Chapter 1: Operating System Overview (3hrs)

Operating System:

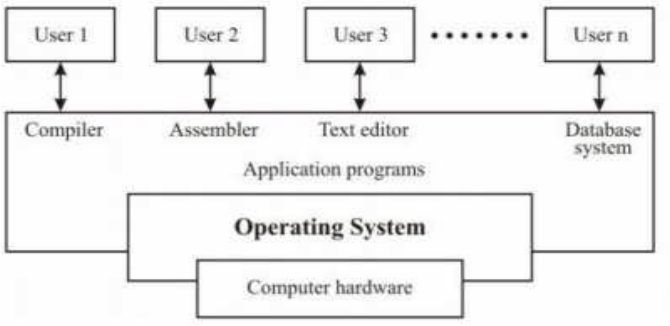
An operating system is a program that controls the execution of application program an act as an interface between the user of a computer and the computer hardware. An operating system is a lower level of software that user programs run on. Os is built directly on the hardware interface and provides an interface between the hardware and the user program. It shares the characteristics with software and hardware. Os keeps tracks of the status of each resource and decides who gets a resource for how long and when. An operating system performs basic tasks such as, controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems.

An Operating System (OS) is a collection of programs that acts as an interface between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user may execute the programs. Operating Systems are viewed as resource managers.

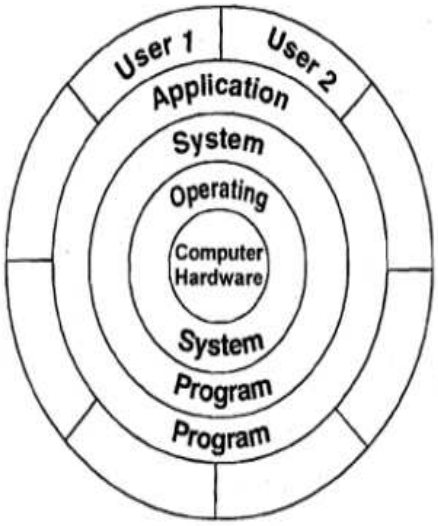
OS is designed to for two basic purposes:

1. It controls the allocation and the use of computing system's resource among the various users and tasks.
2. It provides an interface between the computer hardware and programmers that simplifies and makes feasible for coding, creating, debugging of application program.

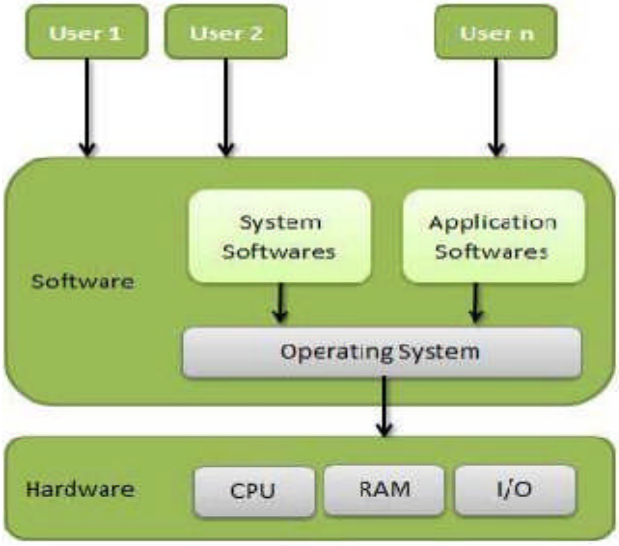
The abstract view of the components of a computer system and the positioning of OS is shown in the figure.



Abstract View of Operating System



Positioning of Operating System in Computer



Conceptual View of Operating System

1.1 Operating System Objectives and Function

An OS is a program that controls the execution of application program and acts as an interface between application and the computer hardware. It can be thought of as having three objectives:

1. Convenience: An OS makes a computer more convenient to use.
2. Efficiency: An Os allows the computer system resources to be used in an efficient manner.
3. Ability to evolve: An OS should be constructed in such a way as to permit the effective development, testing and introduction of new system function without interfering with service.

Operating Systems functions:

An operating system provides services to programs and to the users of those programs. It provided by one environment for the execution of programs. The services provided by one operating system is difficult than other operating system. Operating system makes the programming task easier.

The main functions of an operating system are as follows:

- Process Management
- Memory Management
- Secondary Storage Management

- I/O Management
- File Management
- Protection
- Networking Management
- Command Interpretation.

Process Management:

The CPU executes a large number of programs. A process is a program in execution. The operating system is responsible for the following activities in connection with processes management:

- The creation and deletion of both user and system processes,
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization.
- The provision of mechanisms for deadlock handling.

Memory Management:

Memory is the most expensive part in the 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. 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.
- Allocate and de allocates memory space as needed.

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 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 modem computer systems use hard disk as the primary 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. Hence the proper management of disk storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with disk management:

- Free space management.
- Storage allocation.
- Disk Scheduling

I/O Management:

One of the purposes of an operating system is enables the user to use various kinds of I/O devices on computer and hide the complexities of using hardware devices from the user. The operating system is responsible for the following activities in connection to I/O management:

- A buffer caching system.
- To activate a general device driver code.
- To run the driver software for specific hardware devices as and when required.

File Management:

File management is one of the most visible services of an operating system. 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, sound or video. The operating system is responsible for the following activities in connection to the file management:

- The creation and deletion of files.
- The creation and deletion of directory.
- The support for manipulating files and directories.
- The mapping of files onto disk storage.
- Backup of files on stable (nonvolatile) storage.ϖ Protection and security of the files.

Protection:

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 and users that have gained proper authorization from the operating system. The operating system is responsible for the following activities in connection to the protection:

- Provides a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls.
- Improve reliability by detecting latent errors at the interfaces between component subsystems.
- Early detection of interface errors to prevent healthy subsystem by a subsystem that is malfunctioning.
- Provide a mean of authentication and authorization

Networking:

With the advancement in technology need of distributed system and sharing or accessing information from remote computer has becomes the order of day. Almost every organization is dependent of Intranet or Internet. So there is a need of networking protocols and there implementation. The operating system is responsible for the following activities in connection to the networking:

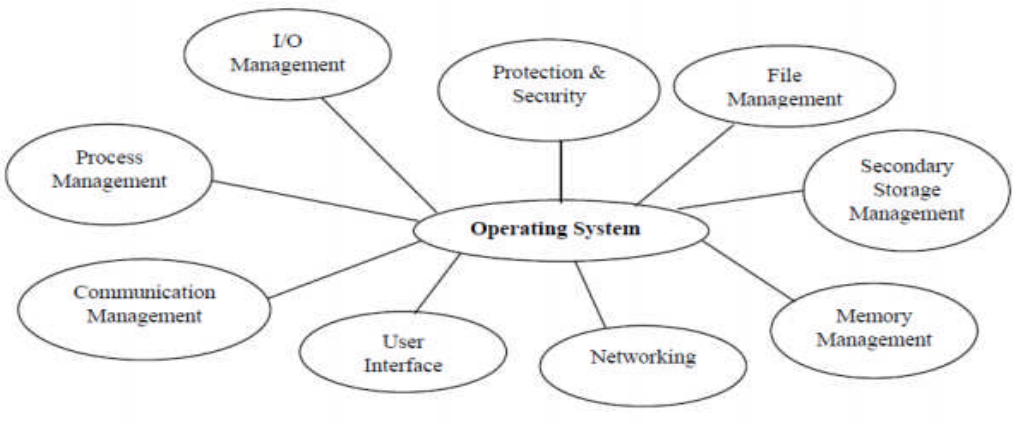
- Setup of a networking connection (Wireless & LAN)
- Mechanism for maintaining IP Address (Manual & by using DHCP).
- Managing networking protocol.
- Security of network connection.
- Firewall management for blocking unauthorized access of computer.

Command Interpretation

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system. The command statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking. The operating system is responsible for the following activities in connection to the Command Interpretation:

- Provides a mechanism to read command from user or process.
- Interpret command into low level language.

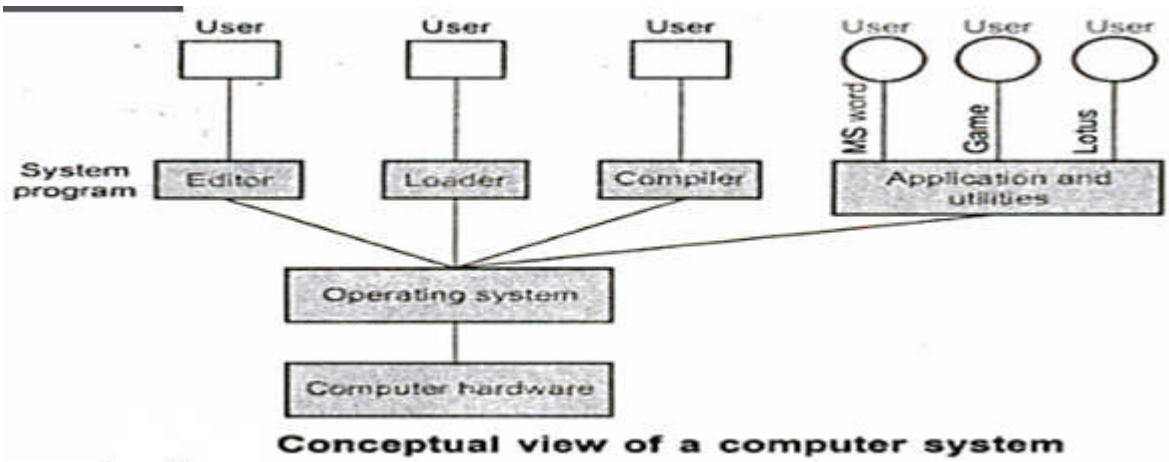
- Interact with required hardware, resource or file.
- Generate output defined in the class.



Functions performed by Operating System

1.1.1 OS as a user/computer interface

Every general purpose computer consists of the hardware, operating system, system program, application program. The hardware consists of memory, CPU, ALU, I/O device, peripheral device and storage device. System program consists of compilers, loaders, editors, OS etc. The application program consists of business program, database program. Every computer must have an operating system control and co-ordinates the use of the hardware among the various system program and application program for a various users. It simply provides an environment within which other program can do useful work. Figure below shows the conceptual view of operating system.



Conceptual view of a computer system

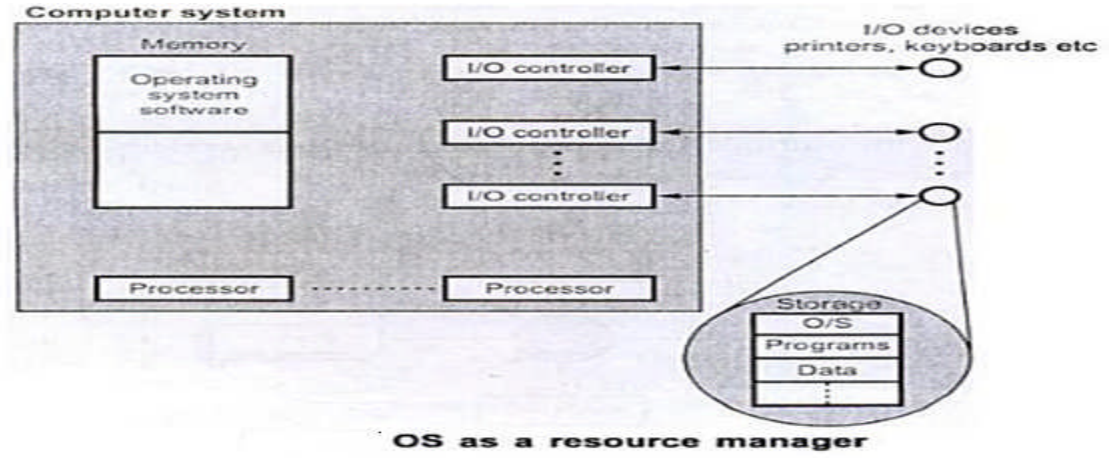
The operating system is a set of special program that run on a computer system that allow it to work properly. If performs basic task such as recognizing input from the keyboard, keeping track of files and directories on the disk, sending output to the display screen and controlling a peripheral devices. OS is designed to for two basic purposes:

3. It controls the allocation and the use of computing system's resource among the various users and tasks.
4. It provides an interface between the computer hardware and programmers that simplifies and makes feasible for coding, creating, debugging of application program.

The operating system must follow the task:

1. Provide the facilities to create, modification of program and data files using an editor.
2. Access to the compiler for translating the user program from high level language to machine language.
3. Provide a loader program to move the compiled program code to the computer's memory for execution.
4. Provide routines that handle the details of I/O programming.

1.1.2 OS as a Resource Manager



OS as a resource manager

The main resources are managed by operating system. A portion of the operating system is in main memory which includes kernel, which contains the most frequently used function in the operating system and at a given time other portion of the OS currently in use. The reminder of the main memory contains other user program and data. The allocation of main memory is

controlled jointly by the OS and memory management hardware in the processor. The operating system decides when an I/O device can be used by a program in execution and control access to and use of files. The processor itself is a resource and the operating system must determine how much processor time is to be devoted to the execution of a particular user program.

Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the alternative view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and input/output devices among the various programs competing for them.

When a computer (or network) has multiple users, the need for managing and protecting the memory, input/output devices, and other resources is even greater, since the users might otherwise interface with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of which programs are using which resources, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes **multiplexing** (sharing) resources in two different ways:

1. Time Multiplexing
2. Space Multiplexing

1. Time Multiplexing

When the resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on.For example: With only one CPU and multiple programs that want to run on it, operating system first allocates the CPU to one long enough, another one gets to use the CPU, then another and ten eventually the first one again. Determining how the resource is time multiplexed – who goes next and for how long – is the task of the operating system.

2. Space Multiplexing

In space multiplexing, instead of the customers taking turns, each one gets part of the resource. For example: Main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more

efficient to hold several programs in memory at once rather than give one of them all of it, especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them.

1.2 Evolution of Operating System

1.2.1 **Serial processing**

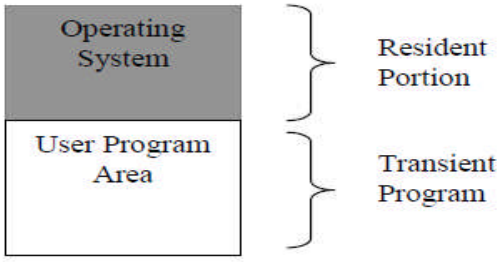
The earliest computer system has no OS at all, and is characterized as serial processing because users have to reserve time slots in advance, and during the allotted period, they occupy the computer exclusively. Thus the computer will be used in sequence by different users. These early systems presented two major problems:

1. Users may finish their tasks earlier than you have expected, and unfortunately the rest time is simply wasted. Or they may run into problems, cannot finish in the allotted time, and thus are forced to stop, which causes much inconvenience and delays the development.

2. In such systems, programs are presented by cards. Each card has several locations on it, where there may be a hole or not, respectively indicating 0 or 1. Programs are loaded into memory via a card reader. With no OS available, to compile their programs, users have to manually load the compiler program first with the user program as input. This involves mounting, or dismounting tapes or setting up card decks. If an error occurred, the user has to repeat the whole process from the very beginning. Thus much time is wasted.

1.2.2 Simple Batch Systems

Some computer systems only did one thing at a time. They had a list of the computer system may be dedicated to a single program until its completion, or they may be dynamically reassigned among a collection of active programs in different stages of execution. Batch operating system is one where programs and data are collected together in a batch before processing starts. A job is predefined sequence of commands, programs and data that are combined in to a single unit called job. Figure shows the memory layout for a simple batch system. Memory management in batch system is very simple.
Memory is usually divided into two areas: Operating system and user program area.



Scheduling is also simple in batch system. Jobs are processed in the order of submission i.e. first come first served fashion. Batch system often provides simple forms of file management. Access to file is serial. Batch systems do not require any time critical device management. Batch systems are inconvenient for users because users can not interact with their jobs to fix problems. There may also be long turnaround times. Example is generating monthly bank statement.

Advantages o Batch System

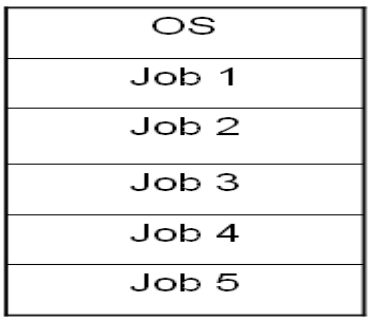
1. Move much of the work of the operator to the computer.
2. Increased performance since it was possible for job to start as soon as the previous job finished.

Disadvantages of Batch System

1. Turnaround time can be large from user standpoint.
2. Difficult to debug program.
3. A job could enter an infinite loop.
4. A job could corrupt the monitor, thus affecting pending jobs.
5. Due to lack of protection scheme, one batch job can affect pending jobs.

1.2.3 Multi-programmed Batch system

When two or more programs are in memory at the same time, sharing the processor is referred to the multiprogramming operating system. Multiprogramming assumes a single processor that is being shared. It increases CPU utilization by organizing jobs so that the CPU always has one to execute. Multiprogramming is a technique to execute number of programs simultaneously by a single processor. In Multiprogramming, numbers of processes reside in main memory at a time. The OS picks and begins to execute one of the jobs in the main memory. If any I/O wait happened in a process, then CPU switches from that job to another job, hence CPU in not idle at any time. Figure depicts the layout of multiprogramming system. The main memory consists of 5 jobs at a time, the CPU executes one by one.



Advantages:

1. High CPU utilization as it is never idle.
2. It appears that many programs are allotted CPU almost simultaneously.
3. Efficient memory utilization
4. Throughput increases

Disadvantage

1. Job may have different sizes, so some powerful memory management policy is desired to accommodate them in memory.
2. CPU scheduling is must because many jobs are ready to be run on the CPU.
3. The user cannot interact with the job when it is being executed.

