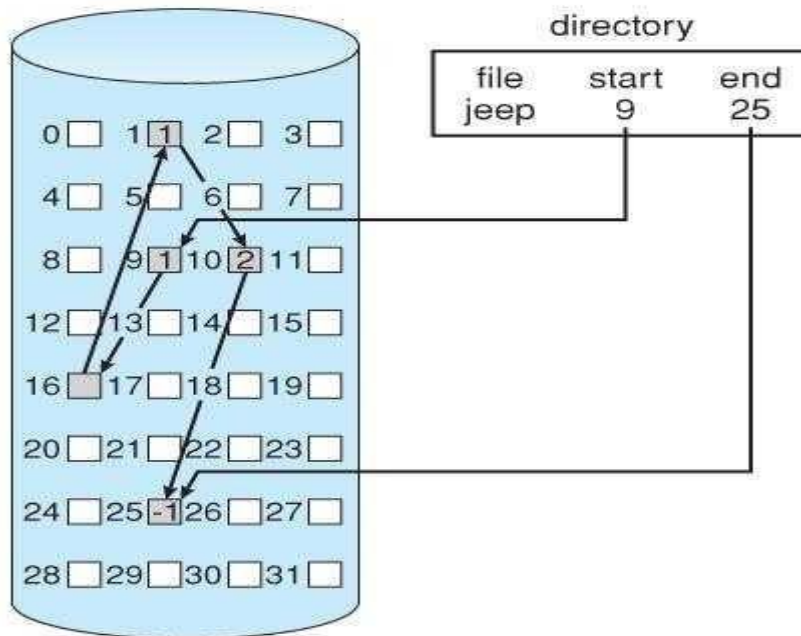
OPERATING SYSTEM (UNIT-3)

- The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.
- (Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process.)
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
 - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
 - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
 - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
- 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.



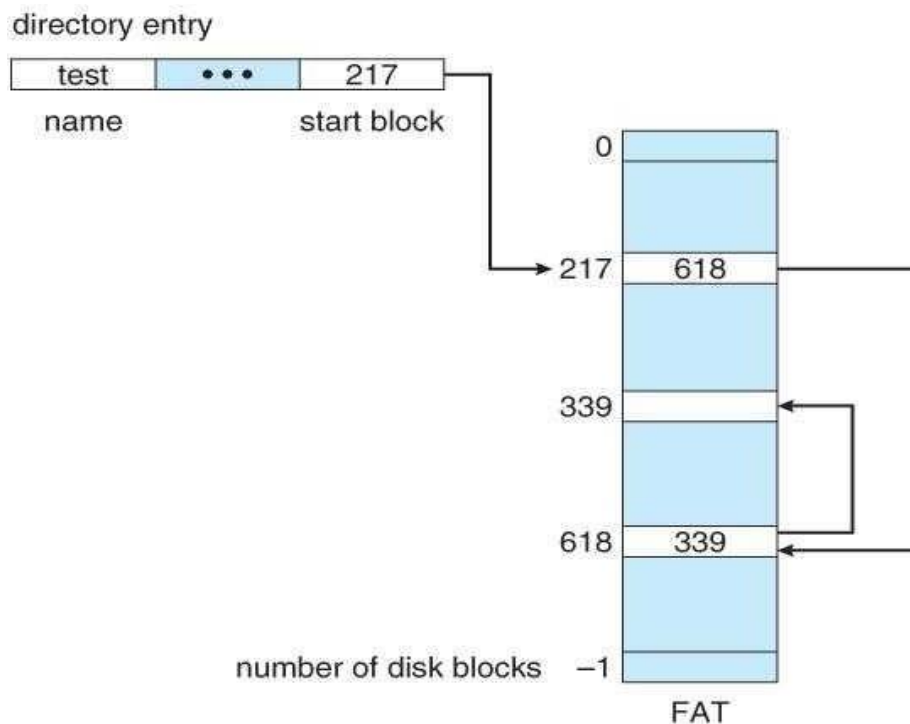
Linked Allocation

- 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.



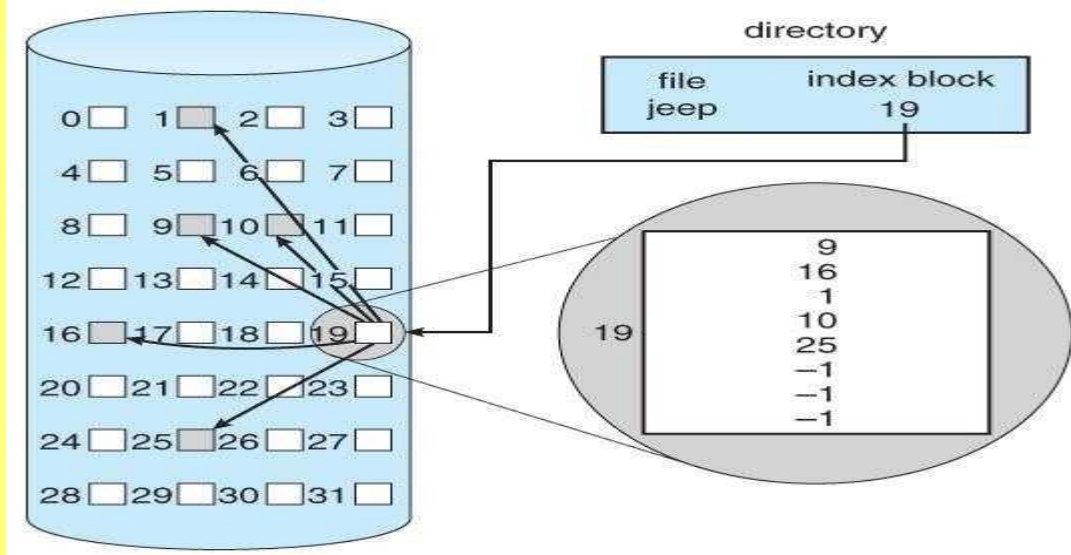
Linked allocation of disk space.

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.



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

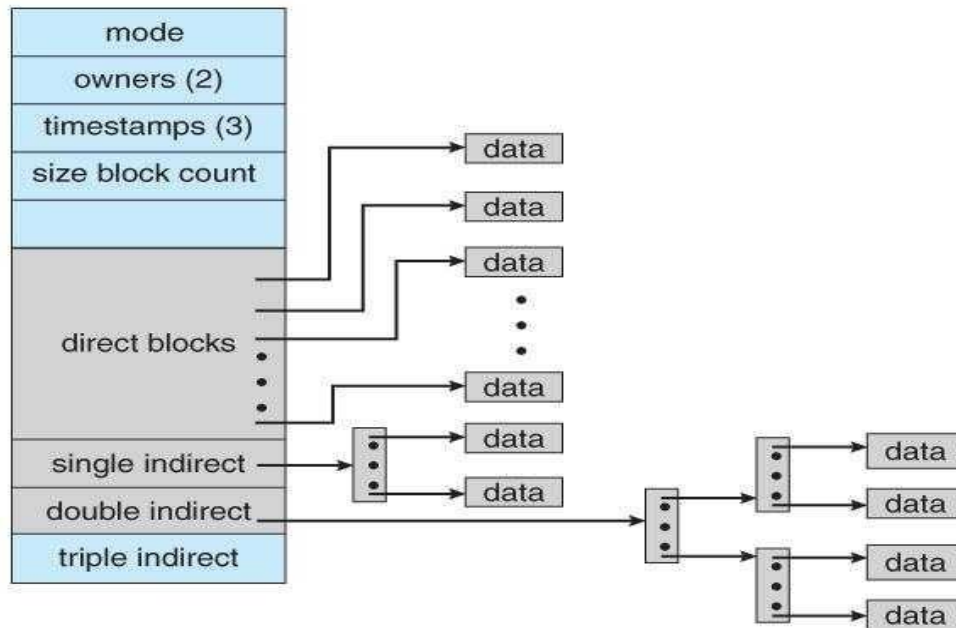


or storing them in a FAT table.

Indexed allocation of disk space.

- Some disk space is wasted (relative to linked lists or FAT tables) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:
 - **Linked Scheme** - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
 - **Multi-Level Index** - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
 - **Combined Scheme** - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. (See below.) The advantage of this scheme is that for small files

(which many are), the data blocks are readily accessible (up to 48K with 4K block sizes); files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached), and huge files are still accessible using a relatively small number of disk accesses (larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers.)



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 fi
- 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.

Free-Space Management

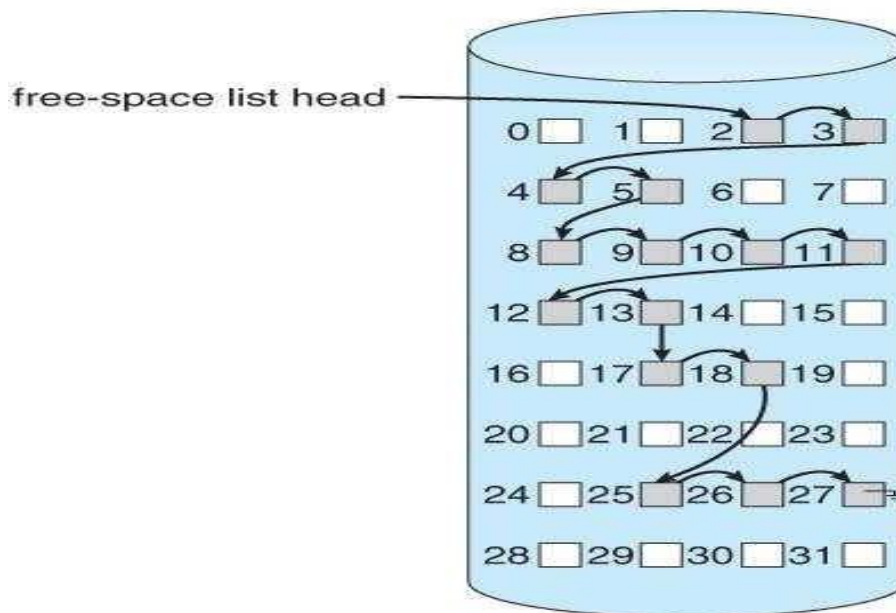
- Another important aspect of disk management is keeping track of and allocating free space.

Bit Vector

- One simple approach is to use a bit vector, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap.
(For example.)

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.



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.

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.)

Space Maps

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into (hundreds of) metaslabs of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

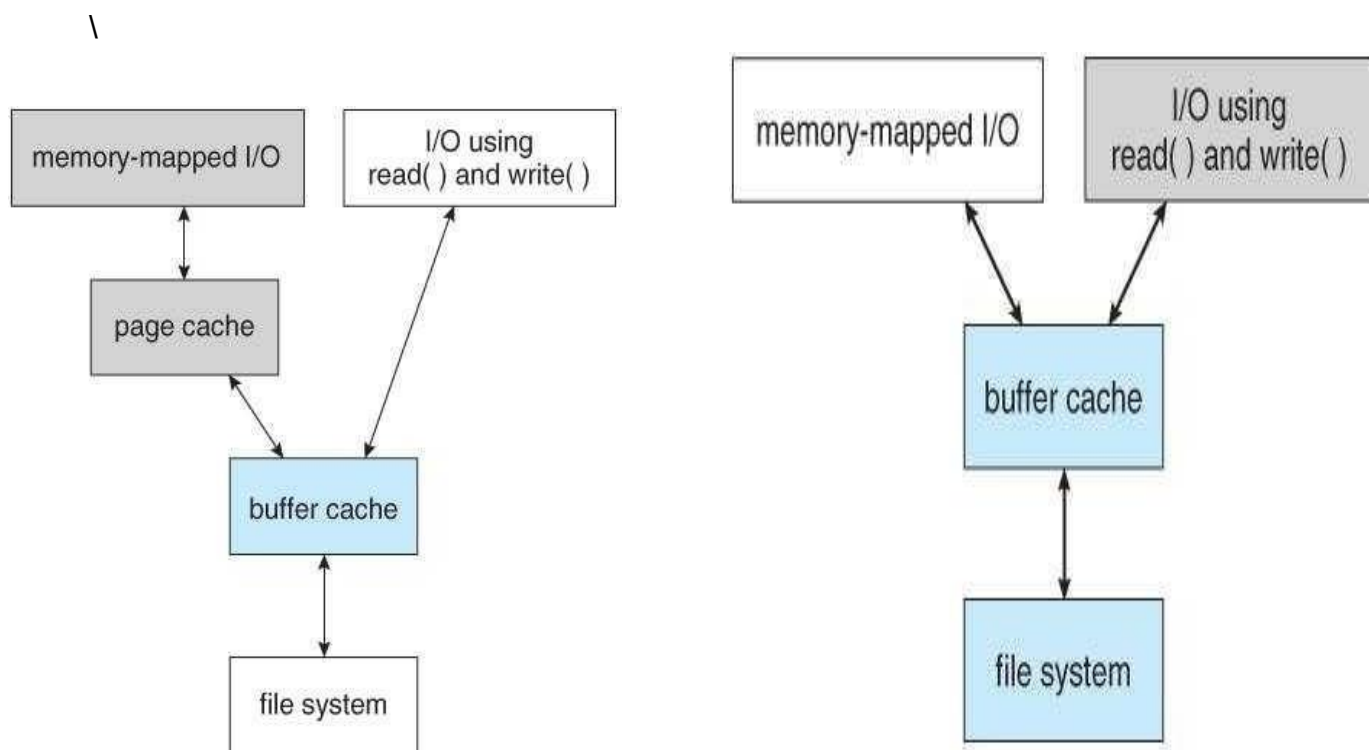
Efficiency and Performance

Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory (e.g. last access time), the more often the directory blocks have to be re-written.
- As technology advances, addressing schemes have had to grow as well.
- Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. (The mass required to store 2^{128} bytes with atomic storage would be at least 272 trillion kilograms!)
- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.