1.2.4 Time sharing is multiprogramming

Time sharing, or multitasking, is a logical extension of multiprogramming. Multiple jobs are executed by switching the CPU between them. In this, the CPU time is shared by different processes, so it is called as "Time sharing Systems". Time slice is defined by the OS, for sharing CPU time between processes. Examples: Multics, Unix, etc.,

Characteristics:

- Using multiprogramming to handle multiple interactive jobs
- Processor's time is shared among multiple users
- Multiple users simultaneously access the system through terminals

A time sharing system allows many users to share the computer resources simultaneously. In other words, time sharing refers to the allocation of computer resources in time slots to several programs simultaneously. For example a mainframe computer that has many users logged on to it. Each user uses the resources of the mainframe i.e. memory, CPU etc. The

users feel that they are exclusive user of the CPU, even though this is not possible with one CPU i.e. shared among different users. As the system switches rapidly from one user to the other, a short time slot is given to each user for their executions.

Advantages

- More than one user can execute their task simultaneously.
- CPU Idle time is reduced and better utilization of resources.

Disadvantages

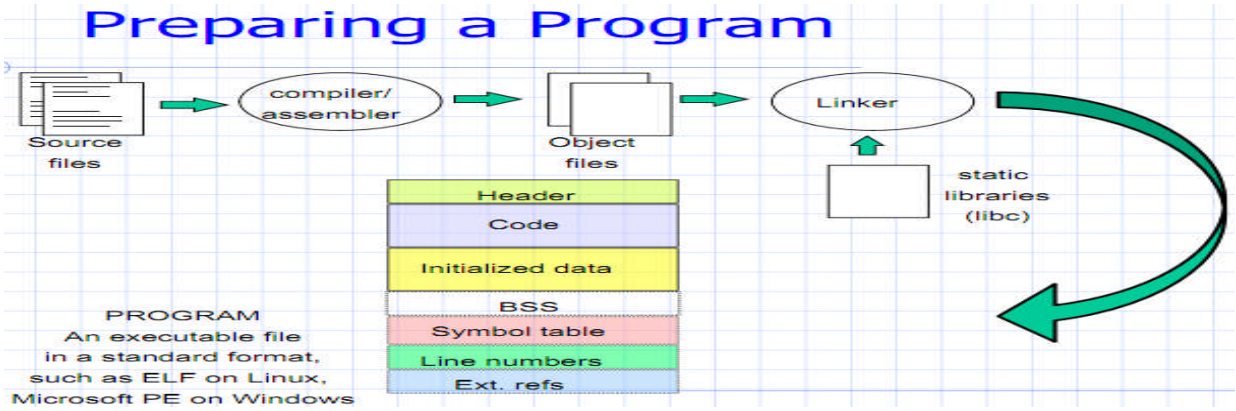
- Question of securing the security and integrity of user's data and programs.
- Since multiple processes are managed simultaneously, so it requires an adequate management of main memory.

Chapter 2: Process (17 hrs)

2.1 Introduction to process

What is a program?

A program is a file containing executable code (machine instructions) and data (information manipulated by these instructions) that together describes a computation. A program by itself is not a process. It is a static entity made up of program statement while process is a dynamic entity. Program contains the instructions to be executed by processor. A program takes a space at single place in main memory and continues to stay there. A program does not perform any action by itself.



Process

A process is a program in execution. The execution of a process must progress in a sequential fashion. A process is defined as an entity which represents the basic unit of work to be implemented in the system.

A process contains all the state of a program in execution:

- The code for the running programs.
- The data for the running program.
- The execution stack showing the state of all calls in progress.
- The program counter, indicating the next instruction.
- The set of CPU registers with current values.

S.N.	Component & Description
1	**Object Program:** Code to be executed.
2	**Data:** Data to be used for executing the program.
3	**Resources:** While executing the program, it may require some resources.
4	**Status:** Verifies the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

- The set of OS resources held by the program (references to open files, network connections)

Components of process are following.

Process States

As a process executes, it changes state. The state of a process is defined as the current activity of the process. Process state is shown in **5-State Model**. A process may not be in exactly two states when there is more than one process. For example, when one process has completed its I/O, it's cannot get back to CPU because another process may be using it. So this process is ready but can't run. Also process may be blocked for more than one reasons like:

- A process may be waiting for an event to occur; e.g. a process has created a child process and is waiting for it to return the result.
- A process may be waiting for a resource which is not available at this time.
- Process has gone to sleep for some time.
- As a process executes, it changes *state.*
- Process can have one of the following five states at a time.

S.N.	State & Description
1	**New:** The process is being created.
2	**Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
3	**Running:** Process instructions are being executed (i.e. The process that is currently being executed).
4	**Waiting:** The process is waiting for some event to occur (such as the completion of an I/O operation).
5	**Terminated:** The process has finished execution.

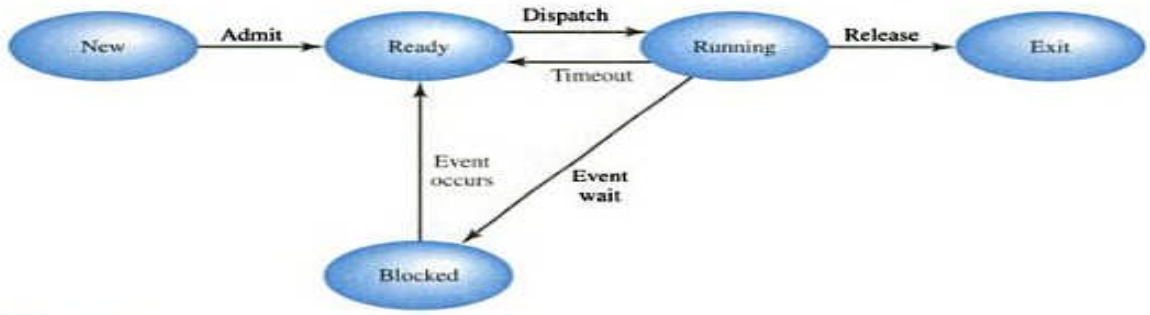


Fig. Five state process model

Process Control Block, PCB

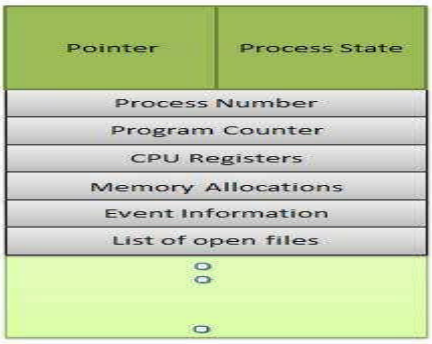
Each process is represented in the operating system by a process control block (PCB) also called a **task control block**. PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process. The PCB contains important information about the specific process including

- The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- Unique identification of the process in order to track "which is which" information.

- A pointer to parent process.
- Similarly, a pointer to child process (if it exists).
- The priority of process (a part of CPU scheduling information).
- Pointers to locate memory of processes.
- A register save area.
- The processor it is running on

PCB contains many pieces of information associated with specific processes which are described below.

S.N.	Information & Description
1	**Pointer:** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2	**Process State:** Process state may be new, ready, running, waiting and so on.
3	**Program Counter:** Program Counter indicates the address of the next instruction to be executed for this process.
4	**CPU registers:** CPU registers include general purpose register, stack pointers, index registers and accumulators' etc. number of register and type of register totally depends upon the computer architecture.
5	**Memory management information:** This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for de-allocating the memory when the process terminates.
6	**Accounting information:** This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.



Process control block includes CPU scheduling, I/O resource management, file management information etc…The PCB serves as the repository for any information which can vary from process to process. Loader/linker sets flags and registers when a process is created. If that process gets suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB. By this technique, the hardware state can be restored so that the process can be scheduled to run again

2.1.1 Process Model

All run able software on the computer is organized into a number of sequential processes that includes the operating system, which consists of a number of running processes. The CPU determines a schedule to run each of the processes for a specific duration. The rapid switching between running programs is called multiprogramming

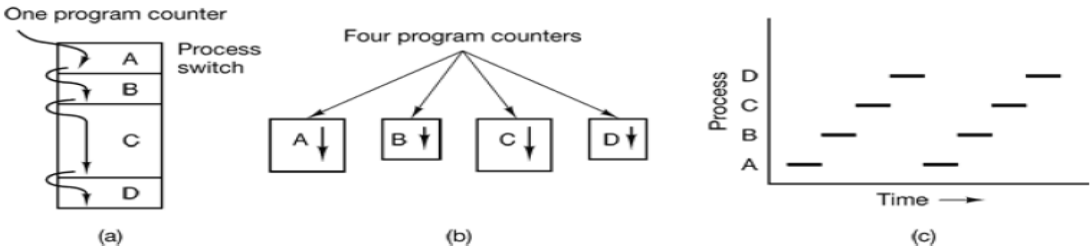
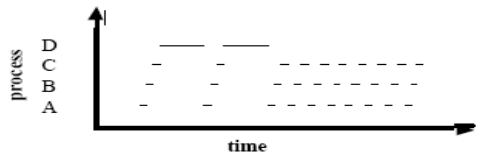


Figure a) multiprogramming of four programs. b) conceptual model of four independent sequential process. C) only one program is active at any instant.

In Figure (a) we see a computer multiprogramming four programs in memory. In Figure (b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory. In Figure (c) we see that viewed over a long enough time interval, all the processes have made progress, but at any

given instant only one process is actually running. All processes run for a fixed amount of time. Process A, B, C are 90% I/O bound, Process D is 100% CPU Bound

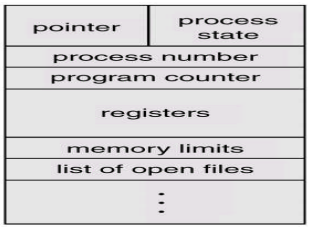


By performing a task switch when a process blocks we can greatly improve the processes execution time.

2.1.2 Implementation of process:-

Process Control block is used for storing the collection of information about the processes. The information of the Process is used by the CPU at the Run time. To implement the process model, Operating system maintains a **process table** with one entry per process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped. The various information which is stored into the PCB as followings:

- Name of the Process.
- State of the Process: Ready, Active, Wait.
- Resources allocated to the Process.
- CPU registers.
- Program Counter.
- Memory which is provided to the Process.
- CPU Scheduling information.
- Accounting information.
- Input and Output Devices used by the Process.
- Process ID or a Identification Number which is given by the CPU when a Process Request for a Service.



Goal: Save enough information so that a process can be restarted (ready to running) as if it had never been stopped. Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained on a machine with one CPU and many I/O devices. Associated with each I/O device class (e.g., floppy disks, hard disks, timers, terminals) is a location (often near the bottom of memory) called the **interrupt vector.** It contains the address of the interrupt service procedure.

Suppose that user process 3 is running when a disk interrupt occurs. User process 3's program counter, program status word, and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the disk interrupt vector. That is all the hardware does. From here on, it is up to the software, in particular, the interrupt service procedure. All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler.

Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is. When this routine is finished, it calls a process 3 procedure to do the rest of the work for this specific interrupt type. When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next. After that, control is passed back to the assembly language code to load up the registers and memory map for the now-current process and start it running.

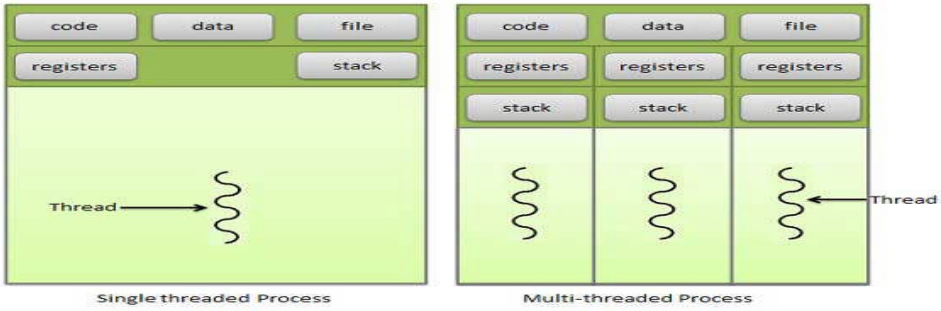
2.1.3Threads

A thread is the smallest unit of processing that can be performed in an OS. In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads. A process has an address space and single thread of control. It is possible to have a single process that has multiple threads of control.

A thread is very similar to a process

- Each thread gets scheduled just like a process
- Each thread gets a private address space.
- Each thread has a state (running, ready, blocked).
- Each thread gets a unique ID.
- A thread also gets access to global process information

A thread is sometimes referred to as a ***lightweight process*.** A thread is a flow of execution through the process code, with its own program counter, system registers and stack. 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.



Single threaded Process Multi-threaded Process

Difference between Process and Thread

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

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

Advantages of Thread

1. Thread minimizes context switching time.
2. Use of threads provides concurrency within a process.
3. Efficient communication.
4. Economy- It is more economical to create and context switch threads.
5. Utilization of multiprocessor architectures to a greater scale and efficiency.

Types of Thread

Threads are implemented in following two ways

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

User Level Threads

In this case, application manages 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 begins with a single thread and begins running in that thread.

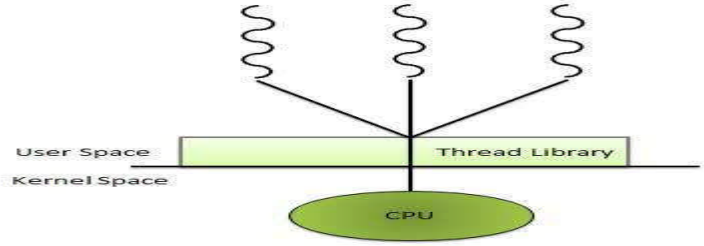
Advantages

1. Thread switching does not require kernel mode privileges.
2. User level thread can run on any operating system.
3. Scheduling can be application specific in the user level thread.

4. User level threads are fast to create and manage

Disadvantage

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



Kernel Level Threads

In this case, thread management 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	**Disadvantages**
1. Kernel can simultaneously schedule multiple threads from the same process on multiple processes. 2. If one thread in a process is blocked, the Kernel can schedule another thread of the same process. 3. Kernel routines themselves can multithreaded.	1. Kernel threads are generally slower to create and manage than the user threads. 2. Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

Multithreading Models

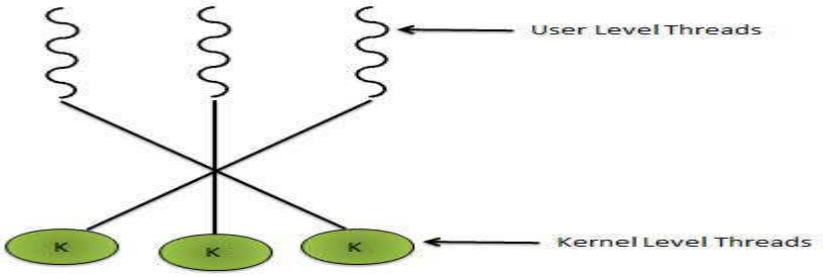
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.

Multithreading models are three types

1. Many to many relationship.
2. Many to one relationship.
3. One to one relationship.

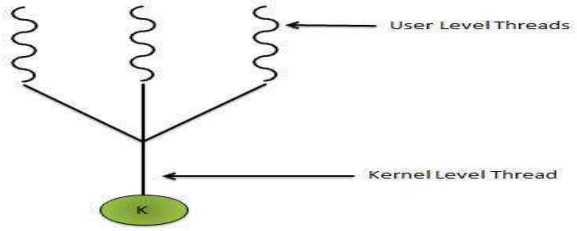
Many to Many Model

In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine. Following diagram shows the many to many models. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



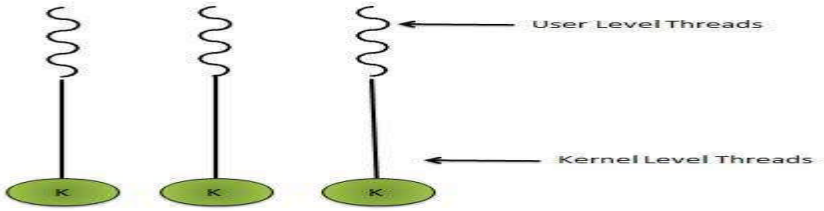
Many to One Model

Many to one model maps many user level threads to one Kernel level thread. Thread management is done in user space. When thread makes a blocking system call, the entire process will be blocks. 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 system does not support them then Kernel threads use the many to one relationship modes.



One to One Model

There is one to one relationship of user level thread to the kernel level thread. This model provides more concurrency than the many to one model. It supports multiple threads to execute in parallel on microprocessors. Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, Windows NT and windows 2000 use one to one relationship model.



Difference between User Level & Kernel Level Thread

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

2.2 Inter Process Communication (IPC):

Inter process communication (IPC) is a set of programming interfaces that allow a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in

the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods. A capability supported by some operating systems that allows one *process* to communicate with another process. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems such as DOS

There are two models:

1. **Shared Memory:-**

In shared memory model. The co operating process shares a region of memory for sharing of information. Some operating systems use the supervisor call to create a share memory space. Similarly, some operating system use file system to create RAM disk, which is a virtual disk created in the RAM. The shared files are stored in RAM disk to share the information between processes. The shared files in RAM disk are actually stored in the memory. The Process can share information by writing and reading data to the shared memory location or RAM disk.

2. **Message Passing:-**

In this model, data is shared between process by passing and receiving messages between co-operating process. Message passing mechanism is easier to implement than shared memory but it is useful for exchanging smaller amount of data. In message passing mechanism data is exchange between processes through kernel of operating system using system calls. Message passing mechanism is particularly useful in a distributed environment where the communicating processes may reside on different components connected by the network. For example, A data program used on the internet could be designed so that chat participants communicate with each other by exchanging messages. It must be noted that passing message technique is slower than shared memory technique.