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 OS 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 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.



OPERATING SYSTEM (UNIT-3)

- 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:
 - **Free-behind** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
 - **Read-ahead** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of

transferring data from the disk controller memory into motherboard main memory.

- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times.
- 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.

Recovery

Consistency Checking

- The storing of certain data structures (e.g. directories and inodes) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.
- A Consistency Checker (fsck in UNIX, chkdsk or scandisk in Windows) is often run at boot time or mount time, particularly if a file system was not closed down properly. Some of the problems that these tools look for include:
 - Disk blocks allocated to files and also listed on the free list.
 - Disk blocks neither allocated to files nor on the free list.
 - Disk blocks allocated to more than one file.
 - The number of disk blocks allocated to a file inconsistent with the file's stated size.
 - Properly allocated files / inodes which do not appear in any directory entry.
 - Link counts for an inode not matching the number of references to that inode in the directory structure.
 - Two or more identical file names in the same directory.
 - Illegally linked directories, e.g. cyclical relationships where those are not allowed, or files/directories that are not accessible from the root of the directory tree.
 - Consistency checkers will often collect questionable disk blocks into new files with names such as chk00001.dat. These files may contain

valuable information that would otherwise be lost, but in most cases they can be safely deleted, (returning those disk blocks to the free list.)

- UNIX caches directory information for reads, but any changes that affect space allocation or metadata changes are written synchronously, before any of the corresponding data blocks are written to.

Log-Structured File Systems

Log-based transaction-oriented (a.k.a. journaling) filesystems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:

- All metadata changes are written sequentially to a log.
- A set of changes for performing a specific task (e.g. moving a file) is a transaction.
- As changes are written to the log they are said to be committed, allowing the system to return to its work.
- In the meantime, the changes from the log are carried out on the actual filesystem, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.
- When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
- At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed.
- From the log, the remaining transactions can be completed,

or if the transaction was aborted, then the partially completed changes can be undone.

NFS

- NFS is an abbreviation of the **Network File System**. It is a protocol of a distributed file system. This protocol was developed by the **Sun Microsystems** in the year of 1984.

OPERATING SYSTEM (UNIT-3)

- It is an architecture of the client/server, which contains a client program, server program, and a protocol that helps for communication between the client and server.
- It is that protocol which allows the users to access the data and files remotely over the network. Any user can easily implement the NFS protocol because it is an open standard.
- Any user can manipulate files as same as if they were on like other protocols. This protocol is also built on the ONC RPC system.
- This protocol is mainly implemented on those computing environments where the centralized management of resources and data is critical.
- It uses the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) for accessing and delivering the data and files.
- Network File System is a protocol that works on all the networks of IP-based. It is implemented in that client/server application in which the server of NFS manages the authorization, authentication, and clients.
- This protocol is used with Apple Mac OS, Unix, and Unix-like operating systems such as Solaris, Linux, FreeBSD, AIX.

The NFS Protocol

Implemented as a set of remote procedure calls (RPCs):

- Searching for a file in a directory
- REading a set of directory entries
- Manipulating links and directories
- Accessing file attributes
- Reading and writing files

UNIT- 4

Chapter -12

I/O SYSTEMS

TOPICS:

- I/O Hardware
- Application I/O Interface
- Kernel I/O Subsystem
- Transforming I/O to Hardware Operations
- STREAMS
- Performance

Overview:

- Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation.
- (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
I/O Subsystems must contend with two trends:
 - (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and
 - (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.**Device drivers** are modules that can be plugged into an OS to handle a particular device or category of similar devices.

1.) I/O Hardware:

- Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks,tapes), transmission devices (network cards, modems), and human interface devices (screen, keyboard, mouse).
- A device communicates with a computer system by sending signals over a cable or even through the air.
- VGThe device communicates with the machine via a connection point(or port), for example, a serial port. If one or more devices use a common set of wires, the connection is called a **bus**.
- A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.

- I/O devices can be roughly categorized as storage, communications, user-interface, and other.
- Devices communicate with the computer via signals sent over wires or through the air. Devices connect with the computer via ports, e.g. a serial or parallel port. A common set of wires connecting multiple devices is termed a bus.
- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
- Figure below illustrates three of the four bus types commonly found in a modern PC:
- The **PCI bus** connects high-speed high-bandwidth devices to the memory subsystem. The **expansion bus** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering).
- The **SCSI bus** connects a number of SCSI devices to a common SCSI controller.
- A **daisy-chain bus**, (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

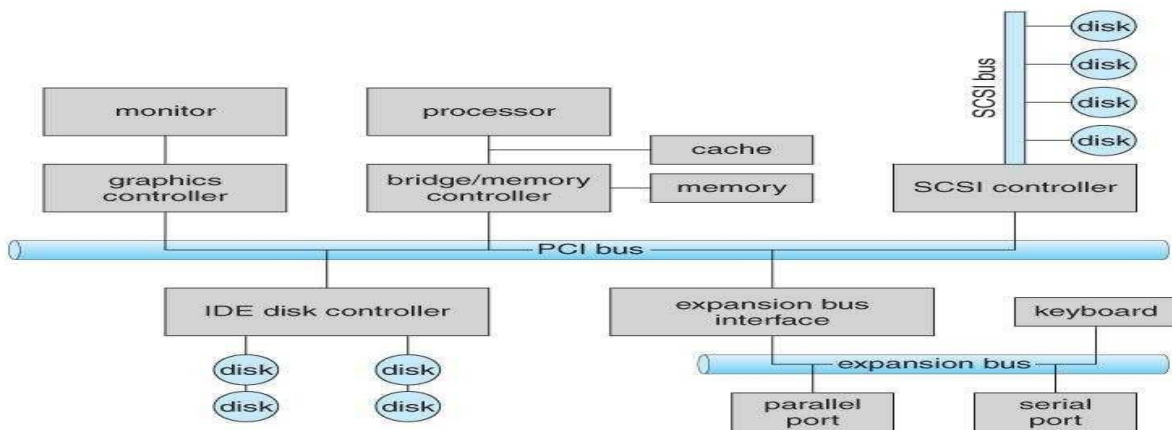


Figure - A typical PC bus structure.

One way of communicating with devices is through **registers** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:

1. **The data-in register** is read by the host to get input from the device.
2. **The data-out register** is written by the host to send output.
3. The **status register** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
4. **The control register** has bits written by the host to issue commands or to change

settings of the device such as parity checking, word length, or full- versus half-duplex operation.

Figure shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure - Device I/O port locations on PCs (partial);-

1.1 Polling:

- One simple means of device handshaking involves polling: The host repeatedly checks the busy bit on the device until it becomes clear.
- The host writes a byte of data into the data-out register, and sets the write bit in the command register (in either order).\
- The host sets the command ready bit in the command register to notify the device of the pending command.
- When the device controller sees the command-ready bit set, it first sets the busy bit.
- Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
- The device controller then clears the error bit in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.
- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer.
- It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

1.2.) Interrupts:

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an interrupt-request line that is sensed after every instruction. A device's controller raises an interrupt by asserting a signal on the interrupt request line.
- The CPU then performs a state save, and transfers control to the interrupt handler routine at a fixed address in memory. (The CPU catches the interrupt and dispatches the interrupt handler).
- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return control to the CPU. (The interrupt handler clears the interrupt by servicing the device).
- (**Note** that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing).

Figure illustrates the interrupt-driven I/O procedure:

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
- The need to defer interrupt handling during critical processing, the need to determine which interrupt handler to invoke, without having to poll all devices to see which one needs attention, and 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.

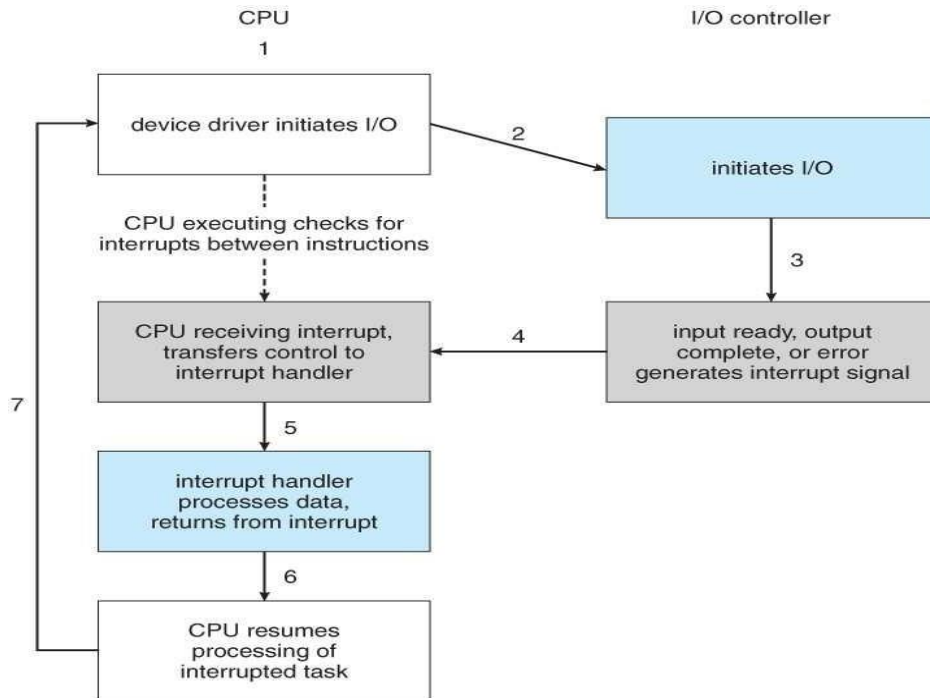


Figure - Interrupt-driven I/O cycle.

- Most CPUs now have two interrupt-request lines: One that is non-maskable for critical error conditions and one that is maskable, that the CPU can temporarily ignore during critical processing.
- The interrupt mechanism accepts an address, which is usually one of a small set of numbers for an offset into a table called the interrupt vector.
- This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be interrupt chained.
- Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are **nonmaskable** and reserved for serious hardware and other errors.
- **Maskable** interrupts, including normal device I/O interrupts begin at interrupt 32.
- Modern interrupt hardware also supports interrupt priority levels, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

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

Fig: Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
 - During operation, devices signal errors or the completion of commands via interrupts.
 - Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
 - The scheduler sets a hardware timer before transferring control over to a user process.
When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
 - The scheduler does a state-restore of a different process before resetting the timer and issuing the return-from-interrupt instruction.
 - A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run.
 - When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context

switch it back onto the CPU.)

System calls are implemented via software interrupts, a.k.a. traps.

- When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt.(E.g. 21 hex in DOS.)
- The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.
- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves two interrupts:
A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
- A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers.

This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

1.3.) Direct Memory Access:

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the Direct Memory Access, **DMA, Controller**.
- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.

- A simple DMA controller is a standard component in modern PCs, and many bus-mastering I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.

DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as Direct Virtual Memory Access, DVMA, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation figure showing below illustrates the DMA process.