A message contains the following information:

- Header of message that identifies the sending and receiving processes
- Block of data
- Pointer to block of data
- Some control information about the process

Message passing allows processes to communicate and to synchronies their actions without sharing the same address space.IPC facility provides two operations

- send(message)
- receive (message)

Message passing requires a communication link between two processes

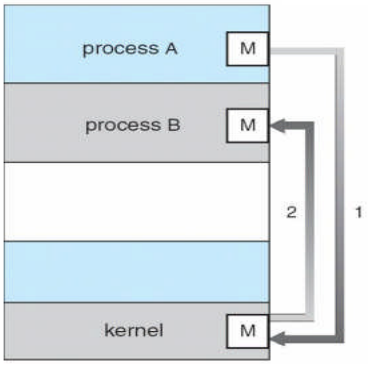


Fig. Message Passing

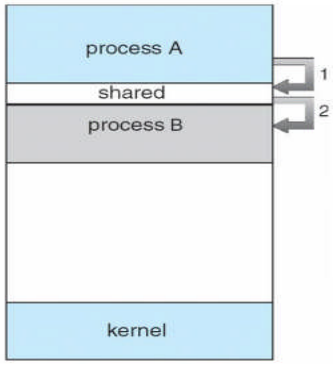


Fig. Shared memory

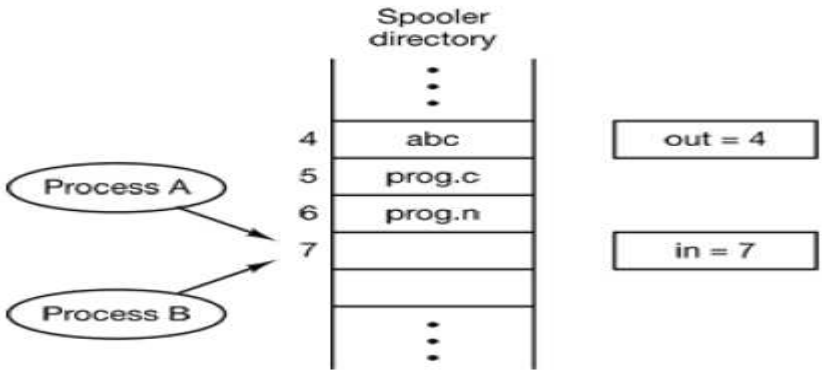
There are three issues:

1. How one process can pass information to another?
2. The second has to do with making sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 1 MB of memory).
3. The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print.

2.2.1 Race Condition

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file: the location of the shared memory does not change the nature of the communication or the problems that ariseTo see how inter process communication works in practice, let us consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory.** Another process, the **printer daemon,** periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory. Imagine that our spooler directory has a very large number of slots,

numbered 0, 1, 2, …, each one capable of holding a file name. Also imagine that there are two shared variables, *out* , which points to the next file to be printed, and *in* , which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing.



Two processes want to access shared memory at the same time.

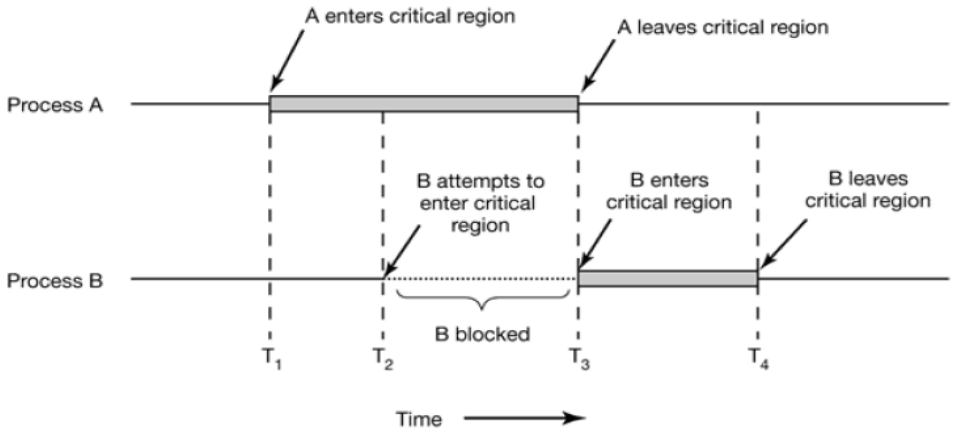
For example, two processes P_A and P_B share the global variable X. In the execution, process P_A updates the variable X to value 2 and process P_B updates X to value 3. These two processes update the value but in this condition last update value will be the final value of X. If another process P_C wants to use the value by executing X=X+1. This gives the variable X with the value 4 but the value assign by the process P_A is lost and in that variable P_B updates another value in the variable. This situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**.

A **race condition** is a situation where two or more control threads are working with the same memory elements, and simultaneously decide to write different values in the same place.

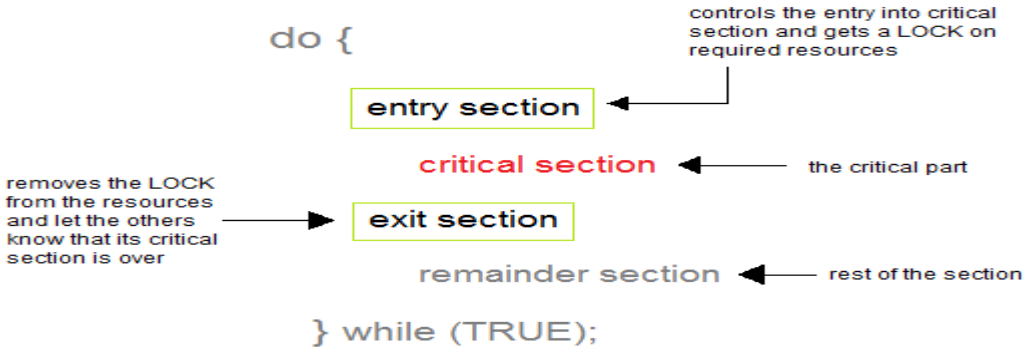
Without any priority scheme, lock or other control mechanism, the persisting value of the memory location will be the one that was written last, and which of the two values that is becomes subject to whatever random circumstances caused one thread to run more quickly than the other. In a network, a race condition may occur if two users attempt to access an available channel at the same instant, and neither computer receives notification the channel is occupied before the system grants access.

2.2.2 Critical section

The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion,** that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it. The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process has to access shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section.** If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.



Process *A* enters its critical region at time *T* 1 , A little later, at time *T* 2 process *B* attempts to enter its critical region but fails because another process is already in its critical region only one at a time. Consequently, *B* is temporarily suspended until time *T* 3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at *T* 4) and we are back to the original situation with no processes in their critical regions. In critical section the process may be changing common variable, updating a table, writing a file and so on. Only one process can enter into the critical section at one time.Figure below shows the general structure of process p. Each process containing three section. They are entry section, exit section and remainder section.



Solution to Critical Section Problem

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

1. Mutual Exclusion: No two processes may be simultaneously inside the same critical section.
2. Bounded Waiting: No process should have to wait forever to enter a critical section.
3. Progress: No process running outside its critical region may block other processes.
4. Arbitrary Speed: No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed).

2.2.3 Mutual Exclusion with Busy waiting

For achieving mutual exclusion, while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble. In computer science, **mutual exclusion** refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions.

Requirements for Mutual Exclusion

Mutual exclusion should meet the following requirement:

1. Mutual exclusion must be enforced only one process at a time is allowed into its critical section, among all processes that have critical section for the same resource or shared object.
2. A process that halts in its non-critical section must do so without interfering with other processes.

3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock or starvation.
4. When no process is in a critical section, any process that request entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speed or number of processors.
6. A process remains inside its critical section for a finite time only.

There are various proposals for achieving mutual exclusion. They are

1. Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Problem in this approach

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.

1. Suppose that one of them did it and never turned them on again? That could be the end of the system. Furthermore if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

2. On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

Conclusion: The conclusion is disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

2. Lock Variables

Another method is to assign a lock variable. It is a software solution. Consider a single shared variable lock initially set to zero. Lock=0 when no process is its critical section, lock = 1 when a process is in its critical section.

Working of this scheme

If a process wants to enter its critical section, it tests the lock. If lock = 1, it waits. If lock = 0 the process sets it to 1 and enters its critical section. After finishing its critical section the process sets the Lock back to zero.

Thus, a 0 means that no process is in its critical region and a 1 means that some process is in its critical region.

Problem in this approach

Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Conclusion: This problem can be solved by first reading out the lock value, then checking it again just before storing into it, but that really does not help. The race now occurs if the second process modifies the lock just after the first process has finished its second check.

3. Strict Alternation

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspects turn, finds it to be 0, and enters in its critical section sets turn to 1. Process B also finds it to be 1 and sits in a loop continually testing 'turn' to see when it becomes 0.

In Figure, the integer variable *turn* , initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn* , finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Busy waiting for a process refers to checking a variable for a particular expected value. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock** .

a) Process 0	b) Process 1
while (TRUE) {	while (TRUE) {
while (turn != 0)/* loop */ ;	while (turn != 1);/* loop */ ;
critical_region();	critical_region();
turn = 1;	turn = 0;
noncritical_region();}	noncritical_region();}

(a) Process 0. (b) Process 1.

Essentially there is a shared variable, which is initially 0:

```
int turn = 0;
```

This means that only process 0 is allowed to enter its critical region. If process 2, for example, wishes to enter its critical region, it must busy wait for the variable to change to 2.

```
void enterCriticalSection(int myProcessNumber)
{
        while(turn!=myProcessNumber)
        { //While it's not my turn,// ...busywait.
         }
  }
```

Eventually, process 0 enters its critical region, and when it exits, it sets the shared variable to 1.

```
VoidleaveCriticalSection(){
```

turn = (turn + 1) % NUMBER_OF_PROCESSES; // Let the next process have a turn. }

When process 1 enters its critical region, it sets the variable to 2 when it leaves. Process 2 would still be waiting by this time, and would enter its critical region as soon as the variable goes to 2. If it were the last of the cooperating processes, it would set the variable back to 0 again when it left its critical region.

Problem in this approach

1. Requires you know in advance how many processes will want to share the memory
2. A process may be held up indefinitely waiting for its turn, even if no other process is in its critical section (which violates one of the requirements)

This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region. Going back to the spooler directory discussed above, if we now associate

the critical region with reading and writing the spooler directory, process 0 would not be allowed to print another file because process 1 was doing something else.

In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

4. Peterson's Solution

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual exclusion problem that does not require strict alternation. In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's solution obsolete. Peterson's algorithm is shown in Figure. This algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show the prototypes in this or subsequent examples.

```
#define FALSE 0
#define TRUE 1
#define N 2 /* number of processes */
int turn; /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */
void enter_region(int process) /* process is 0 or 1 */
{
int other; /* number of the other process */
other = 1 - process; /* the opposite of process */
interested[process] = TRUE; /* show that you are interested */
turn = process; /* set flag */
while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region (int process) /* process, who is leaving */
{
interested[process] = FALSE; /* indicate departure from critical region */
```

```
}
```

Figure: Peterson's solution for achieving mutual exclusion.

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter _region.* It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_region,* it will hang there until *interested* [0] goes to *FALSE,* an event that only happens when process 0 calls *leave_region* to exit the critical region. Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn.* Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region. Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and Pi. For convenience, when presenting P,-, we use *Pj* to denote the other process; that is, j equals 1 — i.

Peterson's solution requires two data items to be shared between the two processes:

```
int turn;
boolean f l a g [2]
```

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P; is allowed to execute in its critical section. The flag array is used to indicate if a process *is ready* to enter its critical section. For example, if f lag[i] is true, this value indicates that P; is ready to enter its critical section. With an explanation of these data structures complete. To enter the critical section, process P, first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

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

            critical section

        flag[i] = FALSE;

            remainder section

} while (TRUE);
```

Problem in this approach

Suppose both process start executing concurrently .P0 sets flag[0]=1 and die. Then P1 starts. It's while condition will be satisfied as flag[0]=1 (set by P0 and turn =0) and it will stuck in this loop forever which is a dead lock.

5. The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Many computers, especially those designed with multiple processors in mind, have an instruction (Test and Set Lock) that works as follows.

TSL RX, LOCK

It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock.* The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

To use the TSL instruction, we will use a shared variable, *lock* , to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

Solution to mutual exclusion

There a four-instruction subroutine in a fictitious (but typical) assembly language is shown. The first instruction copies the old value of *lock* to the register and then sets *lock* to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is simple. The program just stores a 0 in *lock* . No special instructions are needed.

```
enter_region:
TSL REGISTER, LOCK | copy lock to register and set lock to 1
CMP REGISTER,#0 | was lock zero compare?
JNE enter_region | if it was non zero, lock was set, so loop
RET | return to caller; critical region entered
leave_region:
MOVE LOCK,#0 | store a 0 in lock
RET | return to caller
```

Figure: Entering and leaving a critical region using the TSL instruction.

Before entering its critical region, a process calls *enter_region,* which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls *leave_region* , which stores a 0 in *lock* .

Problem in this approach

As with all solutions based on critical regions, the processes must call *enter_region* and *leave_region* at the correct times for the method to work. If a process cheats, the mutual exclusion will fail. Locks are tremendously difficult to get right, hard to reason about, have scalability issues and behave badly in the face of failures, so they're not an ideal (but definitely practical) solution.

Mutual Exclusion with Sleep and Wakeup

In busy waiting CPU time is wasted for continuous checking of the processes. So there is another way by which mutual exclusion can be solved that is sleep and wakeup puts much less stress on the CPU compared to the busy waiting solutions. As its name suggests, it uses two system calls: Sleep and Wakeup. If a process is unable to enter a critical section, it is put to sleep/temporarily suspend. Afterwards, when the critical section is no longer occupied, the

process is woken up. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

1. Priority inversion problem

In computer science, **priority inversion** is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks. This violates the priority model that high priority tasks can only be prevented from running by higher priority tasks and briefly by low priority tasks which will quickly complete their use of a resource shared by the high and low priority tasks.

Consider a computer with two processes, *H*, with high priority and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever

2. The Producer-Consumer Problem

In computing, the **producer–consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Problem

1. The producer wants to insert information in the buffer; but the buffer is full
2. The consumer wants to remove an item from the buffer; but the buffer is empty
3. Race conditions at the buffer.

Race condition: To keep track of the number of items in the buffer, a variable, *count* is needed. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*. The consumer's code is similar: first test *count* to see if it is

0. If it is, go to sleep, if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up. The buffer is empty and the consumer has just read *count* to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up. Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

```
#define N 100 /* number of slots in the buffer */
int count = 0; /* number of items in the buffer */
void producer (void)
{
int item;
while (TRUE) { /* repeat forever */
item = produce_item(); /* generate next item */
if (count == N) sleep(); /* if buffer is full, go to sleep */
insert_item (item); /* put item in buffer */
count = count + 1; /* increment count of items in buffer */
if (count == 1) wakeup(consumer); /* was buffer empty? */
}}
void consumer(void)
{
int item;
while (TRUE) { /* repeat forever */
if (count == 0) sleep(); /* if buffer is empty, got to sleep */
item = remove_item(); /* take item out of buffer */
count = count - 1; /* decrement count of items in buffer */
if (count == N - 1) wakeup(producer); /* was buffer full? */
consume_item(item); /* print item */
}}
```

Figure: The producer-consumer problem

2.2.5 Semaphores

The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra in 1963. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Example: Suppose a library has 10 identical study rooms, to be used by one student at a time. To prevent disputes, students must request a room from the front desk if they wish to make use of a study room. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that one room has become free. In the simplest implementation, the clerk at the front desk does not need to keep track of which rooms are occupied or who is using them, nor does she know if any given room is actually being used, only the number of free rooms available, which she only knows correctly if all of the students actually use their room while they've signed up for them and return them when they're done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. Once access to a room is granted, the room can be used for as long as desired, and so it is not possible to book rooms ahead of time. In this scenario the front desk count-holder represents a counting semaphore, the rooms are the resources, and the students represent processes. The value of the semaphore in this scenario is initially 10. When a student requests a room she is granted access and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7 and so on. If someone requests a room and the resulting value of the semaphore would be negative, they are forced to wait until a room is freed (when the count is increased from 0).

Problem in this approach

1. Requesting a resource and forgetting to release it.
2. Releasing a resource that was never requested.
3. Holding a resource for a long time without needing it.
4. Using a resource without requesting it first (or after releasing it

Semaphores are commonly used for two purposes: to share a common memory space and to share access to files.

Operation in semaphore

There are two operations, the down () and the up() operations on a semaphore that are atomic actions.

1. Down () or wait operation: Check if the semaphore is 0: if it is, the process goes to sleep; otherwise, subtract 1 from the semaphore. Wait for semaphore to become positive and then decrement

```
P(s)
{
while (s <= 0) ;
s--;
}
```

2. Up () or signal operation: If one or more processes are sleeping on the semaphore, choose one and wake it up; otherwise, add 1 to the semaphore.

```
V(s){ s++; }
```

A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending. Dijkstra proposed having two operations, down and up (generalizations of sleep and wakeup respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it and possibly going to sleep, is all done as a single, indivisible **atomic action?** It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked.

This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions. The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down. For example, we have semaphore s, and two processes, P1 and P2 that want to enter their critical sections at the same time. P1 first calls wait(s). The value of s is decremented to 0 and P1 enters its critical section. While P1 is in its critical section, P2 calls wait(s), but because the value of s is zero, it must wait until P1 finishes its critical section and executes signal(s). When P1 calls signal, the value of s is incremented to 1, and P2 can then proceed to execute

in its critical section (after decrementing the semaphore again). Mutual exclusion is achieved because only one process can be in its critical section at any time.

Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

Solving the Producer-Consumer Problem using Semaphores

Semaphores solve the lost-wakeup problem. The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore. Be sure you understand that using TSL to prevent several CPUs from accessing the semaphore at the same time is quite different from busy waiting by the producer or consumer waiting for the other to empty or fill the buffer. The semaphore operation will only take a few microseconds, whereas the producer or consumer might take arbitrarily long.

```
#define N 100 /* number of slots in the buffer */
typedef int semaphore;/* semaphores are a special kind of int */
semaphore mutex = 1;/* controls access to critical region */
semaphore empty = N;/* counts empty buffer slots */
semaphore full = 0;/* counts full buffer slots */
void producer(void)
{
int item;
while (TRUE) { /* TRUE is the constant 1 */
item = produce_item(); /* generate something to put in buffer */
down(&empty); /* decrement empty count */
down(&mutex);/* enter critical region */
insert_item(item); /* put new item in buffer */
up(&mutex); /* leave critical region */
```

```
up(&full); /* increment count of full slots */
}}
void consumer(void)
{
int item;
while (TRUE) {/* infinite loop */
down(&full);/* decrement full count */
down(&mutex);/* enter critical region */
item a= remove_item();/* take item from buffer */
up(&mutex);/* leave critical region */
up(&empty);/* increment count of empty slots */
consume_item(item);/* do something with the item */
}}
```

Figure: The producer-consumer problem using semaphores.

This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores.** If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Application of semaphore

1. The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables.
2. The other use of semaphores is for **synchronization.** The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty.

Mutex: A mutual exclusion (mutex) is a program object that prevents simultaneous access to a shared resource. This concept is used in concurrent programming with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, thus a mutex with a unique name is created when a program starts. When

a thread holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. Upon releasing the resource, the thread unlocks the mutex.

Drawback of semaphore

1. They shared global variable.
2. Access to semaphore can come from anywhere in the program
3. There is control over the proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore control access.

2.2.4 Monitors

A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

```
monitor example
integer i ;
condition c ;
procedure producer ( );
end ;
procedure consumer ( );
end ;
end monitor ;
```

Figure: A monitor.

Monitor simply defines a lock and a condition variable for managing concurrent access to the shared data. Lock insures that only one single thread is active in the monitor at a time by providing mutual exclusion. Condition variable enable thread to go to sleep inside of critical section by releasing their lock at the same time it puts the thread to go to sleep.

Monitor operation are:

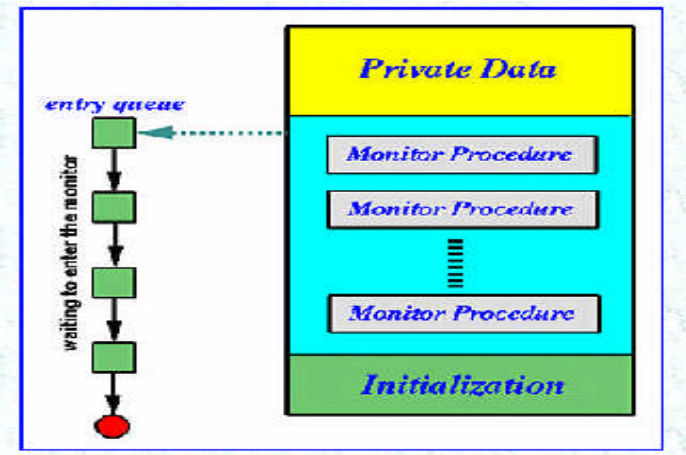
1. Encapsulate the shared data that need to be protect.
2. Acquire the mutex at the start.
3. Operate on the shared data
4. Temporarily release the mutex if it cannot complete

5. Reacquires the mutex when it can continue
6. Release the mutex at the end

Characteristics of monitor

1. Only one process can be active within the monitor at a time
2. The local data variable are accessible only by the monitor's procedures and not by any external procedure.
3. A process enters the monitor by invoking one of its procedures
4. Monitor provides high level of synchrozation. The synchronization of the process is accomplished through two special operation namely, wait and signal, which are executed with in the monitors procedure
5. Monitors are a high level data abstraction tool combining three features:
    a. Shared data
    b. Operation on data
    c. synchronization

A monitor has ***four*** components as shown below: initialization, private data, monitor procedures, and monitor entry queue. The ***initialization*** component contains the code that is used exactly once when the monitor is created, the ***private data*** section contains all private data, including private procedures that can only be used *within* the monitor. Thus, these private items are not visible from outside of the monitor. The ***monitor procedures*** are procedures that can be called from outside of the monitor. The ***monitor entry queue*** contains all threads that called monitor procedures but have not been granted permissions.



Monitor is a software module consisting of one or more procedure.

Syntax:

Monitor monitor-name

{

Declaration of share variable

Procedure body

P1()

{

}

P2()

{

}

Pn()

{

}

{

Initialization code

}

How monitor work in the process

Typically, when a process calls a monitor procedure, the first few instructions of the procedure will check to see, if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter. It is up to the compiler to implement the mutual exclusion on monitor entries, but a common way is to use a mutex or binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong.

A condition variable is a queue of thread waiting for something inside a critical section. There are two operation wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, *full*. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now. This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the

condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal. If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

```
monitor ProducerConsumer
condition full , empty ;
integer count ;
procedure insert (item : integer );
begin
if count = N then wait (full );
insert_item (item );
count := count + 1:
if count = 1 then signal (empty )
end ;
function remove : integer ;
begin
if count = 0 then wait (empty);
remove = remove_item ;
count := count - 1;
if count = N - 1 then signal (full )
end ;
count := 0;
end monitor ;
procedure producer ;
begin
while true do
begin
item = produce_item ;
ProducerConsumer .insert (item )
end
end ;
procedure consumer ;
begin
```

```
while true do
begin
item = ProducerConsumer .remove ;
consume_item (item )
end
end ;
```

Figure: producer-consumer problem with monitors.

Only one monitor procedure at a time is active. The buffer has *N* slots. The operations wait and signal look similar to sleep and wakeup. They *are* very similar, but with one crucial difference: sleep and wakeup failed because while one process was trying to go to sleep, the other one was trying to wake it up. But in monitors it that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the wait operation without having to worry about the possibility that the scheduler may switch to the consumer just before the wait completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer run able. By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error-prone than with semaphores. Still, they too have some drawbacks.

Only one program may modify local data of the monitor at a given time. If more than one program attempts to enter at the same time, only one will succeed, and the remaining programs will remain on a queue. Since monitors are used to provide access to resources, there may be cases in which a monitor procedure has been entered, but the resource is not available to be granted yet. Hence, we need wait & signal operations. If a procedure calls wait, the calling program will block until some other procedure calls signal. When wait is called, the lock on the monitor is released, such that another program may enter a monitor procedure that will call signal. When a procedure calls signal, the lock on the monitor is also released, and another program that previously called wait is run immediately.

Drawback of monitors

1. Major weakness of monitors is the absence of concurrency if a monitor encapsulates the resources, since only one process can be active within a monitoe at a time.
2. There is the possibility od deadlock in the case of nested monitor calls
3. Monitor concept is its lack of implementation most commonly used programming language
4. Monitors cannot easily be added if they are not natively supported by the language

Similarities and difference between semaphore and monitors

Even though both the semaphores and monitors are used to achieve mutual exclusion in parallel programming environments, they differ in the techniques used to achieve this task. In monitors, the code that is used to achieve mutual exclusion is in a single place and is more structured, while code for semaphores are distributed as wait and signal function calls. Also, it is very easy to make mistakes when implementing semaphores, while there is a very little chance to make mistakes when implementing monitors. Further, monitors use condition variables, while semaphores do not.

2.2.5 Message Passing

Message passing is processes to communicate with each other without resorting to shared variables. In computer science, **message passing** sends a message to a process (which may be an actor or object) and relies on the process and the supporting infrastructure to select and invoke the actual code to run. Message passing differs from conventional programming where a process, subroutine, or function is directly invoked by name. Message passing is key to some models of concurrency and object-oriented programming. Message passing is used ubiquitously in modern computer software. It is used as a way for the objects that make up a program to work with each other and as a way for objects and systems running on different computers (e.g., the Internet) to interact. Message passing may be implemented by various mechanisms, including channels.

Message passing involves the passing of messages between processes using simple primitives to send and receive messages. Communication in the message passing paradigm, in its

simplest form, is performed using the send() and receive() primitives. The syntax is generally of the form:

send (receiver, message)

receive (sender, message)

The send() primitive requires the name of the destination process and the message data as parameters. The addition of the name of the sender as a parameter for the send() primitive would enable the receiver to acknowledge the message. The receive() primitive requires the name of the anticipated sender and should provide a storage buffer for the message.

Decisions have to be made, at the operating system level, regarding the semantics of the send() and receive() primitives. The most fundamental of these are the choices between blocking and non-blocking primitives and reliable and unreliable primitives.

Blocking/non-blocking: A blocking send () blocks the process and does not execute the following instruction until the message has been sent and the message buffer has been cleared. In the same way a blocking receive blocks at the receive () until the message arrives. A non-blocking send returns control to the caller immediately. The message transmission is then executed concurrently with the sending process. This has the advantage of not leaving the CPU idle while the send is being completed. However, the disadvantage of this approach is that the sender does not know and will not be informed when the message buffer has been cleared. To overcome this the kernel can either make a copy of the message buffer or send an interrupt to the sender when the message buffer has been cleared. At the implementation level, although non-blocking primitives are flexible they make programming and debugging very difficult, hence, for the sake of easier programming, blocking primitives are often chosen.

Buffered/unbuffered messages: An unbuffered receive() means that the sending process sends the message directly to the receiving process rather than a message buffer. The address, receiver, in the send() is the address of the process, but in the case of an buffered send() the address is that of the buffer. There is a problem, in the unbuffered case, if the send() is called before the receive() because the address in the send does not refer to any existing process on the server machine. Buffered messages are saved in a buffer until the server process is ready to receive them. They can best be implemented through a mechanism in the operating system which can keep a backlog of sends, a port in which messages are queued waiting until requested by the receiver. The buffer capacity can either be bounded, where a predetermined number of messages can be stored, or unbounded. An unbounded buffer could be

implemented using dynamic memory allocation, thus its capacity would be fixed only by the size of available memory.

Reliable/unreliable send: Unreliable send() sends a message to the receiver and does not expect acknowledgement of receipt, nor does it automatically retransmit the message to ensure receipt. A reliable send() guarantees that, by the time the send() is complete, the message has been received. The primitive itself handles acknowledgements and retransmission in response to lost messages. At the implementation level the operating system must wait only for a specified length of time, so that a process does not remain blocked indefinitely waiting for a response from a receiver that has terminated. Lost messages are handled either by the operating system retransmitting the message or informing the sender of the message's loss or by the sender detecting the loss itself **Direct/indirect communication:** Ports allow indirect communication. Messages are sent to the port by the sender and received from the port by the receiver. Direct communication involves the message being sent direct to the process itself, which is named explicitly in the send, rather than to the intermediate port.

Fixed/variable size messages: Fixed size messages have their size restricted by the system. The implementation of variable size messages is more difficult but makes programming easier; the reverse is true for fixed size messages. Passing data by reference/value/address mapping — Data, in message passing, is often passed by value, since the processes execute in separate address spaces. However, another parameter passing mechanism is available which would be suitable for a message passing system. This is referred to as call-by-copy/restore. The variable is essentially passed by value but the returned value overwrites the original value so that the final result is the same as if it were passed by reference.

There are various design issue that arises in transmitting the message from one process to another process.

1. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.
2. What happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a

duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks.

3. Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that he is communicating with the real file server, and not with an imposter?
4. At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor.

The Producer-Consumer Problem with Message Passing

How the producer-consumer problem can be solved with message passing and no shared memory. We assume that all messages are the same size and that messages is sent but not received are buffered automatically by the operating system.

Solution

A total of *N* messages is used, analogous to the *N* slots in a shared memory buffer. The consumer starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance. If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.

```
#define N 100 /* number of slots in the buffer */
void producer(void)
{
int item;
message m;/* message buffer */
while (TRUE) {
item = produce_item( );/* generate something to put in buffer */
```

```
receive(consumer, &m);/* wait for an empty to arrive */
build_message (&m, item);/* construct a message to send */
send(consumer, &m);/* send item to consumer */
}}
void consumer(void) {
int item, i;
message m;
for (i = 0; i < N; i++) send(producer, &m);/* send N empties */
while (TRUE) {
receive(producer, &m);/* get message containing item */
item = extract_item(&m);/* extract item from message */
send(producer, &m);/* send back empty reply */
consume_item(tem);/* do something with the item */
}}
```

Figure: The producer-consumer problem with *N* messages.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way is to assign each process a unique address and have messages be addressed to processes. A different way is to invent a new data structure, called a **mailbox.** A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes are used, the address parameters, in the send and receive calls, are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one. For the producer-consumer problem, both the producer and consumer would create mailboxes large enough to hold *N* messages. The producer would send messages containing data to the consumer's mailbox, and the consumer would send empty messages to the producer's mailbox. When mailboxes are used, the buffering mechanism is clear: the destination mailbox holds messages that have been sent to the destination process but have not yet been accepted. The other extreme from having mailboxes is to eliminate all buffering.

2.2 Classical IPC problems

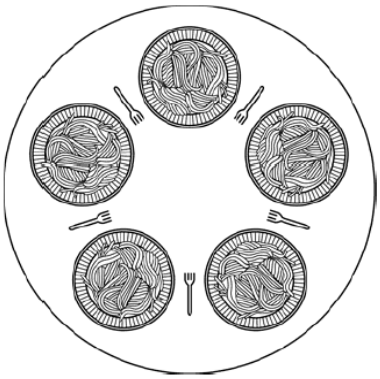
Some well-known problems are:

1. Dining Philosophers Problem
2. The Readers and Writers Problem
3. The Sleeping Barber Problem

2.2.1 The Dining Philosophers Problem

There are N philosophers sitting around a circular table eating spaghetti and discussing philosophy. The problem is that each philosopher needs 2 forks to eat, and there are only N forks, one between each 2 philosophers. Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.

• Philosophers eat/think
• Eating needs 2 forks
• Pick one fork at a time
• How to prevent deadlock



Lunch time in the Philosophy Department

The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible. One idea is to instruct each philosopher to behave as follows:

• think until the left fork is available; when it is, pick it up
• think until the right fork is available; when it is, pick it up
• eat
• put the left fork down
• put the right fork down
• repeat from the start

When a philosopher gets hungry, she tries to acquire her left and right fork, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

```
#define N 5/* number of philosophers */
void philosopher(int i)/* i: philosopher number, from 0 to 4 */
{
while (TRUE) {
think( ); /* philosopher is thinking */
take_fork(i); /* take left fork */
take_fork((i+1) % N);/* take right fork; % is modulo operator */
eat(); /* yum-yum, spaghetti */
put_fork(i); /* Put left fork back on the table */
put_fork((i+1) % N);/* put right fork back on the table */
}}
```

Figure: A non solution to the dining philosophers problem.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, and picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation. The solution presented below is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, state, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move into eating state only if neither neighbor is eating. Philosopher i's neighbors are defined by the macros LEFT and RIGHT. In other words, if i is 2, LEFT is 1 and RIGHT is 3

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as

its main code, but the other procedures, *take_forks* , *put_forks*, and *test* are ordinary procedures and not separate processes.

```
#define N5/* number of philosophers */
#define LEFT(i+N-1)%N/* number of i's left neighbor */
#define RIGHT(i+1)%N/* number of i's right neighbor */
#define THINKING0/* philosopher is thinking */
#define HUNGRY1/* philosopher is trying to get forks */
#define EATING2/* philosopher is eating */
typedef int semaphore;/* semaphores are a special kind of int */
int state[N];/* array to keep track of everyone's state */
semaphore mutex = 1;/* mutual exclusion for critical regions */
semaphore s[N];/* one semaphore per philosopher */
void philosopher (int i)/* i: philosopher number, from 0 to N-1 */
{
while (TRUE) {/* repeat forever */
think();/* philosopher is thinking */
take_forks(i);/* acquire two forks or block */
eat();/* yum-yum, spaghetti */
put_forks(i);/* put both forks back on table */
}}
void take_forks(int i)/* i: philosopher number, from 0 to N-1 */
{
down(&mutex); /* enter critical region */
state[i] = HUNGRY;/* record fact that philosopher i is hungry */
test(i);/* try to acquire 2 forks */
up(&mutex); /* exit critical region */
down(&s[i]); /* block if forks were not acquired */
}
void put_forks(i)/* i: philosopher number, from 0 to N-1 */
{
down(&mutex); /* enter critical region */
state[i] = THINKING;/* philosopher has finished eating */
test(LEFT); /* see if left neighbor can now eat */
```

```
test(RIGHT);/* see if right neighbor can now eat */
up(&mutex); /* exit critical region */
}
void test(i)/* i: philosopher number, from 0 to N-1 */
{
if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
state[i] = EATING;
up(&s[i]);
}}
```

Figure: A solution to the dining philosophers problem.

2.2.2 The Readers and Writers Problem

In this problem there are number of reader and writer. Reader and writer are using the shared file.

Condition:

1. If reader is reading on the file no other writer can write on that file.
2. At the same time both reader and writer cannot read and write on the shared file.
3. No two writers are allowed to access the file.

Solution

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

```
typedef int semaphore;/* use your imagination */
semaphore mutex = 1;/* controls access to 'rc' */
semaphore db = 1;/* controls access to the database */
int rc = 0;/* # of processes reading or wanting to */
void reader(void)
{
while (TRUE) {/* repeat forever */
```

```
down(&mutex); /* get exclusive access to 'rc' */
rc = rc + 1;/* one reader more now */
if (re == 1) down(&db); /* if this is the first reader… */
up{&mutex);/* release exclusive access to 'rc' */
read_data_base();/* access the data */
down(&mutex);/* get exclusive access to 'rc' */
rc = rc - 1; /* one reader fewer now */
if (rc == 0) up(&db);/* if this is the last reader… */
up(&mutex);/* release exclusive access to 'rc' */
use_data_read();/* noncritical region */
}}
void writer(void)
{
while (TRUE) {/* repeat forever */
think_up_data(); /* noncritical region */
down(&db);/* get exclusive access */
write_data_base(); /* update the data */
up(&db); /* release exclusive access */
}}
```

Figure: A solution to the readers and writers problem.

2.2.3 The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and *n* chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Figure. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers without getting into race conditions. This problem is similar to various queuing situations, such as a multi-person helpdesk with a computerized call waiting system for holding a limited number of incoming calls.

Figure: The sleeping barber.

Solution

Our solution uses three semaphores: *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. It is essentially a copy of *customers.* The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

When the barber shows up for work in the morning, he executes the procedure *barber*, causing him to block on the semaphore *customers* because it is initially 0. The barber then goes to sleep, as shown in figure. He stays asleep until the first customer shows up.