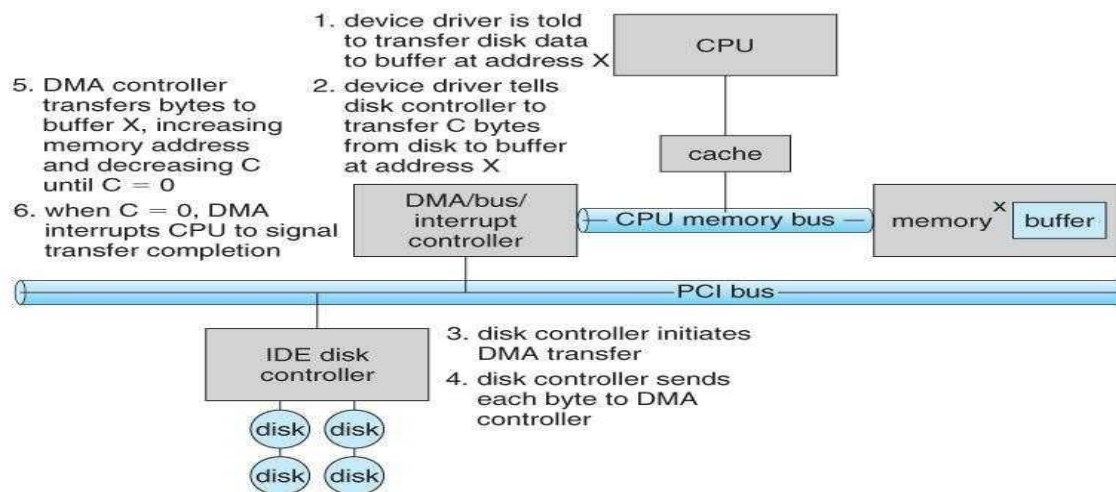


Figure 13.5 - Steps in a DMA transfer.

The main concepts:

- A bus
- A controller
- An i/o port and its registers
- The handshaking relationship between the host and a device controller

- The offloading of the work to a DMA controller for large transfer.

2.) Application I/O Interface:--

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into **device drivers**, while application layers are presented with a common interface for all (or at least large general categories of) devices.

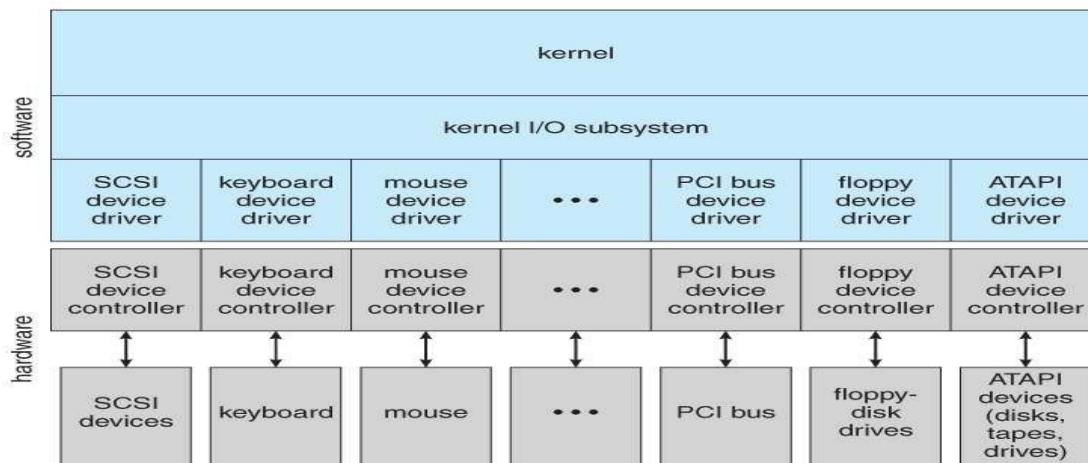


Fig: A kernel I/O structure.

Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer. Most of also have an **escape, or back door**, which allows applications to send commands directly to device drivers if needed. In UNIX this is the **ioctl()** system call (I/O Control).

ioctl() takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 13.7 - Characteristics of I/O devices.

- **Character-stream or block:** A Character-stream device transfers bytes one by one, where as block device transfers a block of bytes as a unit.
- **Sequential or random-access:** A sequential device transfers data in affixed order determined by the device, where as the user of random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous:** A Synchronous Device Is One That Performs Data Transfers With Predictable Response Times. An asynchronous Device Exhibits Irregular Or Unpredictable Response Times.
- **Sharable or dedicated:** A sharable device can be used concurrently by several process or threads; a dedicated device cannot.
- **Speed of operation:** Device Speeds Range From A Few Bytes Per Second To A Few Gigabytes Per Second,
- **Read-write, read only, or write only:** some devices perform both input and output, but others support only one data direction.

2.1 Block And Character 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.

2.2 Network Devices.

- Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
- One common and popular interface is the socket interface, which acts like a cable or pipeline connecting two networked entities.
- Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
- The select() system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

2.3 Clocks and Timers.

- Three types of time services are commonly needed in modern systems:
- Get the current time of day.
- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.
- Unfortunately time operations are not standard across all systems.

✚ **A programmable interrupt timer, PIT** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.

✚ The scheduler uses a PIT to trigger interrupts for ending time slices. The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.

✚ Networks use PIT to abort or repeat operations that are taking too long to complete. I.e resending packets if an acknowledgement is not received before the timer goes off.

✚ More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.

✚ On most systems the system clock is implemented by counting interrupts generated by the PIT.

- ✚ Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time.
- ✚ An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

2.4 Blocking and Nonblocking I/O:

- Some control over how the wait for I/O to complete is accommodated is available to the programmer of user applications.
- Most I/O requests are considered blocking requests, meaning that control does not return to the application until the I/O is complete.
- The delayed from systems calls such read() and write() can be quite long.
- Using systems calls that block is sometimes called synchronous programming.

In most cases, the wait is not really a problem because the program can not do anything else until the I/O is finished. However, in cases such as network programming with multiple clients or with graphical user interface programming, the program may wish to perform other activity as it continues to wait for more data or input from users.

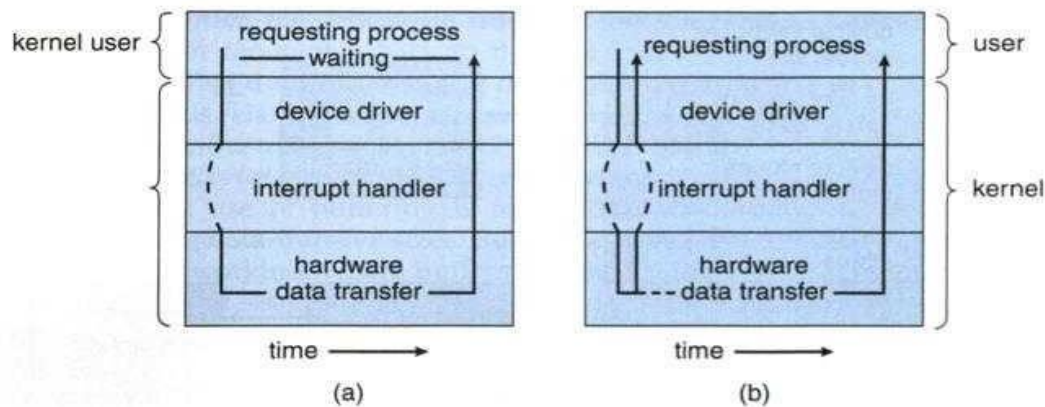
One solution for these situations is to use multiple threads so that one part of the program is not waiting for unrelated I/O to complete. Another alternative is to use asynchronous programming techniques with nonblocking system calls.

An asynchronous call returns immediately, without waiting for the I/O to complete. The completion of the I/O is later communicated to the application either through the setting of some variable in the application or through the triggering of a signal or call-back routine that is executed outside the linear control flow of the application.

Blocking I/O system calls (a) do not return until the I/O is complete. Nonblocking I/O system calls return immediately. The process is later notified when the I/O is complete.

A good example of nonblocking behavior is the select() system call for network sockets. Using select(), an application can monitor several resources at the same time and can also poll for network activity without blocking.

The select() system call identifies if data is pending or not, then read() or write() may be used knowing that they will complete immediately.



Two I/O methods: A) Synchronous and b) Asynchronous.

3. Kernel I/O Subsystem:

Kernels provide many services related to I/O. Several services—Scheduling, buffering, caching, spooling, device reservation, and error handling are provided by the kernel's I/O subsystem and build on the hardware and device- driver infrastructure.

I/O

Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part of request scheduling. Buffering and caching can also help, and can allow for more flexible scheduling options. On systems with many devices, separate request queues are often kept for each device:

Scheduling:

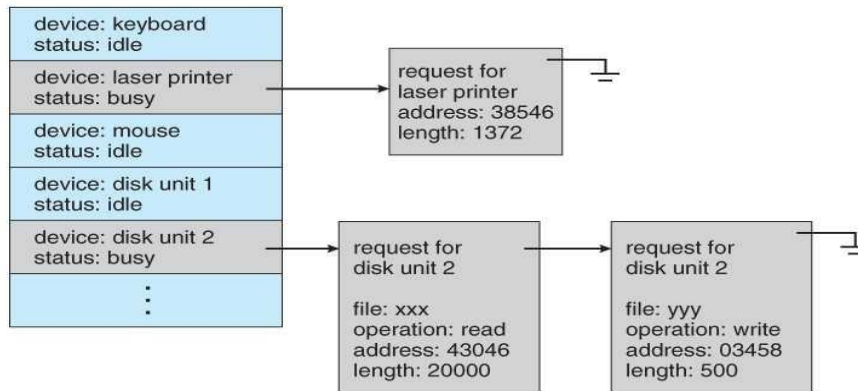


Figure 13.9 - Device-status table.

3.2 Buffering:

A Buffer is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering of I/O is performed for 3 major reasons:

1.Speed differences between two devices. See Figure below A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once.

So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full.

This is known as double buffering. (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen.

This prevents the user from ever seeing any half-finished screen images.)

2.Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.

3.To support copy semantics. For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer.

Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.

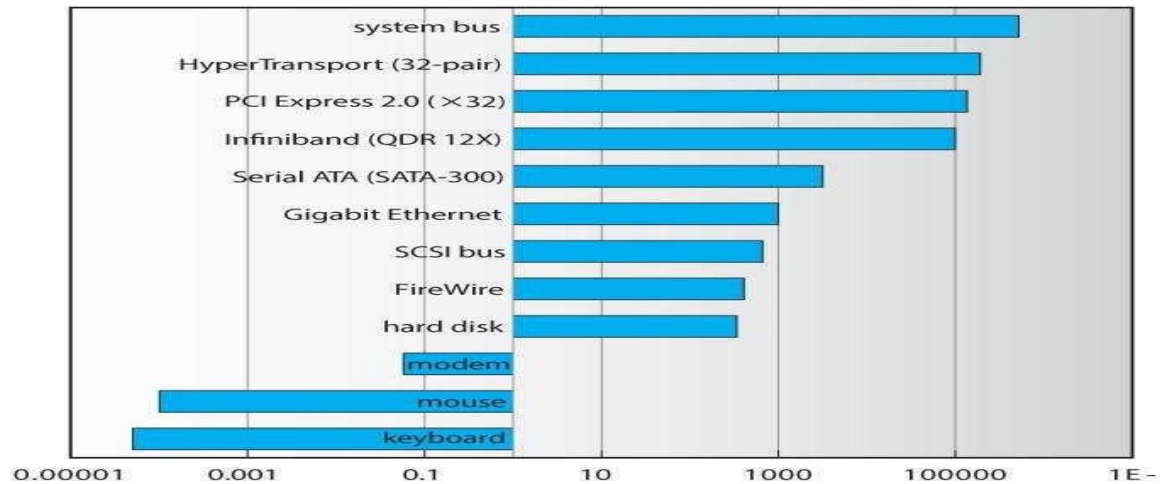


Figure - Sun Enterprise 6000 device-transfer rates (logarithmic).

3.3

Caching:

Caching involves keeping a copy of data in a faster-access location than where the data is normally stored. Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.

Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes.)

3.4 Spooling and Device Reservation:

A spool (Simultaneous Peripheral Operations On-Line) buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.

Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.

Spool queues can be general (any laser printer) or specific (printer number 42.)
 Oses can also provide support for processes to request / get exclusive access to a particular device, and/or to wait until a device becomes available

3.5 Error Handling: I/O requests can fail for many reasons, either transient (buffers overflow) or permanent(disk crash). I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable `errno` to one of a hundred or so well-defined values to indicate the specific error that has occurred. (See `errno.h` for a complete listing, or `man errno`.)
 Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

3.6 Kernel Data Structures:
 The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table.

These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure below.)
 Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.

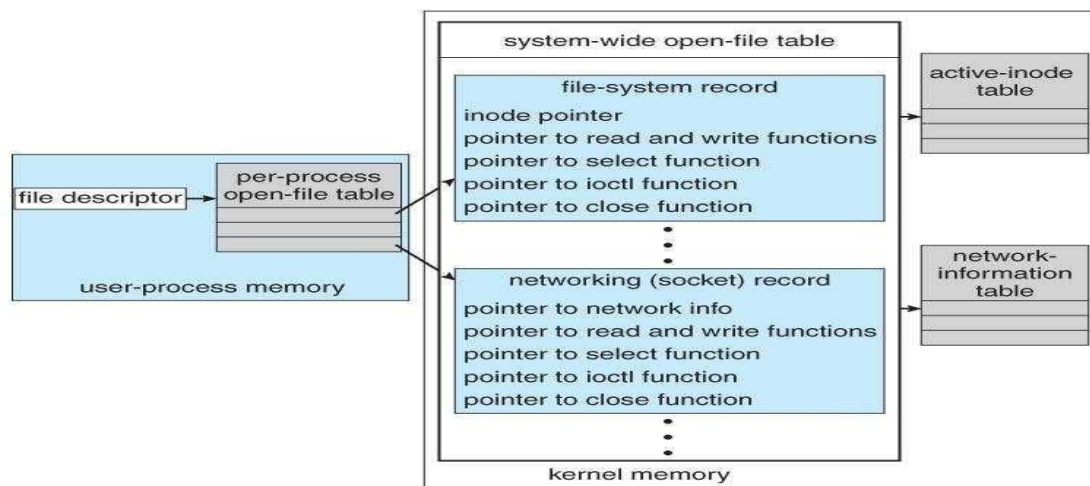


Figure - UNIX I/O kernel structure.

The I/O subsystem supervises:

- The management of the name space for files and devices.
- Access control to files and devices.

- Operation control
- File system space allocation
- Device allocation
- Buffering, catching, and spooling
- I/O

scheduling

4. Transforming I/O Requests to Hardware Operations:--

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.
- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
UNIX uses a **mount table** to map filename prefixes (e.g. /usr) to specific mounted devices.
- Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file).
- UNIX **uses special device files**, usually located in /dev, to represent and access physical devices directly.
Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
- The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler).
- The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver.

A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.

Below Figure showing the steps taken to process a (blocking) read request:

1. A process issues a blocking read() system call to a file descriptor of a file that has been **opened** previously.

2. The system call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process and the I/O request is completed.
3. Otherwise, a physical I/O needs to be performed, so the process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled.

The I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or via an in-kernel message.

4. The device driver allocates kernel buffer space to receive the data, and schedules the I/O.
The driver sends commands to the device controller by writing into the device control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request completed, determines the requests status, and signal the kernel I/O system that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process, and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocked the process.

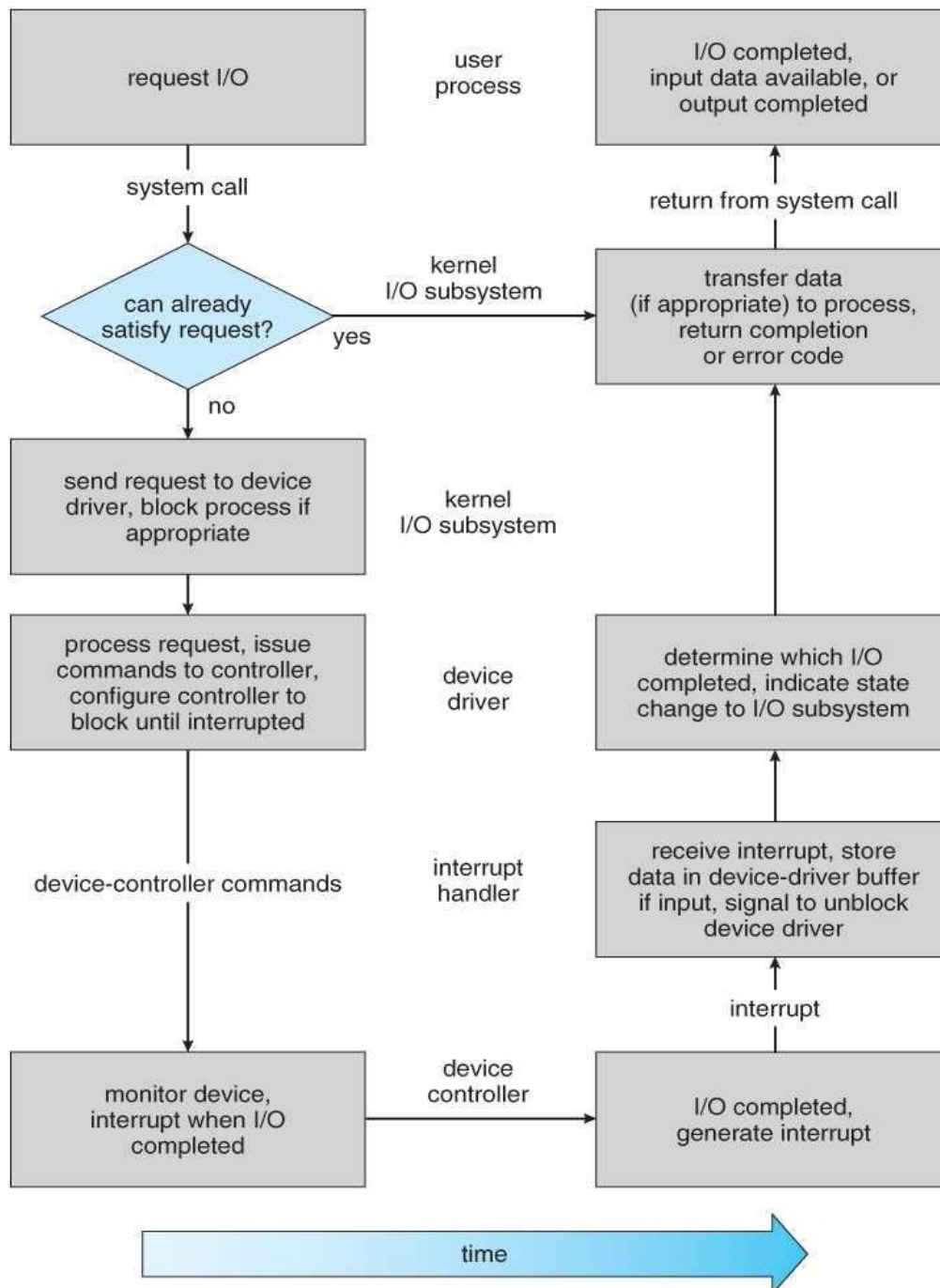


Figure - The life cycle of an I/O request.

5. STREAMS :--

- The streams mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.
- The user process interacts with the stream head. The device driver interacts with the device end.
- Zero or more stream modules can be pushed onto the stream, using `ioctl( )`. These modules may filter and/or modify the data as it passes through the stream. Each module has a read queue and a write queue.
- Flow control can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it. Without flow control, data is passed along as soon as it is ready.
- User processes communicate with the stream head using either `read( )` and `write( )` (or `putmsg( )` and `getmsg( )` for message passing). Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.

The device driver must respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, then data is typically dropped.

Streams are widely used in UNIX, and are the preferred approach for device drivers. For example, UNIX implements sockets using streams.

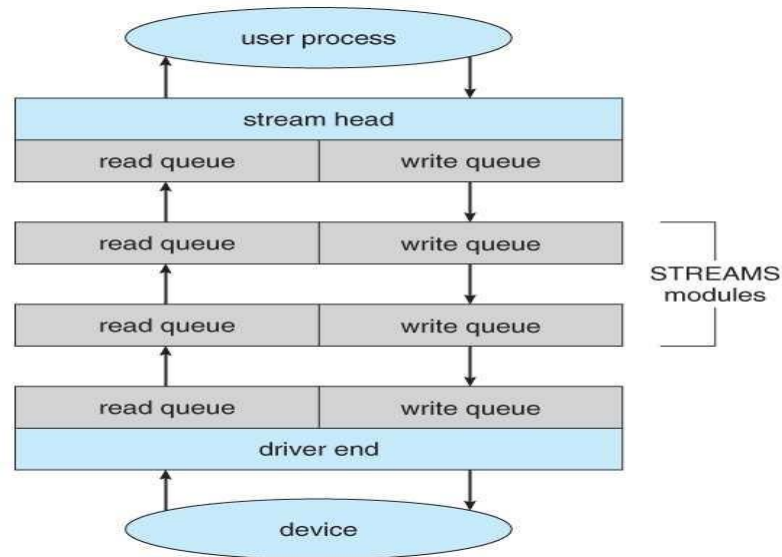


Figure - The STREAMS structure.

6. 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 .

(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.

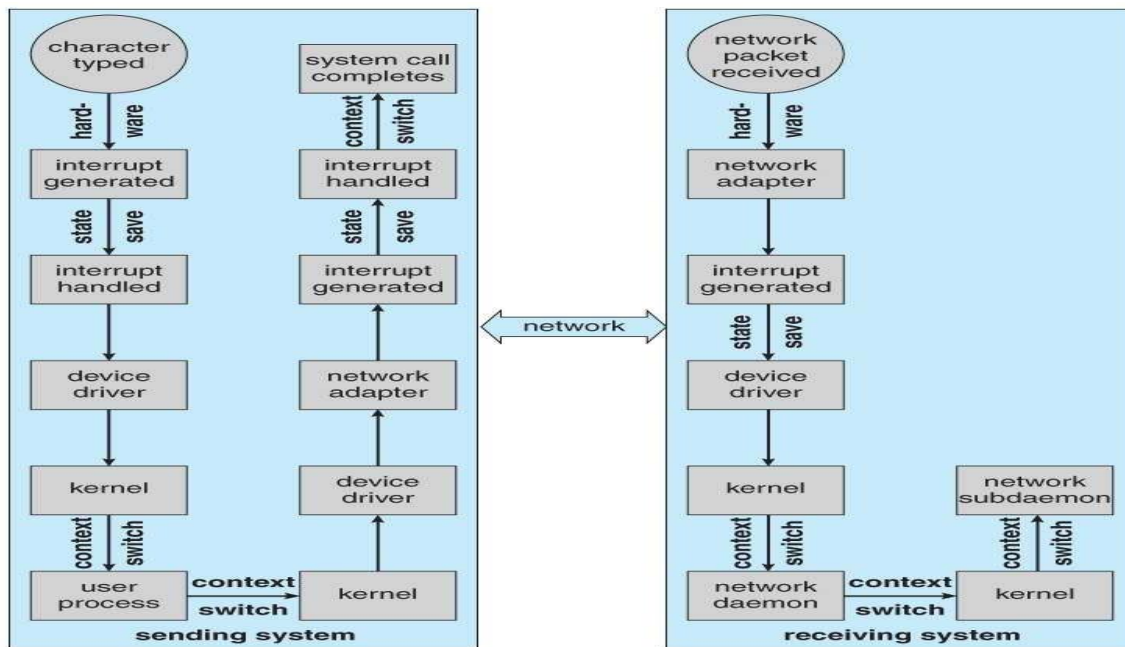


Figure 13.15 - Intercomputer communications.

Other systems use front-end processors to off-load some of the work of I/O processing from the CPU. For example a terminal concentrator can multiplex with hundreds of terminals on a single port on a large computer.

- ✚ Several principles can be employed to increase the overall efficiency of I/O processing:
 - Reduce the number of context switches.Reduce the number of times data must be copied.
- ✚ Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.
- ✚ Increase concurrency using DMA.
- ✚ Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
- ✚ 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. 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.

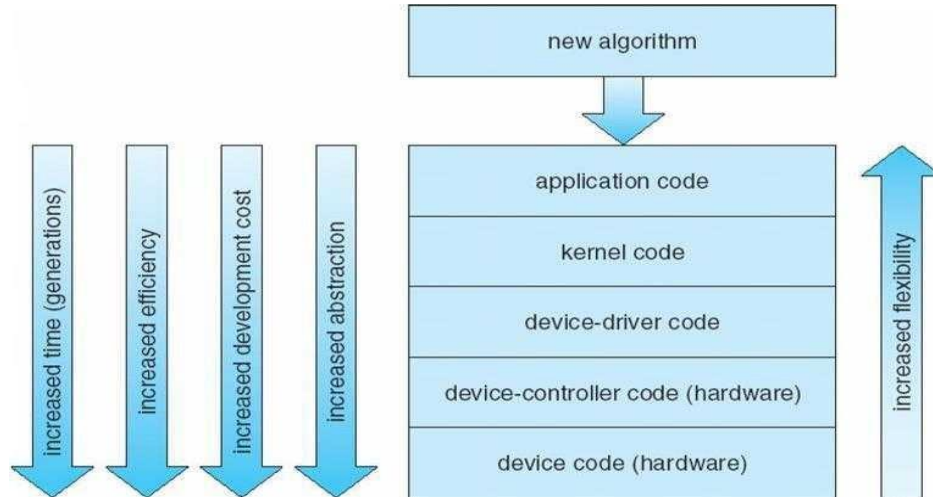


Figure: Device-functionality progression

Chapter -13

MASS STORAGE STRUCTURE

TOPICS:

- Disk Structure
- Disk Scheduling
- Disk Management
- Swap –space Management
- RAID Structure

Chapter -13

MASS STORAGE STRUCTURE

TOPICS:

- Disk Structure
- Disk Scheduling
- Disk Management
- Swap –space Management
- RAID Structure

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 is 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.)