```
#define CHAIRS 5 /* # chairs for waiting customers */
typedef int semaphore; /* use your imagination */
semaphore customers = 0; /* # of customers waiting for service */
semaphore barbers = 0; /* # of barbers waiting for customers */
semaphore mutex = 1; /* for mutual exclusion */
int waiting = 0;/* customers are waiting (not being cut) */
void barber(void)
{
white (TRUE) {
down(&customers);/* go to sleep if # of customers is 0 */
down(&mutex);/* acquire access to 'waiting' */
```

```
waiting = waiting - 1;/* decrement count of waiting customers */
up(&barbers);/* one barber is now ready to cut hair */
up(&mutex);/* release 'waiting' */
cut_hair();/* cut hair (outside critical region) */
}}
void customer(void)
{
down(&mutex);/* enter critical region */
if (waiting < CHAIRS) { /* if there are no free chairs, leave */
waiting = waiting + 1;/* increment count of waiting customers */
up(&customers);/* wake up barber if necessary */
up(&mutex);/* release access to 'waiting' */
down(&barbers);/* go to sleep if # of free barbers is 0 */
get_haircut();/* be seated and be serviced */
} else {
up(&mutex);/* shop is full; do not wait */
}}
```

Figure: A solution to the sleeping barber problem.

When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released *mute*. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases *mutex* and leaves without a haircut. If there is an available chair, the customer increments the integer variable, *waiting*. Then he does an up on the semaphore *customers*, thus waking up the barber. At this point, the customer and barber are both awake. When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut. When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, another haircut is given. If not, the barber goes to sleep. As an aside, it is worth pointing out that although the readers and writers and sleeping barber problems do not involve data transfer, they are still belong to the area of IPC because they involve synchronization between multiple processes.

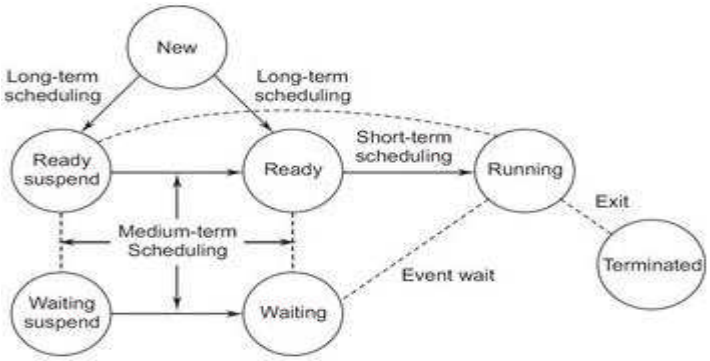
2.2 Process scheduling

Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next jobs on the tape. With timesharing systems, the scheduling algorithm became more complex because there were generally multiple users waiting for service. Some mainframes still combine batch and timesharing service, requiring the scheduler to decide whether a batch job or an interactive user at a terminal should go next. When a computer is multi-programmed, it frequently has multiple processes competing for the CPU at the same time. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler** and the algorithm it uses is called the **scheduling algorithm.** The aim of processor scheduling is to assign processes to be executed by the processor. Scheduling affects the performance of the system, because it determines which process will wait and which will progress.

Types of Scheduling

Long-term Scheduling

Long term scheduling is performed when a new process is created. It is shown in the figure below. If the number of ready processes in the ready queue becomes very high, then there is an overhead on the operating system (i.e., processor) for maintaining long lists, context switching and dispatching increases. Therefore, allow only limited number of processes in to the ready queue. Long-term scheduler determines which programs are admitted into the system for processing. Once when admit a process or job, it becomes process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition; in which case it is added to a queue for the medium-term scheduler scheduling manage queues to minimize queueing delay and to optimize performance.



The long-term scheduler limits the number of processes to allow for processing by taking the decision to add one or more new jobs, based on FCFS (First-Come, first-serve) basis or priority or execution time or Input/Output requirements. Long-term scheduler executes relatively infrequently.

Medium-term Scheduling

Medium-term scheduling is a part of the swapping function. When part of the main memory gets freed, the operating system looks at the list of suspend ready processes, decides which one is to be swapped in (depending on priority, memory and other resources required, etc). This scheduler works in close conjunction with the long-term scheduler. It will perform the swapping-in function among the swapped-out processes. Medium-term scheduler executes somewhat more frequently.

Short-term Scheduling

Short-term scheduler is also called as dispatcher. Short-term scheduler is invoked whenever an event occurs, that may lead to the interruption of the current running process. For example clock interrupts, I/O interrupts, operating system calls, signals, etc. Short-term scheduler executes most frequently. It selects from among the processes that are ready to execute and allocates the CPU to one of them. It must select a new process for the CPU frequently. It must be very fast.

Typically the CPU runs for a while without stopping, and then a system call is made to read from a file or write to a file. When the system call completes, the CPU computes again until it needs more data or has to write more data and so on.

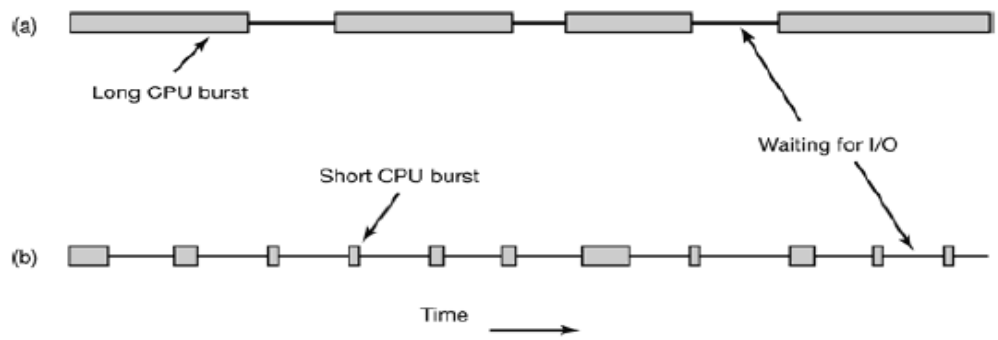


Figure:. (a) A CPU-bound process. (b) An I/O-bound process.

Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits and most of their time in computing, whereas I/O-bound processes have short CPU bursts and thus frequent I/O waits and spend most of their time waiting for I/O.

When to Schedule

A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed.

1. First, when a new process is created, a decision needs to be made whether to run the parent process or the child process. Since both processes are in ready state, it is a normal scheduling decision and it can go either way, that is, the scheduler can legitimately choose to run either the parent or the child next.
2. Second, a scheduling decision must be made when a process exits. That processes can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes. If no process is ready, a system-supplied idle process is normally run.
3. Third, when a process blocks on I/O, on a semaphore, or for some other reason, another process has to be selected to run. Sometimes the reason for blocking may play a role in the choice. For example, if *A* is an important process and it is waiting for *B* to exit its critical region, letting *B* run next will allow it to exit its critical region and thus let *A* continue. The trouble, however, is that the scheduler generally does not have the necessary information to take this dependency into account.
4. Fourth, when an I/O interrupt occurs, a scheduling decision may be made. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide if the newly ready process should be run, if the process that was running at the

time of the interrupt should continue running, or if some third process should run. If a hardware clock provides periodic interrupts at 50 Hz, 60 Hz, or some other frequency, a scheduling decision can be made at each clock interrupt or at every *k* -th clock interrupt.

Scheduling Criteria

Scheduling criteria is also called as scheduling methodology. Different CPU scheduling algorithms have different properties. The criteria used for comparing these algorithms include the following:

CPU Utilization: Keep the CPU as busy as possible. It ranges from 0 to 100%. In practice, it ranges from 40 to 90%. CPU utilization should be maximum.

Throughput: Throughput is the rate at which processes are completed per unit of time. Throughput should be maximum.

Turnaround time: This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion. It should be minimum

Waiting time: Waiting time is the sum of the time periods spent in waiting in the ready queue. It should be as minimum as it can be.

Response time: Response time is the time it takes to start responding from submission time. It is calculated as the amount of time it takes from when a request was submitted until the first response is produced. It should be as minimum it can be.

Fairness: Each process should have a fair share of CPU.

2.4.1 Preemptive Vs. Non Preemptive Scheduling

Non-preemptive Scheduling: In non-preemptive mode, once if a process enters into running state, it continues to execute until it terminates or blocks itself to wait for Input/output or by requesting some operating system service. In non-preemptive scheduling, a running task is executed till completion. It cannot be interrupted. In effect, no scheduling decisions are made during clock interrupts. After clock interrupt processing has been completed, the process that was running before the interrupt is always resumed

Preemptive Scheduling: **Preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, non-preemptive scheduling is the only option. In preemptive mode, currently running process may be interrupted and moved to the ready state by the operating system. 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. It is desirable to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time and response time.

Scheduling Algorithms

Scheduling algorithms or scheduling policies are mainly used for short-term scheduling. The main objective of short-term scheduling is to allocate processor time in such a way as to optimize one or more aspects of system behavior. For these scheduling algorithms assume only a single processor is present. Scheduling algorithms decide which of the processes in the ready queue is to be allocated to the CPU is basis on the type of scheduling policy and whether that policy is either preemptive or non-preemptive. For scheduling arrival time and service time are also will play a role.

First-come First-served Scheduling (FCFS)

The first come first serve (FCFS) is the simplest scheduling algorithm. In this the process that requests the CPU first, is allocated the CPU first. The implementation of the FCFS policy is managed by FIFO queue. The FCFS algorithm is non preemptive (i.e. when the CPU has been allocated to a process, that process keeps the CPU until either terminating or requesting I/O). The FCFS algorithm is troublesome for time-sharing systems, as each process do not get to share of the CPU at the regular intervals due to its non preemptive nature

How FCFS process scheduling works

Suppose there are three processes in a queue: P1, P2 and P3. P1 is placed in the processing register with a waiting time of zero seconds and 10 seconds for complete processing. The next process, P2, must wait 10 seconds and is placed in the processing cycle until P1 is processed. Assuming that P2 will take 15 seconds to complete, the final process, P3, must wait 25 seconds to be processed. FCFS may not be the fastest process scheduling algorithm, as it does not check for priorities associated with processes. These priorities may depend on the processes' individual execution times.

Advantages

1. Easy to understand & operate.
2. FIFO method is useful when transactions are not voluminous.
3. FIFO method is useful when prices of process are falling.
4. FIFO Scheduling is suitable for bulky process having high unit prices.
5. Helps to avoid obsolescence & deterioration.
6. It is simpler scheduling algorithm.
7. It is very easy for implement by using software.

Disadvantages

1. Improper if many lots are process during the period at different prices.
2. If the prices of process are rising rapidly, the current work cost may be understated.
3. overstates profit especially in inflation

Example

For example: Consider the following set of processes — P1, P2 and P3 that arrive at time 0. That is,

Process	CPU Burst Time (ms)
$P1$	24
$P2$	3
$P3$	3

Consider now 2 cases:—

Case 1: When P1, P2 and P3 arrive in order $P1 \rightarrow P2 \rightarrow P3$

In this case, the GANTT chart is as shown below:

∴ Waiting time for $P1$ = 0
Waiting time for $P2$ = 24
Waiting time for $P3$ = 27

∴ Average Waiting Time (AWT) = $\frac{(0+24+27)}{3}$ = **17 ms**

and Average Turnaround Time (ATAT) = (24 + 27 + 30)/3 = **27 ms**

Case 2: When P1, P2 and P3 arrive in order $P2 \rightarrow P3 \rightarrow P1$.

In this case, the GANTT chart is as shown below:

Waiting time for $P1$ = 6
Waiting time for P_2 = 0
Waiting time for P_3 = 3

Average waiting time (AWT) = (6 + 0 + 3)/3 = **3 ms**

Please note that in case-1 above, AWT = 17 ms **whereas in case-2,** AWT = 3 ms *only.* **So, case-2 is much better than case-1. This is because in case-1, the small processes** (P_2 and P_3) **had to wait longer for bigger processes to release the CPU. This effect is known as Convoy Effect.** This effect results in lower CPU utilization. Now, in case-2,

ATAT = (3 + 6 + 30)/3 = **13 ms**

whereas in case-1, ATAT = **27 ms**

So, in case-2, turnaround time also gets better.

2.4.2 Round Robin Scheduling

One of the oldest, simplest, fairest and most widely used algorithms is round robin (RR). In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum. If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list. Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for

interactive users. The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS. In any event, the average waiting time under round robin scheduling is often quite long.

Advantages

1. Round-robin is effective in a general-purpose, times-sharing system or transaction-processing system.
2. Fair treatment for all the processes.
3. Overhead on processor is low.
4. Overhead on processor is low.
5. Good response time for short processes.

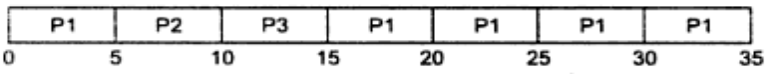
Disadvantages

1. Care must be taken in choosing quantum value.
2. Processing overhead is there in handling clock interrupt.
3. Throughput is low if time quantum is too small.

Example 1

Process	CPU time
P1	25
P2	5
P3	5

Assume that the time slice is of 5ms. Then the Gantt chart is as follows.



P_1 gets the first 5ms, still it requires another 20 ms, it is preempted after the first time slice and the CPU is given to the next process *i.e.* P2. Since P2 just needs 5 ms, it terminates as the time-slice expires. The CPU is then given to the next process P3. once each process has received one time slice, the CPU is then returned to P1 for an additional time-slice.

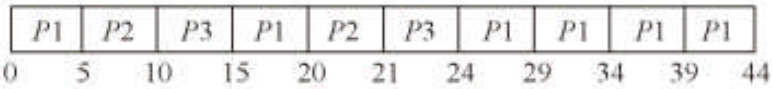
Waiting time for P1 = 0 + 10 = 10
" P2 = 5
" P3 = 10
Average waiting time = 10 + 5 + 10/3 = 25/3 = 8.33ms.
Turn around time for P1 = 35 – 0 = 35
" P2 = 10 – 0 = 10
" P3 = 15 – 0 = 15
Average turn around time = 60/3 = 20ms.

Example 2. Consider another set of three processes as *P*1, *P*2 and *P*3 with their CPU burst times (in ms) as given below—

Process	CPU burst-time (ms)
*P*1	30
*P*2	6
*P*3	8

Assume that time quantum, q = 5ms. Find average waiting time (AWT), Average Turnaround Time (TAT) and Average Response Time (ART)?

Ans. Firstly, *P*1 gets a time quantum of 5ms. But *P*1 still needs 25ms for its execution. As q has expired so, CPU switches from *P*1 to *P*2. When time quantum (q) of *P*2 expires, the CPU switches to process, *P*3. When all finish then CPU switches to *P*1 again as it still needs 25 ms. So, we draw a GANTT chart now.