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

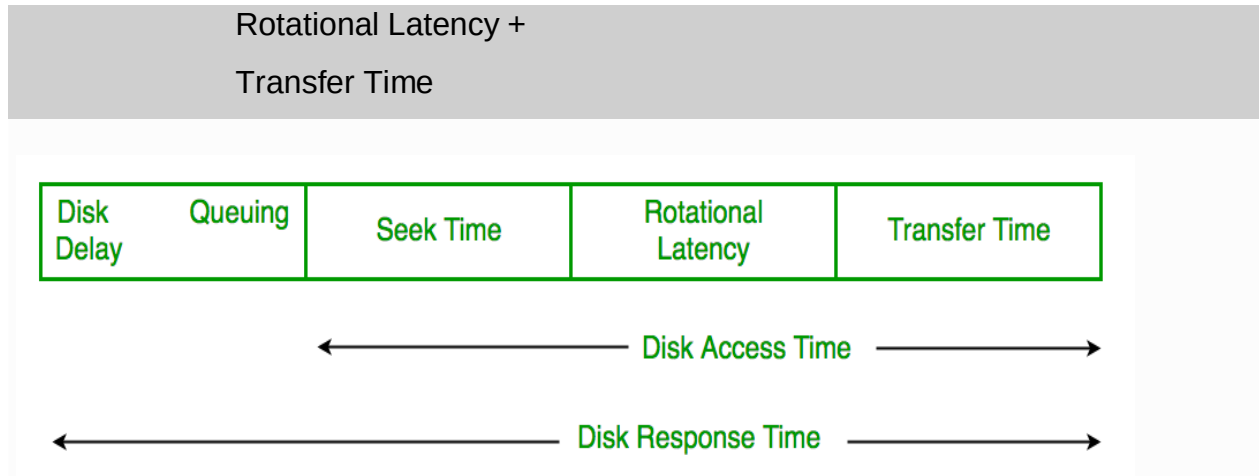
Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- **Seek Time**: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- **Rotational Latency**: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- **Transfer Time**: Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- **Disk Access Time**: Disk Access Time is:

$$\text{Disk Access Time} = \text{Seek Time} +$$



- **Disk Response Time:** Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

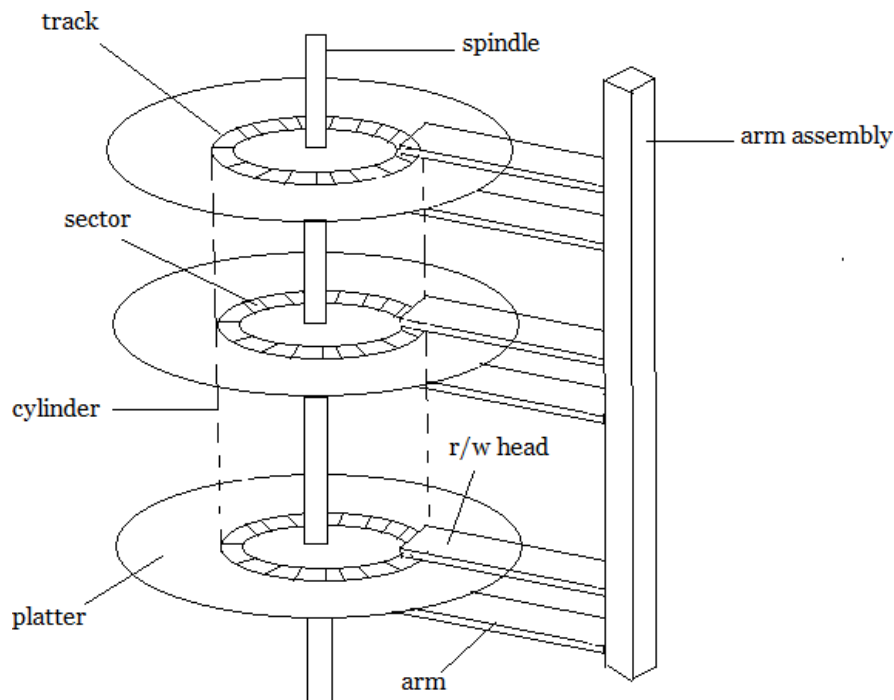
Secondary Storage and Disk Scheduling Algorithms

Secondary storage devices are those devices whose memory is non volatile, meaning, the stored data will be intact even if the system is turned off. Here are a few things worth noting about secondary storage.

- Secondary storage is also called auxiliary storage.
- Secondary storage is less expensive when compared to primary memory like RAMs.
- The speed of the secondary storage is also lesser than that of primary storage.
- Hence, the data which is less frequently accessed is kept in the secondary storage.
- A few examples are magnetic disks, magnetic tapes, removable thumb drives etc.

Magnetic Disk Structure

In modern computers, most of the secondary storage is in the form of magnetic disks. Hence, knowing the structure of a magnetic disk is necessary to understand how the data in the disk is accessed by the computer.



Structure of a magnetic disk

A magnetic disk contains several **platters**. Each platter is divided into circular shaped **tracks**. The length of the tracks near the centre is less than the length of the tracks farther from the centre. Each track is further divided into **sectors**, as shown in the figure.

Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk.

The speed of the disk is measured as two parts:

- **Transfer rate:** This is the rate at which the data moves from disk to the computer.
- **Random access time:** It is the sum of the seek time and rotational latency.

Seek time is the time taken by the arm to move to the required track. **Rotational latency** is defined as the time taken by the arm to reach the required sector in the track.

Even though the disk is arranged as sectors and tracks physically, the data is logically arranged and addressed as an array of blocks of fixed size. The size of a block can be **512** or **1024** bytes. Each logical block is mapped with a sector on the disk, sequentially. In this way, each sector in the disk will have a logical address.

Disk Scheduling Algorithms:

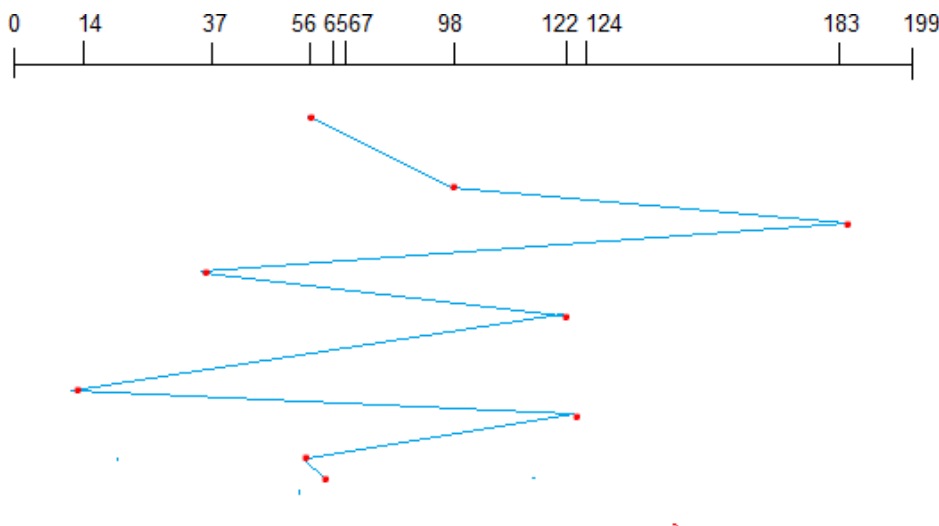
On a typical multiprogramming system, there will usually be multiple disk access requests at any point of time. So those requests must be scheduled to achieve good efficiency. Disk scheduling is similar to process scheduling. Some of the disk scheduling algorithms are described below.

First Come First Serve

This algorithm performs requests in the same order asked by the system. Let's take an example where the queue has the following requests with cylinder numbers as follows:

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The head moves in the given order in the queue i.e., **56→98→183→...→67**.



FCFS: FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

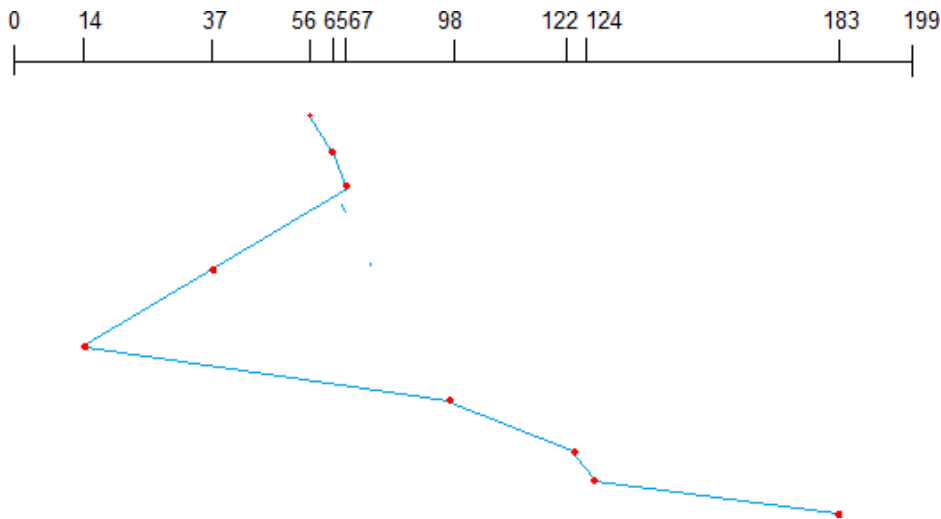
- Does not try to optimize seek time
- May not provide the best possible service

Shortest Seek Time First (SSTF)

Here the position which is closest to the current head position is chosen first. Consider the previous example where disk queue looks like,

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The next closest cylinder to **56** is **65**, and then the next nearest one is **67**, then **37**, **14**, so on.



In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

Advantages:

- Average Response Time decreases
- Throughput increases

Disadvantages:

- Overhead to calculate seek time in advance

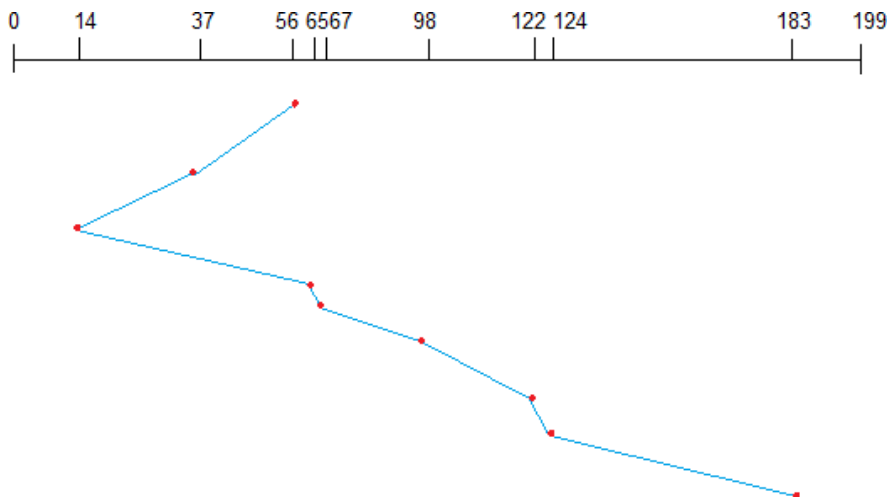
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favours only some requests

SCAN algorithm

- This algorithm is also called the elevator algorithm because of its behavior. Here, first the head moves in a direction (say backward) and covers all the requests in the path.
- Then it moves in the opposite direction and covers the remaining requests in the path. This behavior is similar to that of an elevator. Let's take the previous example,

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The head moves in backward direction and accesses **37** and **14**. Then it goes in the opposite direction and accesses the cylinders as they come in the path.



- In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path.
- So, this algorithm works as an elevator and hence also known as **elevator algorithm**. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

- Long waiting time for requests for locations just visited by disk arm.

CSCAN:

- In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.
- These situations are avoided in CSAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there.
- So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

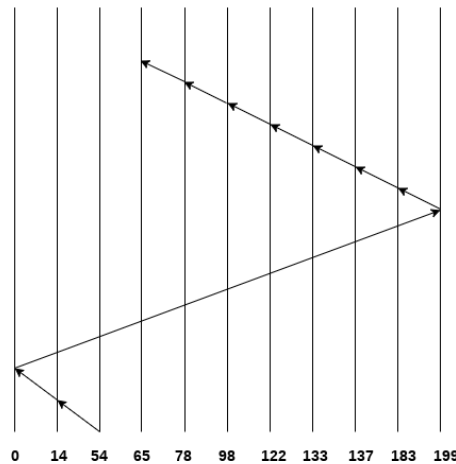
- In C-SCAN algorithm, the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C-SCAN scheduling.



No. of cylinders crossed = $40 + 14 + 199 + 16 + 46 + 4 + 11 + 24 + 20 + 13 = 387$

Advantages:

- Provides more uniform wait time compared to SCAN

LOOK: It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

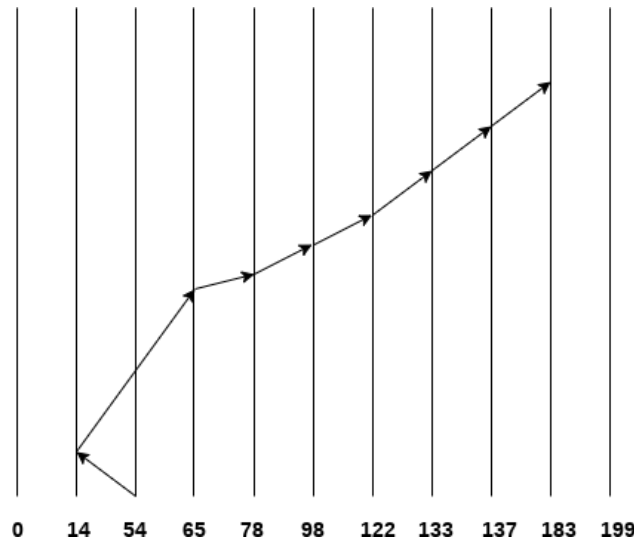
- It is like SCAN scheduling Algorithm to some extent except the difference that, in this scheduling algorithm, the arm of the disk stops moving inwards (or outwards) when no more request in that direction exists.
- This algorithm tries to overcome the overhead of SCAN algorithm which forces disk arm to move in one direction till the end regardless of knowing if any request exists in the direction or not.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using LOOK scheduling.



Number of cylinders crossed = $40 + 51 + 13 + 20 + 24 + 11 + 4 + 46 = 209$

C Look Scheduling

- C Look Algorithm is similar to C-SCAN algorithm to some extent. In this algorithm, the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.
- It is different from C SCAN algorithm in the sense that, C SCAN force the disk arm to move till the last cylinder regardless of knowing whether any request is to be serviced on that cylinder or not.

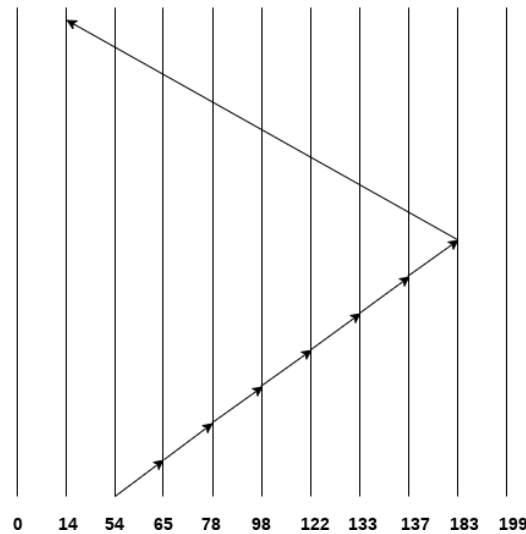
Each algorithm is unique in its own way. Overall Performance depends on the number and type of requests.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C LOOK scheduling.



Number of cylinders crossed = $11 + 13 + 20 + 24 + 11 + 4 + 46 + 169 = 298$

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.

Disk Management:

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.

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.

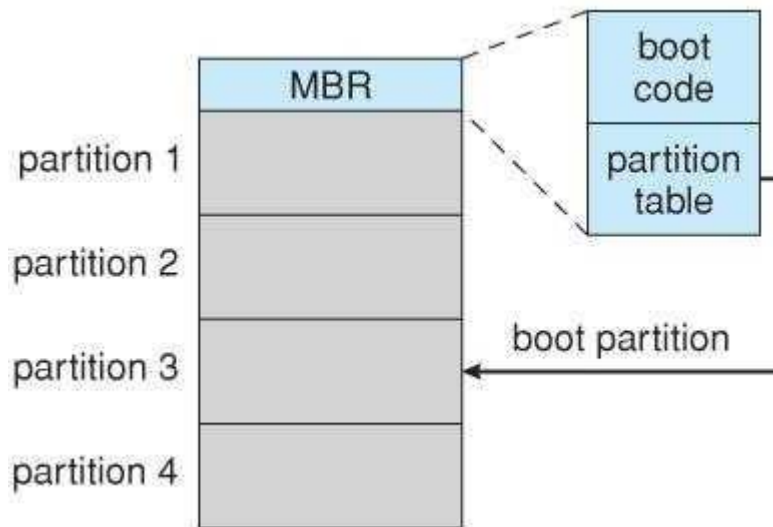


Figure - Booting from disk in Windows 2000.

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.
- **Sector slipping** may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.
- If the data on a bad block cannot be recovered, then a **hard error** has occurred., which requires replacing the file(s) from backups, or rebuilding them from scratch.

Swap-Space Management:

- Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.
- Managing swap space is obviously an important task for modern OS.

Swap-Space Use

- The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!
- Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

Swap-Space Location

Swap space can be physically located in one of two locations:

- As a large file which is part of the regular filesystem. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.
- As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

Swap-Space Management: An Example

- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back in from there than to write them out to swap space and then read them back.)
- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)

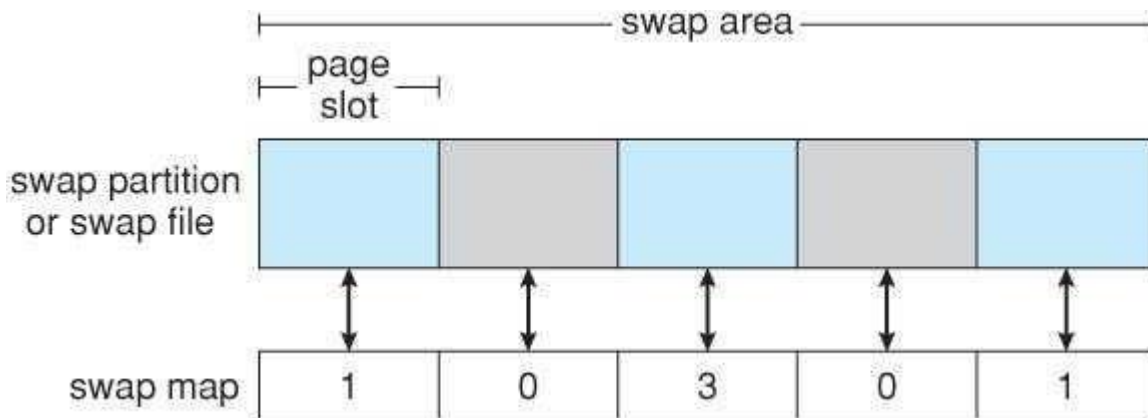


Figure 10.10 - The data structures for swapping on Linux systems.

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.

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.

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.
- 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.

RAID Levels

- Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability.
- Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost.
- These are described by different **RAID levels**, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)

Raid Level 0 - This level includes striping only, with no mirroring.

Raid Level 1 - This level includes mirroring only, no striping.

Raid Level 2 - This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data.

Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)

Raid Level 3 - This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data.

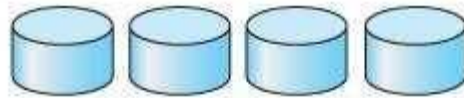
As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance.

Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

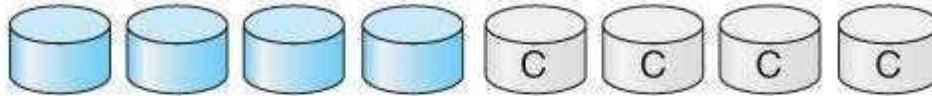
Raid Level 4 - This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.

Raid Level 5 - This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

Raid Level 6 - This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the [Reed-Solomon codes](#)), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



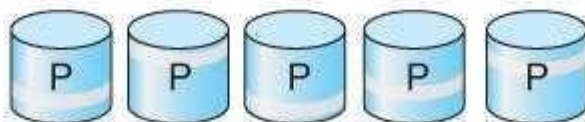
(c) RAID 2: memory-style error-correcting codes.



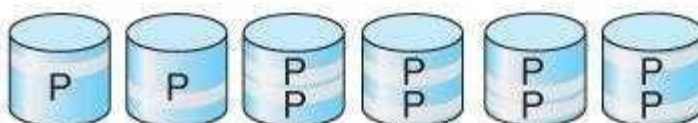
(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

Figure - 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.

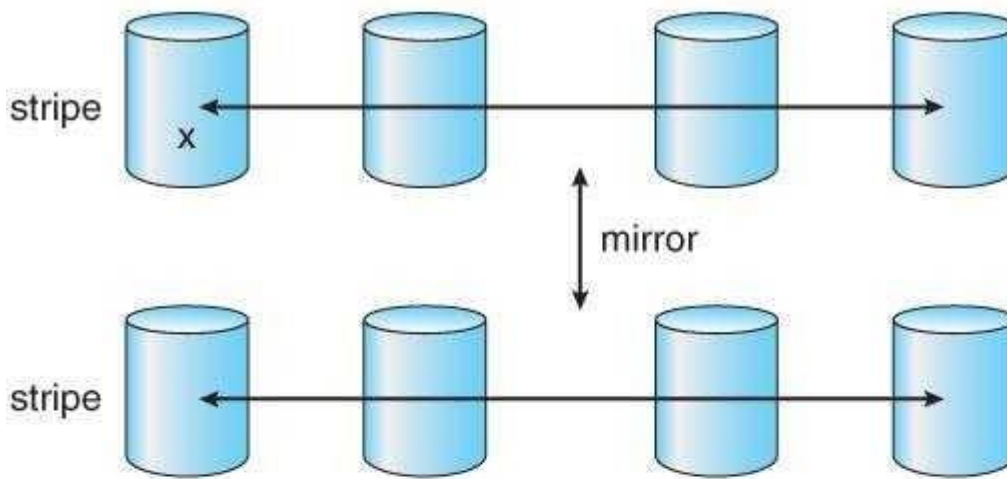
If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.

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.

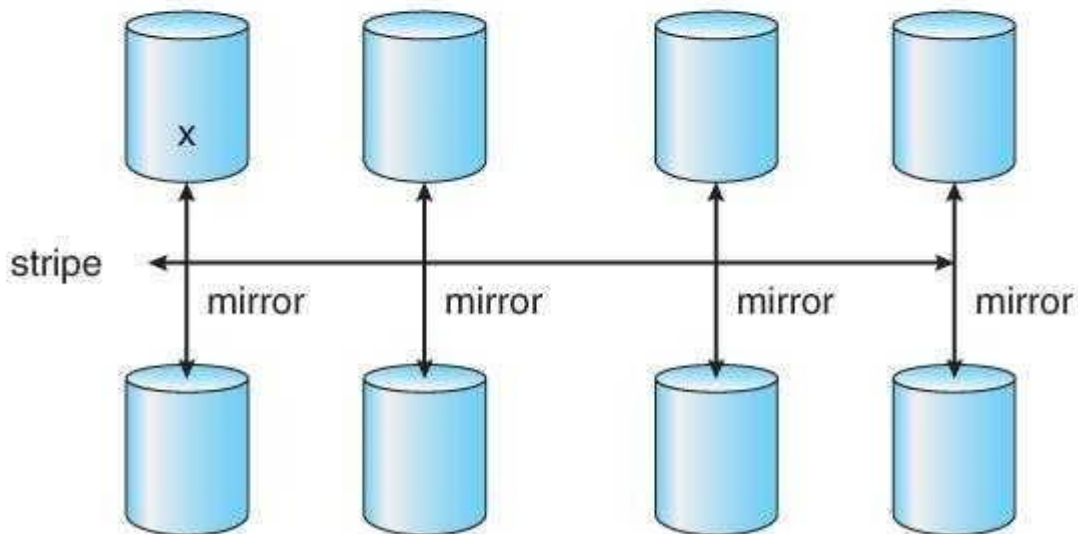
In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.

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.