∴ Waiting time for *P*1 = 0 + (15 – 5) + (24 – 4) = 14
Waiting time for *P*2 = 5 + (20 – 10) = 15
Waiting time for *P*3 = 10 + (21 – 5) = 16

$$\therefore \text{Average Waiting Time (AWT)} = \frac{(14+15+16)}{3} = \frac{45}{3} = 15$$

Also, Turn around time for *P*1 = 44
Turn around time for *P*2 = 21
Turn around time for *P*3 = 24

$$\therefore \text{Average TAT} = \frac{44+21+24}{3} = \frac{89}{3} = \mathbf{29.66\ ms}$$

We can also compute the response time for all these processes.
Response time for *P*1 = 0 ms
Response time for *P*2 = 5 ms
Response time for *P*3 = 10 ms
∴ Average response time = (0 + 5 + 10) = **5 ms**

Example

Process	Arrival Time	Burst Time
P1	0	4
P2	1	5
P3	2	2
P4	3	1

P5	4	6
P6	6	3

Solution

Process	Arrival Time	Burst Time	R.T	C.T	TAT	WT
P1	0	4	4-2=2,2-2=0	8	8-0=8	8-4=4
P2	1	5	5-2=3,3-2=1,1	18	18-1=17	17-5=12
P3	2	2	2-2=0	6	6-2=4	4-2=2
P4	3	1	1	9	9-3=6	6-1=5
P5	4	6	6-2=4,4-2=2,2	21	21-4=17	17-6=11
P6	6	3	3-2=1,1	19	19-6=13	13-3=10

Let time quantum=2

At first process P1 arrives put P1 in the ready queue and Gantt chart. After 2 second P2 and P3 process arrives and the time slot require for P1 is kept after P3. At arrival time 4 two process arrives so it is kept after P1. At arrival time 6 P6 arrives but P2 process need slot so kept after P5 before keeping P6 in the ready queue. P5 arrives then P6 and the Process P2 is kept after P5 and after P2 P6 is kept and then P5.

Ready queue

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

Gantt chart

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

0 2 4 6 8 9 11 13 14 17 18 19 21

Average turnaround time (ATAT) = (8+17+4+6+17+13)/6 = 10.833

Average Waiting time (AWT) = (4+12+2+5+11+10)/6 = 7.333

2.4.3 Priority Scheduling

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm. An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa. Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process. A Priority (an integer) is associated with each process. The CPU is allocated to the process with the highest priority. Generally smallest integer is considered as the highest priority. Process with highest priority is to be executed first and so on. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

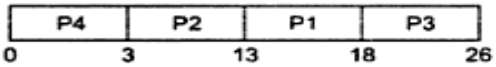
Priority scheduling can be either preemptive or non preemptive

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

Process	CPU time	Priority
P_1	5	3
P_2	10	2
P_3	8	4
P_4	3	1

Assumes that all the processes have arrived at time "0"

Gantt Chart:



Waiting time for $P_1 = 13$
" $P_2 = 3$
" $P_3 = 18$
" $P_4 = 0$
Average waiting time $= 13 + 3 + 18 + 0/4 = 8.5$ms.
Turn around time for $P_1 = 18$
" $P_2 = 13$
" $P_3 = 26$
" $P_4 = 3$
Average turn around time $= 18 + 13 + 26 + 3/4 = 15$ms.

(*a*) **Preemptive (Priority-based) algorithm example**

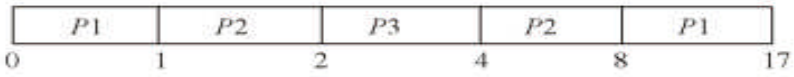
Consider a set of three processes *P*1, *P*2 and *P*3 with their priorities and arrival times as given below—

Process	Burst time	Priority	Arrival time
*P*1	10	3	0
*P*2	5	2	1
*P*3	2	1	2

Where 1 – highest priority.

3 – least priority.

Now, we draw it's GANTT chart—



$\therefore$ Average Waiting Time (AWT) $= \dfrac{0+(8-1)+1+(4-2)+2}{3}$

$= \dfrac{7+1+2+2}{3} = \dfrac{12}{3} = \mathbf{4ms}$

So, here preemption is based on the priority when *P*1 executes, *P*2 arrives with priority = 2 which is higher than priority of 3 of *P*1 and thus, *P*1 is preempted. This goes on.

Problems with preemptive priority scheduling

A major problem with a priority-based scheduling is indefinite blocking of a low priority process by a high priority process. This is called as **starvation. In general, the completion of a process within finite time cannot be guaranteed with this scheduling algorithm.**

A solution to this problem is **aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.** Thus, the older processes attain high priority and are ensured of completion in a finite time.

(*b*) **Non-preemptive (Priority-based) algorithm example**

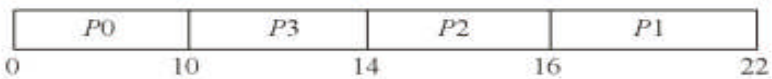
Herein, a higher priority job cannot preempt a low priority job as there is no preemption involved here. Consider a set of four processes – *P*0, *P*1, *P*2 and *P*3. There arrival times and burst times are given next—

Process	AT (ms)	Next Burst (ms)	Priority
*P*0	0	10	5
*P*1	1	6	4
*P*2	3	2	2
*P*3	5	4	0

Where 0 – highest priority

3 – least priority

$\therefore$ It's GANTT chart is—



Now, for each of these processes, we compute the finish time and waiting times from above.

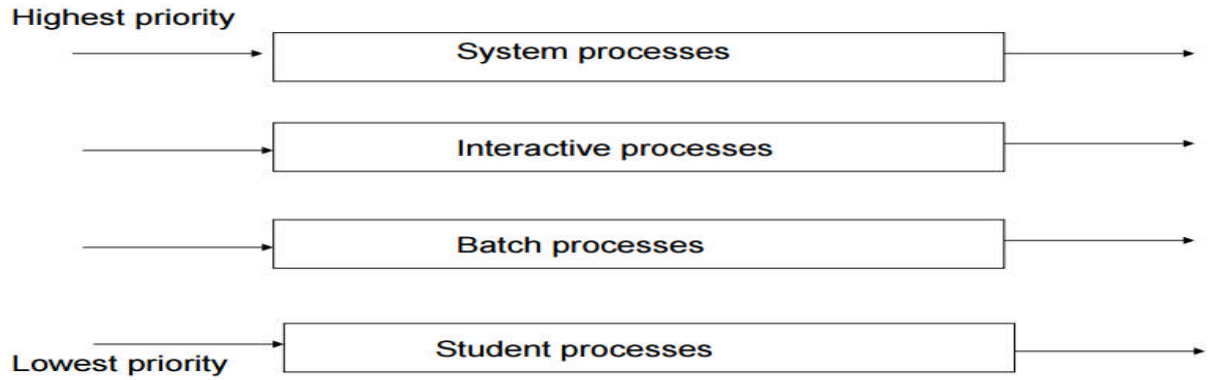
Process	A.T. (T_0)	Next burst (Δt)	F.T. (T_1)	TAT = $T_1 - T_0$	WT = TAT − Δt
*P*0	0	10	10	10	00
*P*1	1	6	22	21	15
*P*2	3	2	16	13	11
*P*3	5	4	14	09	05

$\therefore$ Average TAT = $\frac{(10+21+13+9)}{4} = \frac{53}{4} = \mathbf{13.25\,ms}$

& Average WT = $\frac{(0+15+11+5)}{4} = \frac{31}{4} = \mathbf{7.75\,ms}$

Note here that the average TAT and average WT obtained with this algorithm are much worse than previous algorithms. But, these parameters are not very significant here as our main focus is to give higher priorities to critical processes.

2.4.4 Multilevel queue



Multi level queue scheduling was created for situation in which processes are easily classified into different groups.

Multilevel queue scheduling has the following characteristics:

- Processes are divided into different queue based on their type. Process are permanently assigned to one queue, generally based on some property of process i.e. system process, interactive, batch system, end user process, memory size, process priority and process type.
- Each queue has its own scheduling algorithm. For example interactive process may use round robin scheduling method, while batch job use the FCFS method.

In addition, there must be scheduling among the queue and is generally implemented as fixed priority preemptive scheduling. Foreground process may have higher priority over the

background process. No process in the batch queue could run unless the queue for system processes and interactive processes were all empty. If an interactive process enters the ready queue while a batch process was running, the batch would be preempted

2.4.5 Shortest job first

Shortest job first is a scheduling algorithm in which the process with the smallest execution time is selected for execution next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal. The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available. Shortest-job-first scheduling is also called as shortest process next (SPN). The process with the shortest expected processing time is selected for execution, among the available processes in the ready queue. It gives the minimum average time for a given set of processes. It cannot be implemented at the level of short term CPU scheduling. There is no way of knowing the shortest CPU burst. SJF can be preemptive or non-preemptive. A premptive SJF algorithm will preempt the currently executing process if the next CPU burst of newly arrived process may be shorter than what is left to the currently executing process. A Non-premptive SJF algorithm will allow the currently running process to finish. Preemptive SJF Scheduling is sometimes called Shortest Remaining Time First algorithm.

Advantages

1. It gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
2. Throughput is high.

Disadvantages

1. Elapsed time (i.e., execution-completed-time) must be recorded, it results an additional overhead on the processor.
2. Starvation may be possible for the longer processes.

Example

Non-Preemptive Shortest Job Next

Process	Arrival Time	Burst Time
P1	**0**	**7**
P2	**1**	**5**
P3	**2**	**3**
P4	**3**	**1**
P5	**4**	**2**
P6	**5**	**1**

Solution

The shortest burst time is of Process P6 and P4 but P4 arrives before P6 so P4 is scheduled before P6. So the scheduled processes are P4->P6->P5->P3->P2->P1

The Gantt chart

P4	P6	P5	P3	P2	P1
0	1	2	4	7	12	19

Average waiting time (A.W.T) = 0+1+2+4+7+12/6= 28/6=4.6 ms

Shortest Remaining Time Next

A preemptive version of shortest job first is **shortest remaining time next**. With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short jobs to get good service.

Preemptive Shortest Job Next

Process	Arrival Time	Burst Time
P1	0	7
P2	1	5
P3	2	3
P4	3	1
P5	4	2
P6	5	1

Solution

Process	Arrival Time	Burst Time	Remaining Time
P1	0	7	7-1=6
P2	1	5	5-1=4
P3	2	3	3-1=2, 2-1=1,1-1=0
P4	3	1	0
P5	4	2	2
P6	5	1	1

The arrival time of process 1 is 0 so P1 is scheduled at first. At first process P1 is arrived then after 1 sec another process is P2 arrives with the burst time 5 and process P1 have the remaining burst time 6, whose burst time is less than P1 so it is scheduled after P1. But for process P2 the remaining burst time will be 4. At arrival time 2 Process3 arrives whose burst time is 3 which is less than among process so it is scheduled. At arrival time 3 process P4 arrives whose burst time is 1 which have smaller burst time among the process P1, P2, P3, P4. At arrival time 4 process P5 arrives with burst time 2 but P3 also have the remaining burst time 2 as P3 arrives before P5 with same burst time P3 is scheduled before P5. One cycle is completed as the P6 arrives in the arrival time 5. So after completing one cycle preemptive shortest job next is scheduled according to the non-preemptive scheduling. The remaining processes are P1, P2, P5, and P6 with burst time 6, 4, 2, 1. The shortest burst time is 1 of process P6 is scheduled first then P5, P2, P1 is scheduled.

Gantt chart

P1	P2	P3	P4	P3	P3	P6	P5	P2	P1
0	1	2	3	4	5	6	7	9	13

19

Process	Arrival Time	Burst Time	Remaining Time	Completion Time	TAT	WT
P1	0	7	7-1=6	19	19-0=19	19-7=12

P2	1	5	5-1=4	13	13-1=12	12-5=7
P3	2	3	3-1=2, 2-1=1,1-1=0	6	6-2=4	4-3=1
P4	3	1	0	4	4-3=1	1-1=0
P5	4	2	2	9	9-4=5	5-2=3
P6	5	1	1	7	7-5=2	2-1=1

Note: Completion time is the last time to complete the process.

Turn Around time is completion time (C.T)-arrival time (A.T).

Waiting time is (TAT)-(B.T)

Average waiting time=Summation of all Process waiting tim\e/number of process

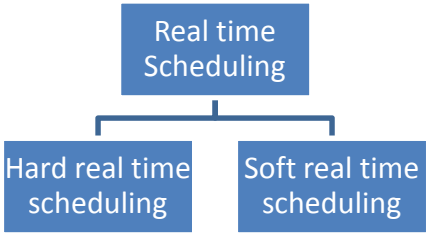
= (12+7+1+0+3+1)/5= 24/5= 4.8 ms

Assignment

Process	Arrival time	Burst Time
P1	5	5
P2	4	6
P3	3	7
P4	1	9
P5	2	2
P6	6	3

2.2.5 Real time Scheduling

Real time system is one in which time plays an important role. Real time system should perform their task in the given time. Having the right answer but having it too late is often just as bad as not having it at all. Real time scheduling is the types of process scheduling which should perform the task and event in the given or real time. Real time scheduling is divided into soft real time scheduling and hard-time scheduling



Hard time scheduling: Hard real time system must execute a set of concurrent real-time task in such a way that all time-critical tasks meet their specified deadline. Every task needs computational and data source to complete the job. The scheduling problem is concerned with the allocation of resource to satisfy timing constraints

Soft time scheduling: Soft real-time system must execute a concurrent real time task in such a way that the task missing the deadline is tolerable.

2.2.6 Two-Level Scheduling

Normally all the runnable processes are in the main memory. If insufficient main memory is available, some of the runnable processes have to keep in the disk in whole or part. This situation has major implication for scheduling, since the process switching time to bring in and run a process from disk should be properly managed. The process of swapping the process between main memory and disk is done by using Two-level scheduler.

In two-level scheduler there are two level of scheduling.

Scheduler 1: Handles running processes that are in the main memory by using standard scheduling algorithm. This is done by low level-scheduler

Scheduler 2: Periodically swap the process into and out from memory to disk and vice-versa. This is done by high-level scheduler.

Chapter 3 Input / Output (3 hrs)

The process of giving input to computer and giving output from computer is called input/ output. The operating system is mainly responsible for input output operating interrupt and error handling is important terms related to input/outputs. So, operating system is responsible to handle interrupt and error. It should also provide an interface between the device and rest of system.

3.1 Principles of I/O Hardware

Different people look at I/O hardware in different ways. Electrical engineer look at in term of chips, wires, power supplies and all other physical components that make up the hardware programmers look at interface presented to the software the commands the hardware accepts, the functions it carries out and the error that can be reported back.

I/O devices

I/O devices are divided into two categories:-

1. **Block devices**:- A block devices is one that store information in fixed-sized blocks, each one, with its own address common blocked size ranges from 512 bytes to 32768 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. In other word, at any instant, the program can read or write any of the blocks. The common examples of block device are disk. A disk is block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for another block to rotate the head.

2. **Character devices**:- A character device is one that delivers or accepts a stream of characters, without regards to any blocks structure. It is not accessible and does not have any such operation. The examples of character devices are printers, paper tapes, network interface card, mice and most other devices that are not disk like can be seen as.

Device controller

I/O units typically consist of mechanical part and the electronic part. The electronic part is also called the device controller or adapter. On pc, device controller takes the form of printed

circuit card that can be inserted into an expansion slots. The controller card actually has a connected on it, into which a cable leading to the device itself can be plugged many controllers can handle more than one identical devices. The standard for interface between controller and device are ANSI, ICE, IDE, SCSI, ISO etc.

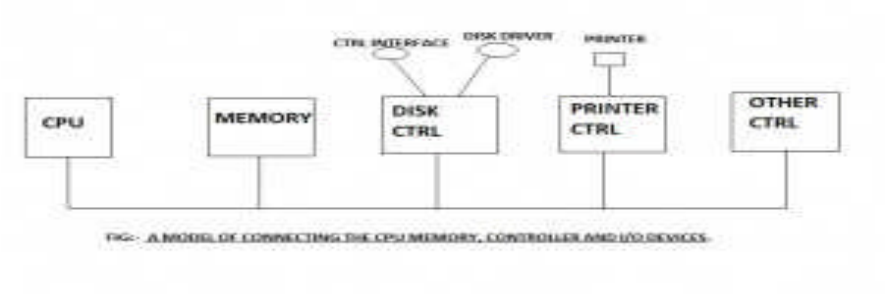
The interface between the controller and device is often a very low level interface. The controller job is to convert the serial bit stream into a block of bytes and perform any error. Correction if necessary the block of bytes is typically first assembled, bit by bit in a buffer inside the controller. After its checksum has been verified and block declared to be error free, it can then be copied to main memory.

Each controller has some registers for communicating with CPU and many devices have data buffer, which the CPU can read and write data. The issues that arise of how the CPU communicates with the controller registers and the device data buffer has two alternatives.

1. **I/o mapped I/o:-** In this approach, each control register is assign an i/o port number and 8 bits or 16 bits integer. The scheme uses I/O instruction for I/O such as in OUT PORT, REG (CPU register).