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

Figure - RAID 0 + 1 and 1 + 0

Selecting a RAID Level

- Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.
- Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost,

but increases the likelihood that a second disk will fail before the first bad disk is repaired.

Extensions

- RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

Problems with RAID

- RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data.
- ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

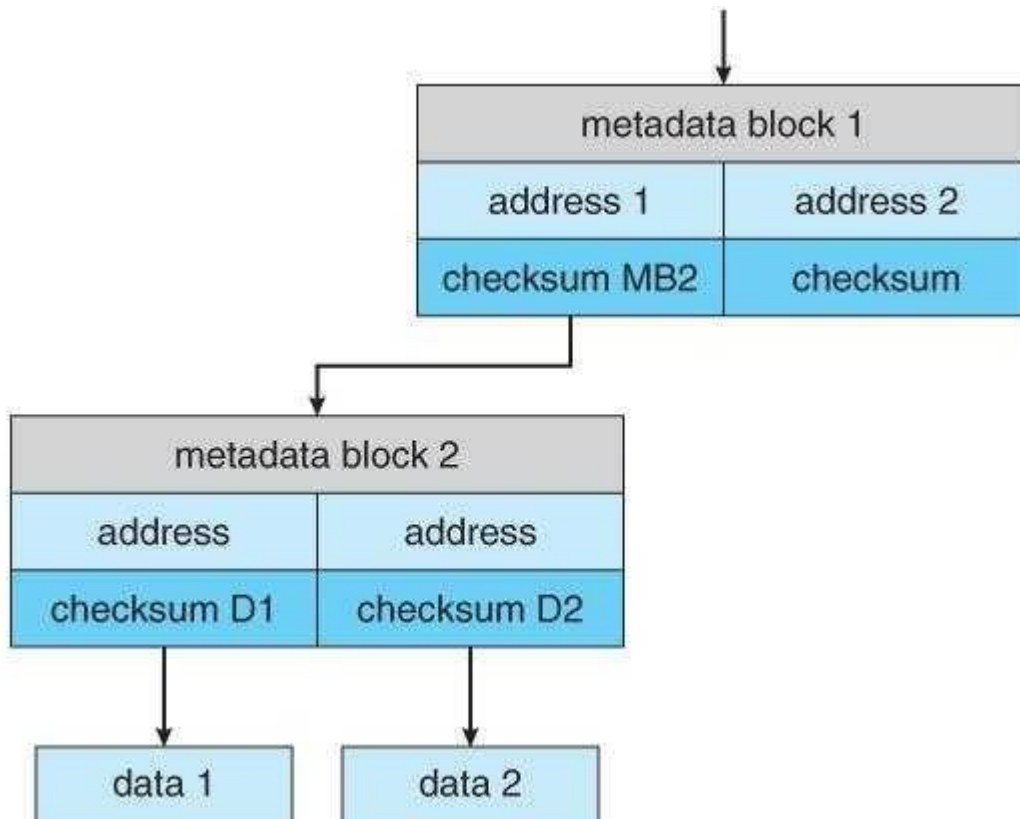
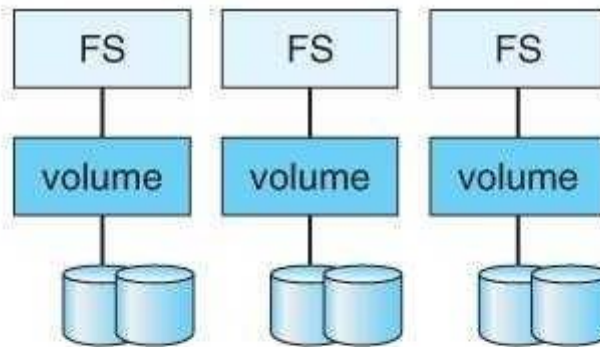


Figure 10.13 - ZFS checksums all metadata and data.

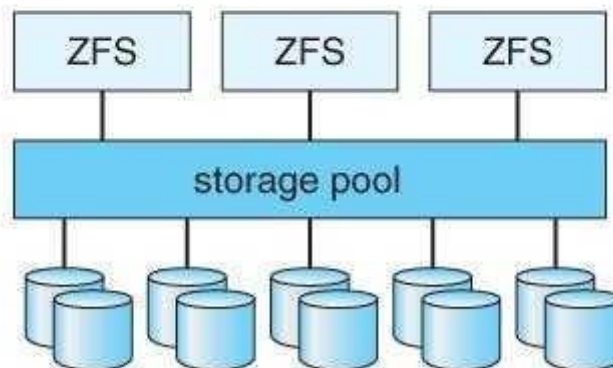
- Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even

harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.

- ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

Figure - (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.

Virtual Memory

- ❖ The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required.
- ❖ The VM manager maintains two separate views of a process's address space:
- ❖ A logical view describing instructions concerning the layout of the address space
- ❖ The address space consists of a set of non-overlapping regions, each representing a continuous, page-aligned subset of the address space
- ❖ A physical view of each address space which is stored in the hardware page tables for the process
- ❖ Virtual memory regions are characterized by:
- ❖ The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (**demand-zero memory**)
- ❖ The region's reaction to writes (page sharing or copy-on-write)
- ❖ The kernel creates a new virtual address space
- ❖ When a process runs a new program with the `exec()` system call
- ❖ Upon creation of a new process by the `fork()` system call
- ❖ On executing a new program, the process is given a new, completely empty virtual-address space; the program-loading routines populate the address space with virtual-memory regions
- ❖ Creating a new process with `fork()` involves creating a complete copy of the existing process's virtual address space
- ❖ The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child
- ❖ The parent's page tables are copied directly into the child's, with the reference count of each page covered being incremented
- ❖ After the fork, the parent and child share the same physical pages of memory in their address spaces

Swapping and Paging

The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else

The VM paging system can be divided into two sections:

- ❖ The **pageout-policy** algorithm decides which pages to write out to disk, and when
- ❖ The **paging mechanism** actually carries out the transfer, and pages data back into physical memory as needed
- ❖ Can page out to either swap device or normal files

- ❖ Bitmap used to track used blocks in swap space kept in physical memory
- ❖ Allocator uses next-fit algorithm to try to write contiguous runs

Kernel Virtual Memory:

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use.
- This kernel virtual-memory area contains two regions:
- A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code.
- The remainder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory.

Executing and Loading User Programs

- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made
- The registration of multiple loader routines allows Linux to support both the ELF and a.out binary formats
- Initially, binary-file pages are mapped into virtual memory
- Only when a program tries to access a given page will a page fault result in that page being loaded into physical memory
- An ELF-format binary file consists of a header followed by several page-aligned sections
- The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory

Static and Dynamic Linking:

- ❖ A program whose necessary library functions are embedded directly in the program's executable binary file is statically linked to its libraries
- ❖ The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- ❖ Dynamic linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once
- ❖ Linux implements dynamic linking in user mode through special linker library
- ❖ Every dynamically linked program contains small statically linked function called when process starts
- ❖ Maps the link library into memory
- ❖ Link library determines dynamic libraries required by process and names of variables and functions needed
- ❖ Maps libraries into middle of virtual memory and resolves references to symbols contained in the libraries
- ❖ Shared libraries compiled to be position-independent code (PIC) so can be loaded anywhere
- ❖ To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- ❖ Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- ❖ The Linux VFS is designed around object-oriented principles and is composed of four components:
 - ❖ A set of definitions that define what a file object is allowed to look like □
The inode object structure represent an individual file
 - ❖ The file object represents an open file
 - ❖ The superblock object represents an entire file system
 - ❖ A dentry object represents an individual directory entry
- ❖ To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
- ❖ Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS)
- ❖ The Linux VFS is designed around object-oriented principles and layer of software to manipulate those objects with a set of operations on the objects

- ❖ For example for the file object operations include (from struct file_operations in /usr/include/linux/fs.h)
 - int open(. . .) — Open a file
 - ssize_t read(. . .) — Read from a file
 - ssize_t write(. . .) — Write to a file
 - int mmap(. . .) — Memory-map a file

The Linux ext3 File System:

- ext3 is standard on disk file system for Linux
- Uses a mechanism similar to that of BSD Fast File System (FFS) for locating data blocks belonging to a specific file
- Supersedes older extfs, ext2 file systems
- Work underway on ext4 adding features like extents
- Of course, many other file system choices with Linux distros.
- The main differences between ext2fs and FFS concern their disk allocation policies
- In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file
- ext3 does not use fragments; it performs its allocations in smaller units
- The default block size on ext3 varies as a function of total size of file system with support for 1, 2, 4 and 8 KB blocks
- ext3 uses cluster allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation on a block group
- Maintains bit map of free blocks in a block group, searches for free byte to allocate at least 8 blocks at a time

Journaling: ext3 implements journaling, with file system updates first written to a log file in the form of transactions. Once in log file, considered committed. Over time, log file transactions replayed over file system to put changes in place. On system crash, some transactions might be in journal but not yet placed into file system. Must be completed once system recovers. No other consistency checking is needed after a crash (much faster than older methods). Improves write performance on hard disks by turning random I/O into sequential I/O.

The Linux Proc File System:

The proc file system does not store data, rather, its contents are computed on demand according to user file I/O requests

proc must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains

It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode

When data is read from one of these files, proc collects the appropriate information, formats it into text form and places it into the requesting process's read buffer

Input and Output:

The Linux device-oriented file system accesses disk storage through two caches: Data is cached in the page cache, which is unified with the virtual memory system

- Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block
- Linux splits all devices into three classes:
- block devices allow random access to completely independent, fixed size blocks of data
- character devices include most other devices; they don't need to support the functionality of regular files
- network devices are interfaced via the kernel's networking subsystem

Block Devices: Provide the main interface to all disk devices in a system

- The block buffer cache serves two main purposes
- it acts as a pool of buffers for active I/O. it serves as a cache for completed I/O.
- The request manager manages the reading and writing of buffer contents to and from a block device driver .
- Kernel 2.6 introduced Completely Fair Queueing (CFQ)
- Now the default scheduler. Fundamentally different from elevator algorithms .Maintains set of lists, one for each process by default.
- Uses C-SCAN algorithm, with round robin between all outstanding I/O from all processes . Four blocks from each process put on at once

Device-Driver Block Structure:

Character Devices: A device driver which does not offer random access to fixed blocks of data .

A character device driver must register a set of functions which implement the driver's various file I/O operations . The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device .The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface.

Line discipline is an interpreter for the information from the terminal device . The most common line discipline is tty discipline, which glues the terminal's data stream onto standard input and output streams of user's running processes, allowing processes to communicate directly with the user's terminal .

Several processes may be running simultaneously, tty line discipline responsible for attaching and detaching terminal's input and output from various processes connected to it as processes are suspended or awakened by user. Other line disciplines also are implemented have nothing to do with I/O to user process – i.e. PPP and SLIP networking protocols.

Interprocess Communication: Like UNIX, Linux informs processes that an event has occurred via signals .There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process .The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and wait_queue structures .Also implements System V Unix semaphores . Process can wait for a signal or a semaphore .Semaphores scale better.Operations on multiple semaphores can be atomic.

Passing Data Between Processes:

The pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read a the other .

Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space .

To obtain synchronization, however, shared memory must be used in conjunction with another Inter process communication mechanism.

Network Structure:

Networking is a key area of functionality for Linux .It supports the standard Internet protocols for UNIX to UNIX communications .It also implements protocols native to non-UNIX operating systems, in particular, protocols used on PC networks, such as

Appletalk and IPX . Internally, networking in the Linux kernel is implemented by three layers of software:

The socket interface Protocol drivers Network device drivers Most important set of protocols in the Linux networking system is the internet protocol suite .

It implements routing between different hosts anywhere on the network . On top of the routing protocol are built the UDP, TCP and ICMP protocols Packets also pass to firewall management for filtering based on firewall chains of rules.

Chapter – 14

Security

Topics:

- **User Authentication**
- **Program threats**

- Authentication refers to identifying each user of the system and associating the execution of those users. It is the responsibility of the Operating System to create a protection system v a user who is running a particular program is authentic.

Operating Systems generally identifies/authenticates users using following **three ways**:

- User possession (a key or card)
- User knowledge (a user identifier and password),
- User attribute (fingerprint, retina pattern, or signature)
- **Username / Password** – User need to enter a registered username and password with Op login into the system.
- The most common approach to authenticating a user identity is the use of passwords.

- **Biometrics**-There are many other variations on the use passwords for authentication. Palm are common to secure physical access, for example access to a data center. These read parameters against what is being read from their hand –reader pad.

One Time passwords

One-time passwords provide additional security along with normal authentication. In One-Time Password, a unique password is required every time user tries to login into the system. Once a one-time password is used, it cannot be used again. One-time passwords are implemented in various ways.