2. **Memory mapped I/O:-** In this approach, all the control register are mapped into the memory space. Each control register is assigned a unique memory address to which no memory is assigned. Usually, the assigned address is at the top of the address space. In this approach memory instruction like mov, stor, load are used.

The general connection of device controller:-

Device controller is interface between I/O device and computer. The computer here indicates CPU and memory. CPU memory controllers all are connected to system bus. The controller takes information and gives information to the operating system from the system memory. Large mainframe computer uses I/O channel for I/O processing where I/O channels are processor.



The general connection of device controller

Direct Memory Access (DMA)

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time but doing so wastes the CPU's time, so a different scheme, called **DMA (Direct Memory Access)** is often used. The operating system can only use DMA if the hardware has a DMA controller, which most systems do. Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the parentboard) for regulating transfers to multiple devices, often concurrently.

No matter where it is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Figure. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

How DMA works?

To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred. Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.

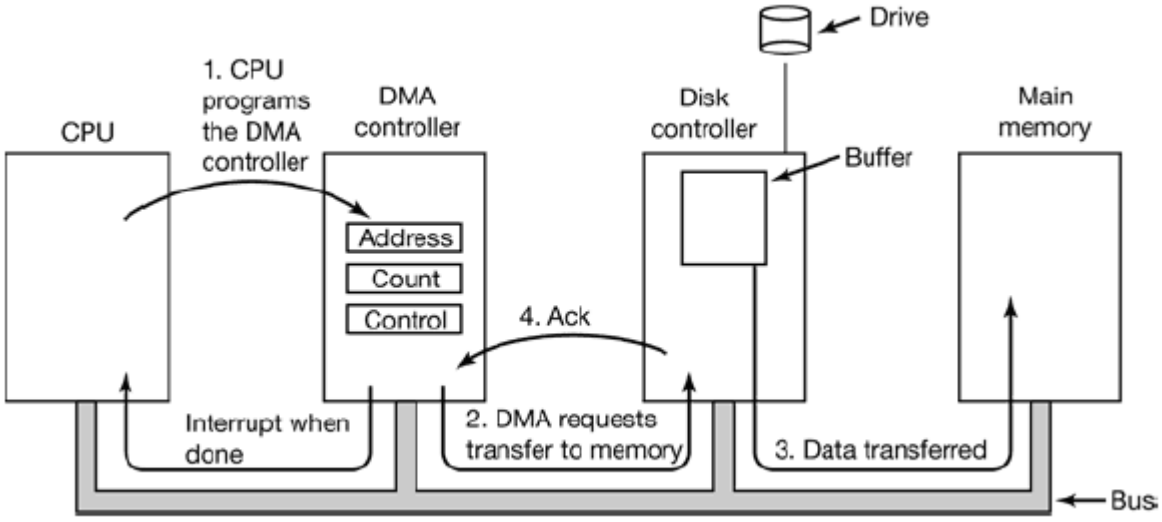


Figure: operation of DMA

When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Figure). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.

The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the disk controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete. When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

DMA controllers vary considerably in their sophistication. The simplest ones handle one transfer at a time, as described above. More complex ones can be programmed to handle multiple transfers at once. Such controllers have multiple sets of registers internally, one for each channel. The CPU starts by loading each set of registers with the relevant parameters for its transfer. Each transfer must use a different device controller. After each word is

transferred (steps 2 through 4) in Figure, the DMA controller decides which device to service next. It may be set up to use a round-robin algorithm, or it may have a priority scheme design to favor some devices over others. Multiple requests to different device controllers may be pending at the same time, provided that there is an unambiguous way to tell the acknowledgements apart. Often a different acknowledgement line on the bus is used for each DMA channel for this reason.

Many buses can operate in two modes: word-at-a-time mode and block mode. Some DMA controllers can also operate in either mode. In the former mode, the operation is as described above: the DMA controller requests for the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly. In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called **burst mode.** It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition. The down side to burst mode is that it can block the CPU and other devices for a substantial period of time if a long burst is being transferred.

In the model we have been discussing, sometimes called **fly-by mode** , the DMA controller tells the device controller to transfer the data directly to main memory. An alternative mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write the word to wherever it is supposed to go. This scheme requires an extra bus cycle per word transferred, but is more flexible in that it can also perform device-to-device copies and even memory-to-memory copies (by first issuing a read to memory and then issuing a write to memory at a different address).

Most DMA controllers use physical memory addresses for their transfers. Using physical addresses requires the operating system to convert the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register. An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead. Then the DMA controller must use the MMU to have the virtual-to-physical translation done. Only in the case that the MMU is part of the memory (possible, but rare) rather than part of the CPU, can virtual addresses be put on the bus.

The disk first reads data into its internal buffer before DMA can start. You may be wondering why the controller does not just store the bytes in main memory as soon as it gets them from the disk. In other words, why does it need an internal buffer? There are two reasons. First, by doing internal buffering, the disk controller can verify the checksum before starting a transfer. If the checksum is incorrect, an error is signaled and no transfer is done. The second reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate, whether the controller is ready for them or not. If the controller tried to write data directly to memory, it would have to go over the system bus for each word transferred. If the bus were busy due to some other device using it (e.g., in burst mode), the controller would have to wait. If the next disk word arrived before the previous one had been stored, the controller would have to store it somewhere. If the bus were very busy, the controller might end up storing quite a few words and having a lot of administration to do as well. When the block is buffered internally, the bus is not needed until the DMA begins, so the design of the controller is much simpler because the DMA transfer to memory is not time critical. (Some older controllers did, in fact, go directly to memory with only a small amount of internal buffering, but when the bus was very busy, a transfer might have had to be terminated with an overrun error.)

Not all computers use DMA. The argument against it is that the main CPU is often far faster than the DMA controller and can do the job much faster (when the limiting factor is not the speed of the I/O device). If there is no other work for it to do, having the (fast) CPU wait for the (slow) DMA controller to finish is pointless. Also, getting rid of the DMA controller and having the CPU do all the work in software saves money, important on lowend (embedded) computers.

Interrupt

An interrupt is a special request signal originated from some device to CPU to achieve the CPU time for some job. Interrupt is identified by some special number and is managed by interrupt controller.

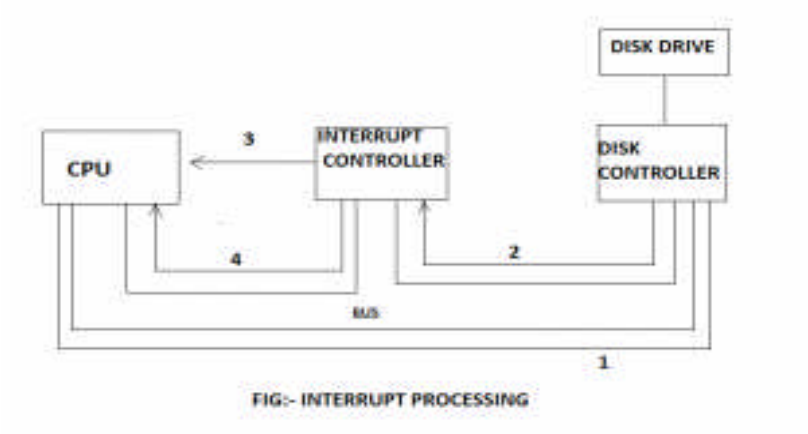


FIG:- INTERRUPT PROCESSING

Input and output can be done in three different ways. In this simplest method, a user program issues the system call, which the kernel then translates into a procedure call to the appropriate driver. The driver then starts the I/O and sits in the tight loop continuously. Polling the device to see if it is done when the I/O has completed, the driver puts the data where they are needed and returns. The operating system then returns control to the caller. This method is called busy waiting and has the disadvantage of typing up the CPU pooling the device until it is finished. The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that print the driver returns. The operating system then blocks the caller if need be and looks for other work to do. When the controller detects the end of transfer, it generates an interrupt to signal completion.

3.2 Principles of I/O Software

The general goals for I/O software are easy to state. The basic idea is to organize the software as a series of layers, while the lower ones and the upper ones concerned with presenting a nice, clean, regular interface to the user.

Layers of I/O software

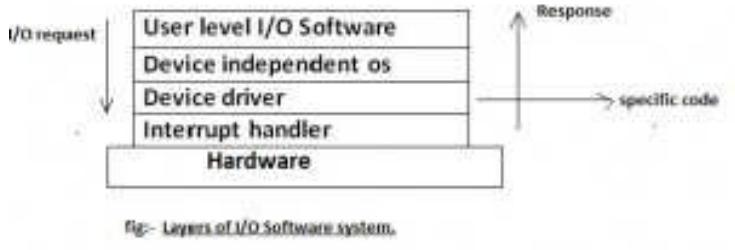


fig:- Layers of I/O Software system.

The layers of I/O software system are described below:-

a. User level I/O software: - These are application level software which use I/O system calls for the input and output for example, in c programming, printf and scanf functions are used for output and input. Not all user level software consists of library procedures. Another important category is the spooling system. It is the way of dealing with delicate I/O devices in the multiprogramming system. In printer, a special process called a daemon and a special directory called a spooling directory are used to manage the output. To print a file, a process that generates the entire file to be printed and puts it in the spooling directory to use the printer's special file, to print the file in directory.

b. Device independent i/o software: - The basic function on device independent i/o software or operating system are:-

- To provide uniform interface to the user level software.
- To perform I/O functions those are common to all devices.
- Buffering.
- Error reporting.
- Allocating and relating dedicated devices.
- Providing a device independent block size.
- Device naming.
- Device protection.

Different disk may have different sizes. It is up to the devices independent software to hide this fact and provide a uniform block size to higher layers.

c. Device driver: - Each and every I/O device attached to computer needs some device specific code for controlling it. This code is called as device driver, written by devices manufacturer and deliver along with the device. Drivers are specific to operating system. A driver in computer science is a software program that expands the capability of computer to control various input and output devices. Many driver programs come as the part of computer's operating system. The software that schedules task, allocates data storage and co-ordinates data transmission between the computer, connect5ed devices and other computers. When connecting peripherals devices, the user often must load a new driver into the computer for each device. All devices require driver program to function.

In computer a driver serves as a translator between the devices and the program that send commands to the devices. When user selects the print command in word processing program, a program sends the generic comment to print, through operating system. The driver interpret commands and translate it, converting it into specialized command that printer can understand. The driver program then access the hardware registers of the devices.

The command issues from the application program is send to device controller and store the information to device controller's register and provide for device driver. In case of printer the information cannot perform by storing information in device driver it to provide error reporting to the caller procedure as well as status reporting of device.

d**. Interrupt handler**: - Many devices have an interrupt handler that notifies user if a device's functionality is interrupted. The driver software may then supplies the user with information on the device and often error co procedure message in dos driver file has .sys extension. In windows driver file has .read.drv.

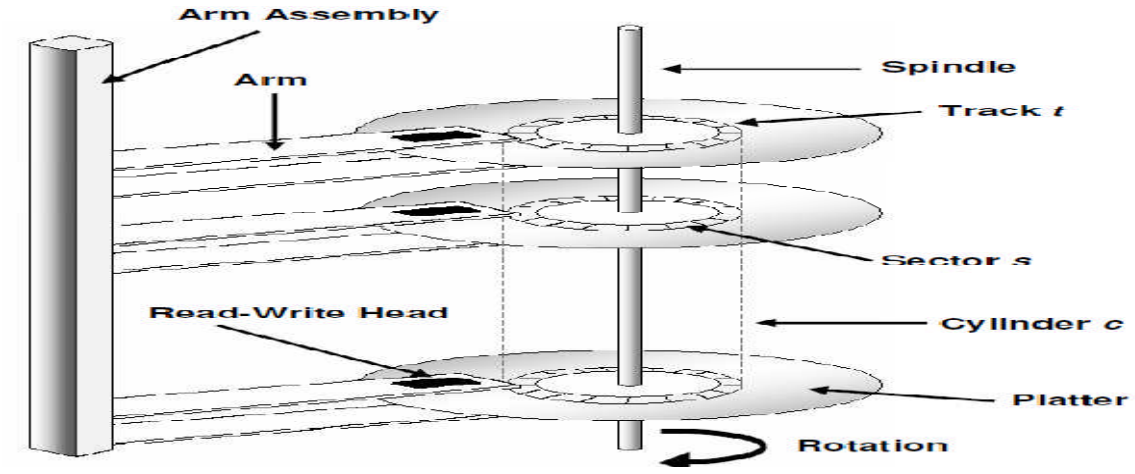
3.3 Disks

A disk drive is a device that reads and/or writes data to a disk. A disk drive is a randomly addressable and rewritable storage device. The most common type of disk drive is a hard drive (or "hard disk drive"), but several other types of disk drives exist as well. Some examples include removable storage devices, floppy drives, and optical drives, which read optical media, such as CDs and DVDs.

While there are multiple types of disk drives, they all work in a similar fashion. Each drive operates by spinning a disk and reading data from it using a small component called a drive head. Hard drives and removable disk drives use a magnetic head, while optical drives use a laser. CD and DVD burners include a high-powered laser that can imprint data onto discs.

A disk is a platter, made of metal or plastic with a magnetiziable coating on it, and in circular shape. It is possible to store information by recording it magnetically on the platters. A conducting coil, called head, which is a relatively small device, facilitates the data recording on and retrieval from the disk. In a disk system, head rotates just above both surfaces of each platter. All heads, being attached to a disk arm, move collectively as a unit. To enable a read and write operation, the platter rotates beneath the stationary head. Data are organized on the platter in tracks, which are in the form of concentric set of rings. The rotation speed increases

as the head moves from the outer to the inner tracks to keep the same data transfer rate. This method is also used in CD-ROM and DVDROM drives.



There are three elements of a disk known as cylinder, track and sector/block. Tracks have subdivisions, called sectors. Data are transferred to and from the disk in blocks, size of which are typically smaller than the capacity of the track. Block-size regions on the disk where data are recorded, are called sectors each having 512 bytes capacity for most disk drives. The request locations are defined with the physical block addresses over these sectors. Adjacent sectors are separated by in track gaps in order to avoid imposing unreasonable precision requirements on the system. A common disk drive has a capacity in the size of gigabytes. While the set of tracks that are at one arm position forms a cylinder, in a disk drive there may be thousands of concentric cylinders.

Working Principle

In a movable-head disk, where there is only one access arm to service all the disk tracks, the time spent by the Read and Write (R/W) head to move from one track to another is called Seek Time. This is differing from a fixed-head disk wherein there is an assigned arm for every track. Seek time can be compared to the movement of a space shuttle from one planetary orbit to another. There are two kinds of seek either inward or outward seek depending on the current location of the R/W head. Moving towards the outer portion of the disk corresponds to an outward seek while moving towards the center of the disk considered an inward seek. Once it reaches the desired track, the disk will rotate to find the block of data to be accessed and position it before the R/W head. The time spent in moving the disk from undesired block to desired block is known as rotational delay or latency time. Rotational

delay positions the tip of the R/W head at the beginning of the sector1 where the data is to be read or written. As soon as the R/W head is positioned at the beginning of the data block, it is electrically switched on in line with the preparation for read or writes function. Once it is switched on, that's the time the read or write functions begin. Finally, the disk will again rotate in order for the data block to get ahead of under read/write. The data is then read from or written to the disk; consequently it is called transfer time.

Disk scheduling Algorithm

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. The access time has two major components. The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order. Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

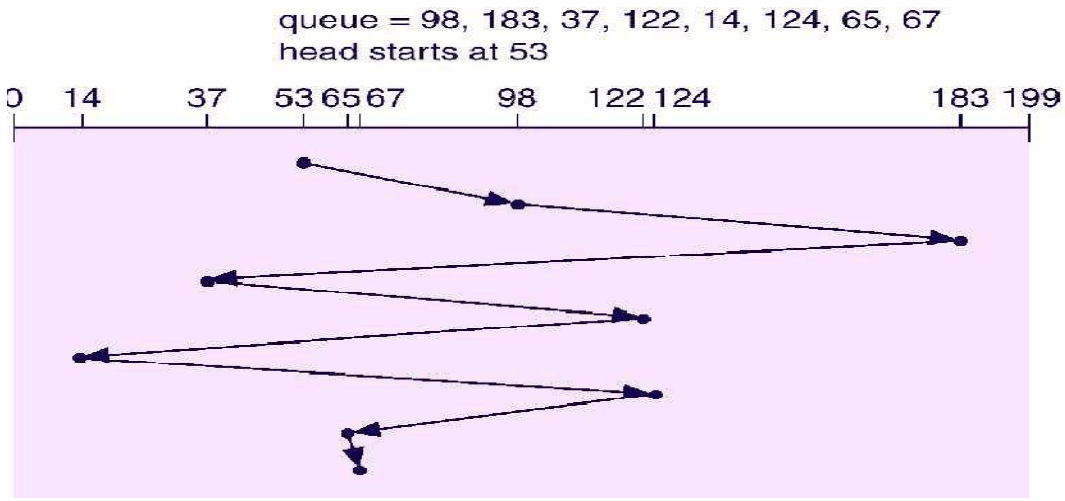
• Whether this operation is input or output