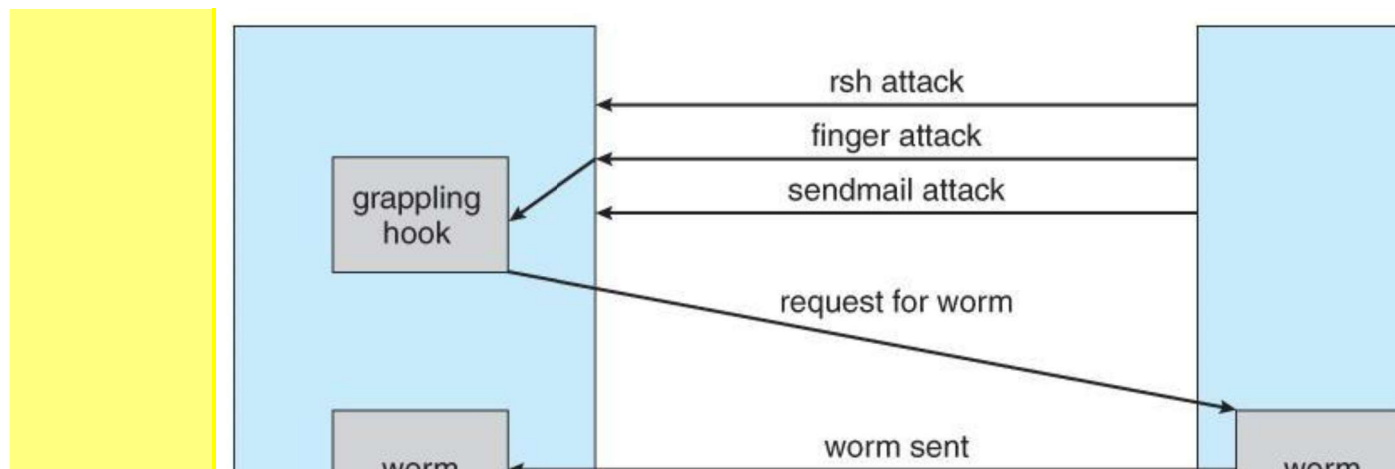
- **Random numbers** – Users are provided cards having numbers printed along with corresponding alphabets. The system asks for numbers corresponding to few alphabets randomly chosen.

2. Overflow the current return address on the stack with the address of the exploit code.
3. Write a simple set of code for the next space in the stack that includes the command the user wishes to execute, for instance, spawn a shell.

(14.3) System Threats:

- System threats refers to misuse of system services and network connections to put user in danger.
- System threats can be used to launch program threats on a complete network called as program threats.
- System threats creates such an environment that operating system resources/ user files are in danger.
- Following is the list of some well-known system threats.

OPERATING SYSTEM (I



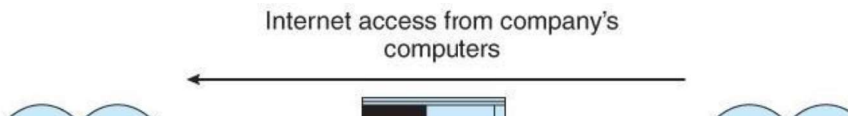
(14.4) Securing Systems and Facilities:

Securing system and facilities is intimately linked to intrusion detection. Both techniques need to assure that a system is secure and that, if a security breach happens, it is detected.

One method of improving system security is periodically to scan the system for security holes. This is best done when the computer has relatively little use, so they will have less effect than logging. Such a scan covers a variety of aspects of the system:

OPERATING SYSTEM (U

- A common architecture is to establish a de-militarized zone, DMZ, which sort of sits "between" the company's internal domain and the outside world, as shown below.
- Company computers can reach either the DMZ or the outside world, but outside computers can only reach the DMZ.
- Perhaps most importantly, the DMZ cannot reach any of the other company computers, so if the DMZ is breached, the attacker cannot get to the rest of the company network.



In addition to the common firewalls protecting a company internal network from the outside world, some specialized forms of firewalls that have been recently developed:

- A **personal firewall** is a software layer that protects an individual computer. It may be a part of the operating system or a separate software package.
- An **application proxy firewall** understands the protocols of a particular service and acts as a relay) for the particular service.
- For example, an SMTP proxy firewall would accept SMTP requests from the outside world for security concerns, and forward only the "safe" ones on to the real SMTP server behind the firewall.
- **XML firewalls** examine XML packets only, and reject ill-formed packets. Similar firewalls exist for other specific protocols.
- **System call firewalls** guard the boundary between user mode and system mode, and reject

History:

Linux is a modern, free operating system based on UNIX standards

- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the m UNIX compatibility
- Its history has been one of collaboration by many users from all around the world, cor exclusively over the Internet
- It has been designed to run efficiently and reliably on common PC hardware, but also ru other platforms

- extra hardware support

Version 1.2 (March 1995) was the final PC-only Linux kernel.

Version 2.0 was released in June 1996

- support for multiple architectures, including a fully 64-bit native Alpha port
- support for multiprocessor architectures
- improved memory-management code
- improved TCP/IP performance
- support for internal kernel threads
- standardized configuration interface

2.4 and 2.6 increased SMP support

OPERATING SYSTEM (OS)

- Ubuntu, Fedora, Debian, Open Suse,

Linux Licensing •

- Linux kernel is distributed under GNU General Public License (GPL)
- GPL is defined by the Free Software Foundation
- GPL implications: •
- anyone using Linux, or creating their own derivative of Linux, may not make the derivative proprietary •
- software released under GPL may not be redistributed as binary-only • LGPL: Lesser GPL •
- allow non-(L)GPL software to link to LGPL licensed software

Components of Linux System

Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** – Portability means software can works on different types of hardware in same and application programs supports their installation on any kind of hardware platform.
- **Open Source** – Linux source code is freely available and it is community based dev Multiple teams work in collaboration to enhance the capability of Linux operating continuously evolving.
- **Multi-User** – Linux is a multiuser system means multiple users can access system resou ram/ application programs at same time.

- **Utilities** – Utility programs that provide the user most of the functionalities of an operating system.

Linux History

- **Evolution of Computer:** In earlier days, computers were as big as houses or parks. So you had a lot of difficulties in operating them.
- Moreover, every computer has a different operating system which made it completely different from others.
- Every software was designed for a specific purpose and was unable to operate on other computers.
- It was extremely costly and normal people neither can afford it nor can understand it.
- **Evolution of Unix:** In 1969, a team of developers of Bell Labs started a project to make a

- In 1991, Linus Torvalds a student at the university of Helsinki, Finland, thought to have an academic version of Unix started writing its own code. Later this project became the Linux kernel.
- He wrote this program specially for his own PC as he wanted to use Unix 386 Intel compiler but he could not afford it.
- He did it on MINIX using GNU C compiler. GNU C compiler is still the main choice to compile Linux. Other compilers are also used like Intel C compiler.
- He started it just for fun but ended up with such a large project. Firstly he wanted to name it 'Linux' but later it became 'Linux'.
- He published the Linux kernel under his own license and was restricted to use as commercial. Most of its tools from GNU software and are under GNU copyright. In 1992, he released the GNU General Public License.
- **Linux Today**

OPERATING SYSTEM (U

- Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way.
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- 1) Components of a Linux System
- The Linux system has three main bodies of code, in sequence with, most of the time, the same implementations.
- **THE KERNEL:** “The kernel” is in charge for maintaining all the vital abstractions of the system, together with such things as virtual memory and processes. The Linux kernel forms the core of the operating system. It provides all the functionality compulsory to run processes, and it also

OPERATING SYSTEM (O

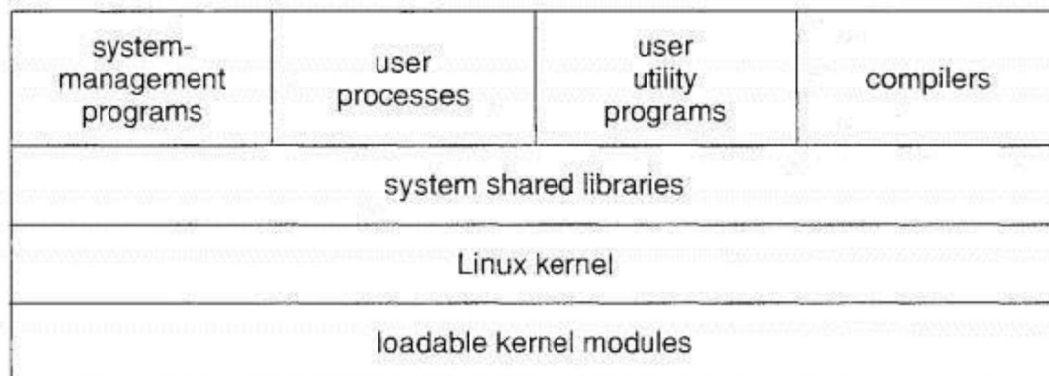


Fig: Components of the Linux system

Kernel module:

A **kernel module** is a binary image containing code and data structures that runs in the UNIX following characteristics:

- Is statically loaded as part of /vmunix or dynamically loaded into memory
- Runs in kernel mode
- Has a file name ending with the extension .mod
- Contains a well-defined routine that executes first to initialize the module
- May be a device driver when it performs any one of these additional tasks:
 - Handles interrupts from hardware devices
 - Accepts I/O requests from applications

- Tools to support kernel modules: lsmod, rmmod, modprobe

Module Management:

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Module loading is split into two separate sections:
- Managing sections of module code in kernel memory
- Handling symbols that modules are allowed to reference
- The module requestor manages loading requested, but currently unloaded, modules; it also allows the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

Conflict Resolution :

- A mechanism that allows different device drivers to reserve hardware resources and to protect resources from accidental use by another driver.
- The conflict resolution module aims to:
 - Prevent modules from clashing over access to hardware resources
 - Prevent auto probes from interfering with existing device drivers
 - Resolve conflicts with multiple drivers trying to access the same hardware:
 - 1. Kernel maintains list of allocated HW resources
 - 2. Driver reserves resources with kernel database first
 - 3. Reservation request rejected if resource not available

Credentials - Each process must have an associated user ID and one or more group IDs to determine the process's rights to access system resources and files

Personality - Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls.

Used primarily by emulation libraries to request that system calls be compatible with certain specific architectures.

Namespace — Specific view of file system hierarchy.

- Most processes share common namespace and operate on a shared file-system hierarchy
- But each can have unique file-system hierarchy with its own root directory and set of mount points

- The **file table** is an array of pointers to kernel file structures.
- When making file I/O system calls, processes s refer to files by their index into **descriptor (fd)**
- Whereas the file table lists the existing open files, the file-system context applies to req files.
- The current root and default directories to be used for new file searches are stored here
- The **signal-handler table** defines the routine in the process' s address space to be ca signals arrive.
- The **virtual-memory context** of a process describes the full contents of the its private a

Scheduling

- The job of allocating CPU time to different tasks within an operating system.
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver
- As of 2.5, new scheduling algorithm – preemptive, priority-based, known as
 - Real-time range

- ❖ If 10 runnable tasks, each runs for 1ms
- ❖ **Minimum granularity** ensures each run has reasonable amount of time (which actually violates fairness idea)

Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
 - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs
 - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt

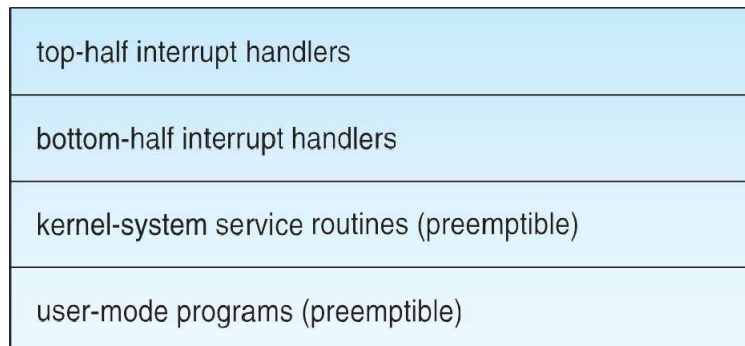
OPERATING SYSTEM (U

single processor	multiple processors
Disable kernel preem	Acquire spin lock.
Enable kernel preem	spin lock.

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration
- Interrupt service routines are separated into a *top half* and a *bottom half*

OPERATING SYSTEM (I

- each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level



Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into four different **zones** due to hardware characteristics

Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
- Each allocatable memory region is paired with an adjacent partner
- Whenever two allocated partner regions are both freed up they are combined to form a larger region
- If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request

Splitting of Memory in a Buddy Heap