• What the disk address for the transfer is
• What the memory address for the transfer is
• What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.

For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. There are various types of disk scheduling. They are

1. **First-Come First-Served (FCFS)**

   The principle used in FCFS is "the disk controller processes the I/O request in the order in which they arrive, thereby moving backwards and forwards across the surface of the disk to get to the next requested location each time.



Total Head Movement = (53 to 98) + (98 to 183) + (183 to 37) + (37 to 122) + (122 to 14) + (14 to 124) + (124 to 65) + (65 to 67)

= (98-53) + (183-98) + (183-37) + (122-37) +(122-14) + (124-14) + (124-65) + (67-65)

=45+85+146+85+108+110+59+2

=640 cylinders

Advantages:

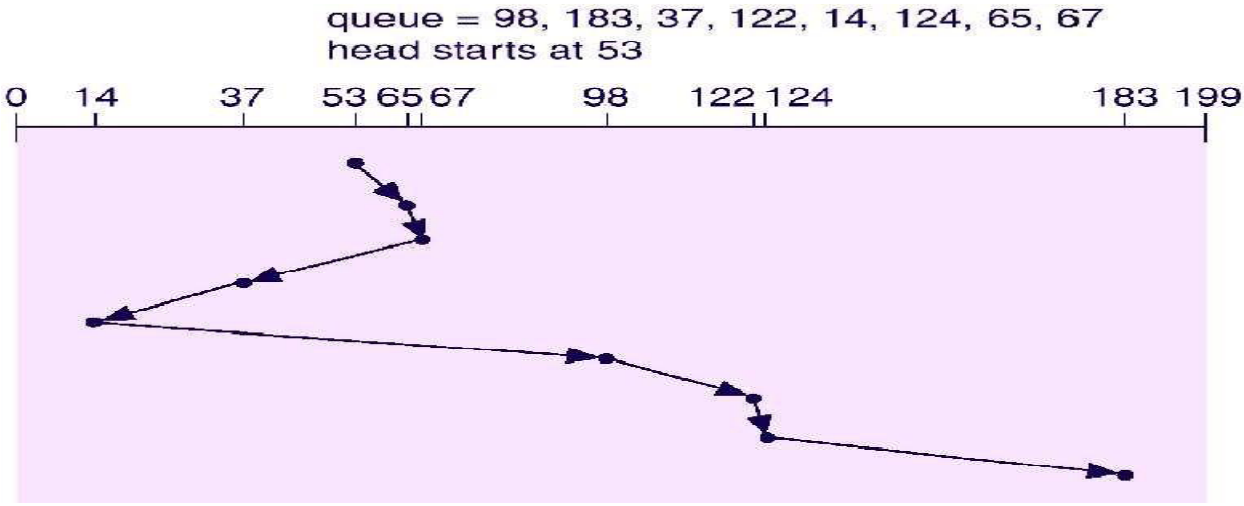
- It is very simple to implement.
- Improved response time as a request in fair amount of time

Disadvantage:

- It involves a lot or random head movement and disk rotation.
- Throughput is not efficient.
- It is used in small system only where I/O efficiency is not very important
- FCFS is acceptable when the load on a disk is light. As the load grows, FCFS tends to saturate the device and the response time becomes longer.

Shortest-Seek-Time-First (SSTF)

This algorithm works on the principle: "When a disk operation finishes, choose the request that is closest to the current head position of choose the request that has minimum seek time from the current head position



Here in the above figure 53 is the current head position and the shortest difference between 53 and 65 is least then other so after 53 the head move to 65 and after that to 67. By calculating the difference between 67 with 98 and 67 with 37 the least difference is 67 with 37 so head move backward to 37. In this way SSTF head movement is calculated

Total Head Movement = 236 cylinders

Advantages:

- It minimizes latency
- Better throughput than FIFO

Disadvantage:

- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- SSTF service request for those tracks which are highly localized. So the innermost and outermost tracks receive poor service as compared to the mid range tracks.

SCAN

The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues. Sometimes it is called the elevator algorithm. The principle used:"The head constantly moves from the most inner cylinder to the outer cylinder and then it changes its direction back towards the center. As the head moves, if there is a request for the current position then it is satisfied."

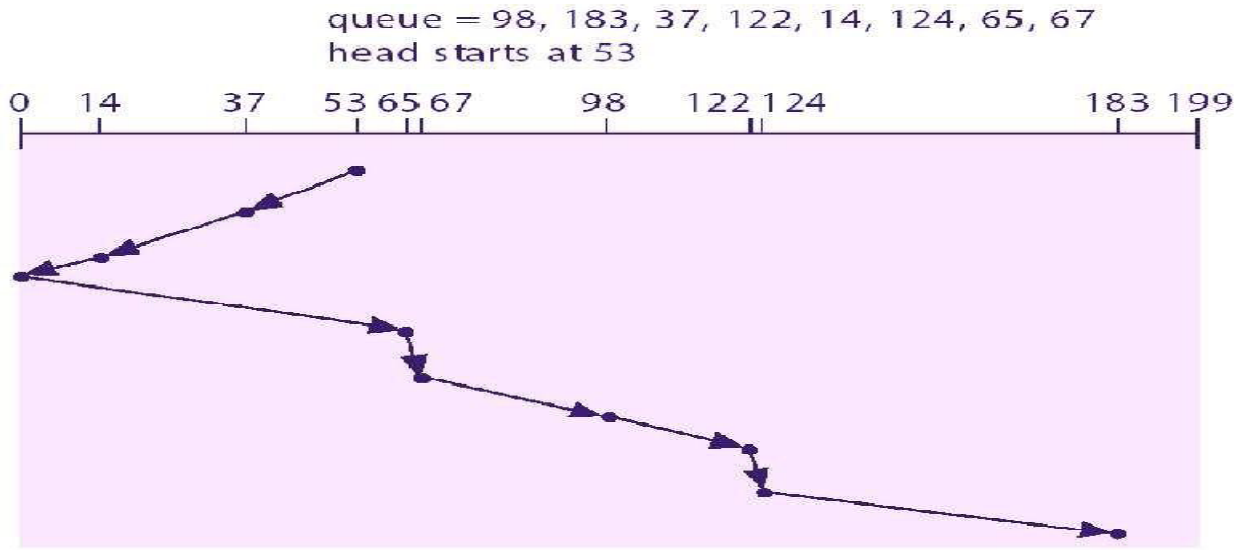
Advantages:

- The throughput id better than FIFO

- It has been the basic of the most disk scheduling strategies
- It eliminates the discrimination inherent in SSTF
- No starvation problem

Disadvantage

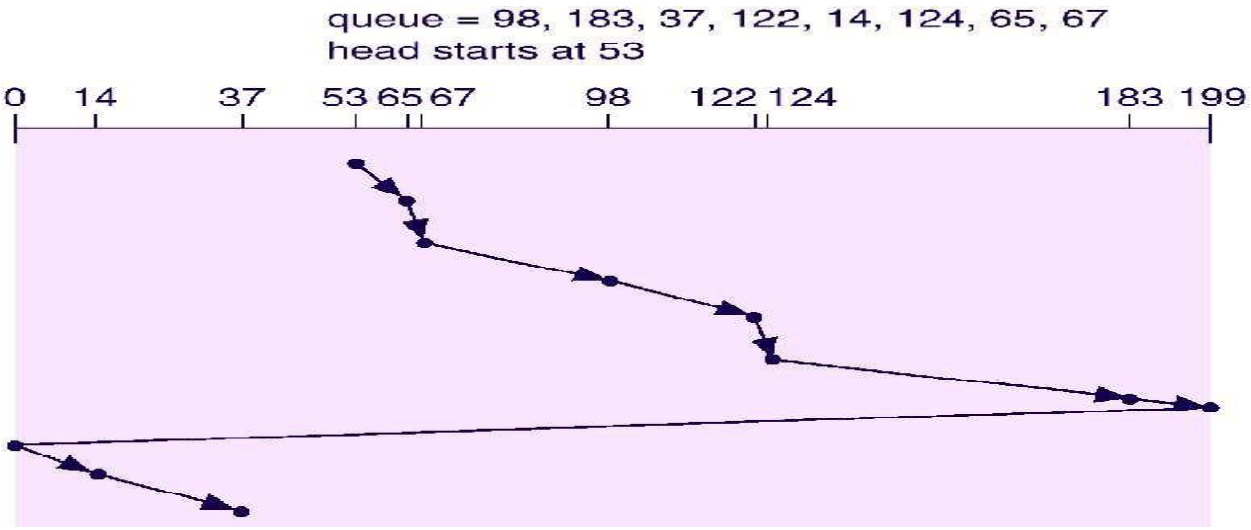
- Because of the continuous scanning of disk from end to end, the outer tracks are visited less often than the mid range tracks.
- Also as the disk arm keeps scanning between two extremes, this may result in wear ad tear of the disk assembly.
- Certain request arriving ahead of arm position would get immediate service but some other request that arrive behind the arm position will have to wait for the arm to return back. So this algorithm is not fair.



Total Head Movement = 208 cylinders

C-SCAN

C-SCAN stands for circular scan and so called bacuase this algorithm treat the cylinder as a circular list. C-scan is similar to scan but I/O request are only satisfied when the drive head is moving in one direction across the surface of the disk. Principle:" The head sweeps from the innermost cylinder to the outermost cylinder satisfying the waiting request in order of their location. When it reaches the outermost cylinder, it sweeps back to the innermost cylinder without satisfying any request and then start again."



Advantage:

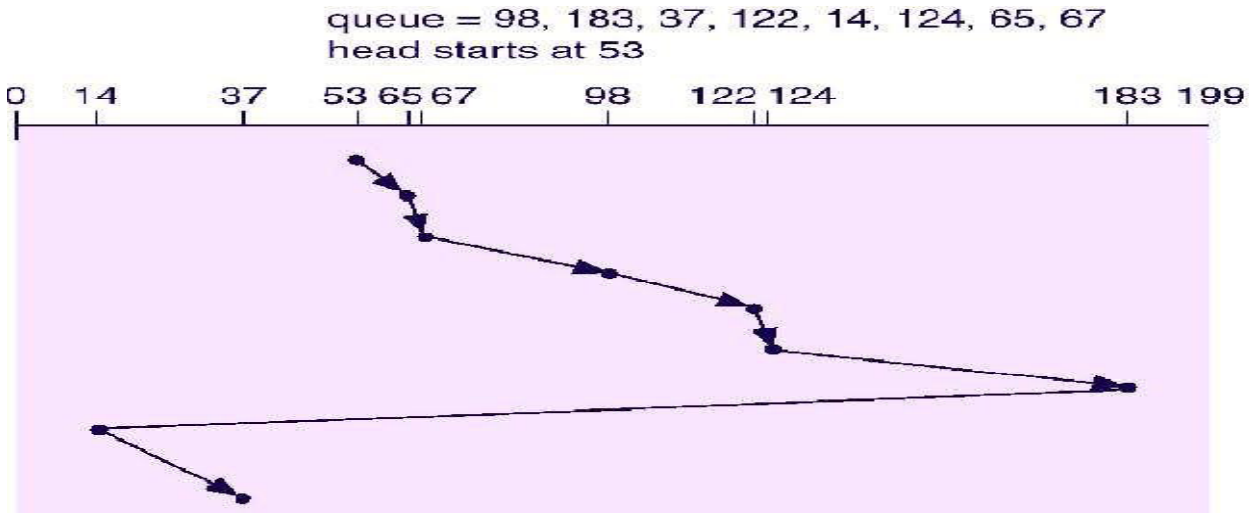
- It is more fair as compared to SCAN
- It provides more uniform waiting time

Disadvantage

- The time taken for the back swing has been ignored
- The average head movement in this algorithm is more as compared to SCAN
- This method increase the total seek time because of the ling seed from the edge back to the hub

C-LOOK

C-LOOK stands for circular LOOK algorithm. It is variation of LOOK where request are satisfied only when the head moves outwards, as in C-SCAN. Thus, no request is satisfied when the head moves inward after determining that no request are there beyond the current point.



3.4 Clocks

Clocks are also called **timers which** are essential to the operation of any multi-programmed system for a variety of reasons. They maintain the time of day and prevent one process from monopolizing the CPU, among other things. The clock software can take the form of a device driver, even though a clock is neither a block device, like a disk, nor a character device, like a mouse.

Clock Hardware

There are two types of clocks are commonly used in computers. The simpler clocks are tied to the 110- or 220-volt power line and cause an interrupt on every voltage cycle, at 50 or 60 Hz. These clocks used to dominate, but are rare nowadays.

The other kind of clock is programmable clock which is built out of three components: a crystal oscillator, a counter, and a holding register. When a piece of quartz crystal is properly cut and mounted under tension, it can be made to generate a periodic signal of very high accuracy, typically in the range of several hundred megahertz, depending on the crystal chosen. Using electronics, this base signal can be multiplied by a small integer to get frequencies up to 1000 MHz or even more. At least one such circuit is usually found in any computer, providing a synchronizing signal to the computer's various circuits. This signal is fed into the counter to make it count down to zero. When the counter gets to zero, it causes a CPU interrupt. Programmable clocks typically have several modes of operation. In **one-shot mode**, when the clock is started, it copies the value of the holding register into the counter and then decrements the counter at each pulse from the crystal. When the counter gets to

zero, it causes an interrupt and stops until it is explicitly started again by the software. In **square wave mode**, after getting to zero and causing the interrupt, the holding register is automatically copied into the counter, and the whole process is repeated again indefinitely. These periodic interrupts are called **clock ticks**.

Clock Software

All the clock hardware generates interrupts at known intervals. Everything else involving time must be done by the software, the clock driver. The exact duties of the clock driver vary among operating systems, but usually include most of the following:

1. Maintaining the time of day.
2. Preventing processes from running longer than they are allowed to
3. Accounting for CPU usage.
4. Handling the alarm system call made by user processes.
5. Providing watchdog timers for parts of the system itself.
6. Doing profiling, monitoring, and statistics gathering.

1. **Maintaining the time of day:** The first clock function, maintaining the time of day (also called the **real time**) is not difficult. It just requires incrementing a counter at each clock tick. The only thing to watch out for is the number of bits in the time-of-day counter. With a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years. Clearly the system cannot store the real time. There are three ways for maintain the time of day.

The first way is to use a 64-bit counter, although doing so makes maintaining the counter more expensive since it has to be done many times a second.

The second way is to maintain the time of day in seconds, rather than in ticks, using a subsidiary counter to count ticks until a whole second has been accumulated. Because 2^{32} seconds is more than 136 years, this method will work until the twenty-second century.

The third approach is to count in ticks, but to do that relative to the time the system was booted, rather than relative to a fixed external moment. When the backup clock is read or the user types in the real time, the system boot time is calculated from the current time-of day value and stored in memory in any convenient form. Later, when the time of day is requested, the stored time of day is added to the counter to get the current time of day.

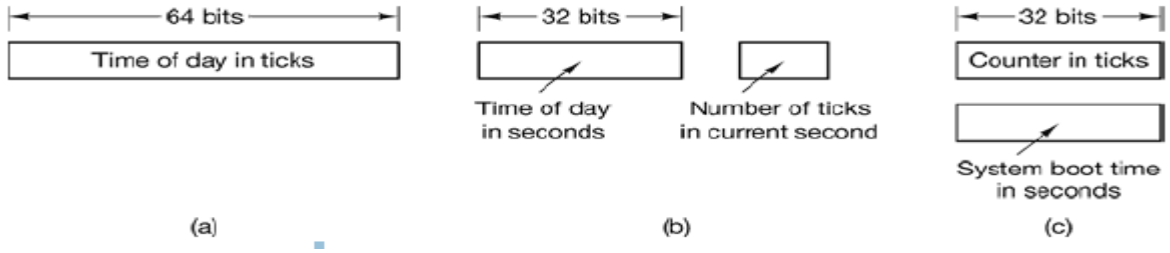


Figure: Three ways to maintain the time of day.

2. **Preventing processes from running longer than they are allowed to:** Whenever a process is started, the scheduler initializes a counter to the value of that process' quantum in clock ticks. At every clock interrupt, the clock driver decrements the quantum counters by 1. When it gets to zero, the clock driver calls the scheduler to set up another process.
3. **Accounting for CPU usage:** The most accurate way to do it is to start a second timer, distinct from the main system timer, whenever a process is started. When that process is stopped, the timer can be read out to tell how long the process has run. To do things right, the second timer should be saved when an interrupt occurs and restored afterward. A less accurate, but much simpler, way to do accounting is to maintain a pointer to the process table entry for the currently running process in a global variable. At every clock tick, a field in the current process' entry is incremented. In this way, every clock tick is "charged" to the process running at the time of the tick. A minor problem with this strategy is that if many interrupts occur during a process' run, it is still charged for a full tick, even though it did not get much work done. Properly accounting for the CPU during interrupts is too expensive and is rarely done.
4. **Handling the alarm system call made by user processes:** In many systems, a process can request that the operating system give it a warning after a certain interval. The warning is usually a signal, interrupt, message, or something similar. One application requiring such warnings is networking, in which a packet not acknowledged within a certain time interval must be retransmitted. If the clock driver had enough clocks, it could set a separate clock for each request. One way is to maintain a table in which the signal time for all pending timers is kept, as well as a variable giving the time of the next one. Whenever the time of day is updated, the driver checks to see if the closest signal has occurred. If it has, the table is searched for the next one to occur. If many signals are expected, it is more efficient to simulate multiple clocks by chaining all the pending clock requests together, sorted on time, in a linked list, as.
5. **Providing watchdog timers for parts of the system itself:** The mechanism used by the clock driver to handle watchdog timers is the same as for user signals. The only

difference is that when a timer goes off, instead of causing a signal, the clock driver calls a procedure supplied by the caller. The procedure is part of the caller's code. The called procedure can do whatever is necessary, even causing an interrupt, although within the kernel interrupts are often inconvenient and signals do not exist. That is why the watchdog mechanism is provided. It is worth nothing that the watchdog mechanism works only when the clock driver and the procedure to be called are in the same address spaces.

6. **Doing profiling, monitoring, and statistics gathering:** Some operating systems provide a mechanism by which a user program can have the system build up a histogram of its program counter, so it can see where it is spending its time. When profiling is a possibility, at every tick the driver checks to see if the current process is being profiled, and if so, computes the bin number (a range of addresses) corresponding to the current program counter, it then increments that bin by one. This mechanism can also be used to profile the system itself.

Generally, there are two ways to manage I/O: interrupts and polling. Interrupts have low latency, that is, they happen immediately after the event itself with little or no delay. On the other hand, with modern CPUs, interrupts have a substantial overhead due to the need for context switching and their influence on the pipeline, TLB and cache.

The alternative to interrupts is to have the application poll for the event expected itself. Doing this avoids interrupts, but there may be substantial latency because an event may happen directly after a poll, in which case it waits almost a whole polling interval. On the average, the latency is half the polling interval.

Soft timers avoid interrupts. Instead, whenever the kernel is running for some other reason, just before it returns to user mode it checks the real time clock to see if a soft timer has expired. If the timer has expired, the scheduled event (e.g., packet transmission or checking for an incoming packet) is performed, with no need to switch into kernel mode since the system is already there. After the work has been performed, the soft timer is reset to go off again. All that has to be done is copy the current clock value to the timer and add the timeout interval to it. Soft timers stand or fall with the rate at which kernel entries are made for other reasons. These reasons include

1. System calls.
2. TLB misses.
3. Page faults.

4. I/O interrupts.
5. The CPU going idle.

3.5 Terminals

Every general-purpose computer has at least one keyboard and one display (monitor or flat screen) used to communicate with it. Although the keyboard and display on a personal computer are technically separate devices, they work closely together. On mainframes, there are frequently many remote users, each with a device containing a keyboard and an attached display. These devices have historically been called **terminals**. Terminals come in many forms. There are three of the type's most commonly encountered terminals:
1. Standalone terminals with RS-232 serial interfaces for use on mainframes.
2. Personal computer displays with graphical user interfaces.
3. Network terminals.

1. Standalone terminals with RS-232 serial interfaces for use on mainframes:

RS-232 terminals are hardware devices containing both a keyboard and a display and which communicate using a serial interface, one bit at a time. These terminals use a 9-pin or 25-pin connector, of which one pin is used for transmitting data, one pin is for receiving data, and one pin is ground. The other pins are for various control functions, most of which are not used. Lines in which characters are sent one bit at a time (as opposed to 8 bits in parallel the way printers are interfaced to PCs) are called **serial lines**. To send a character over a serial line to an RS-232 terminal or modem, the computer must transmit it 1 bit at a time, prefixed by a start bit, and followed by 1 or 2 stop bits to delimit the character. A parity bit which provides rudimentary error detection may also be inserted preceding the stop bits, although this is commonly required only for communication with mainframe systems.

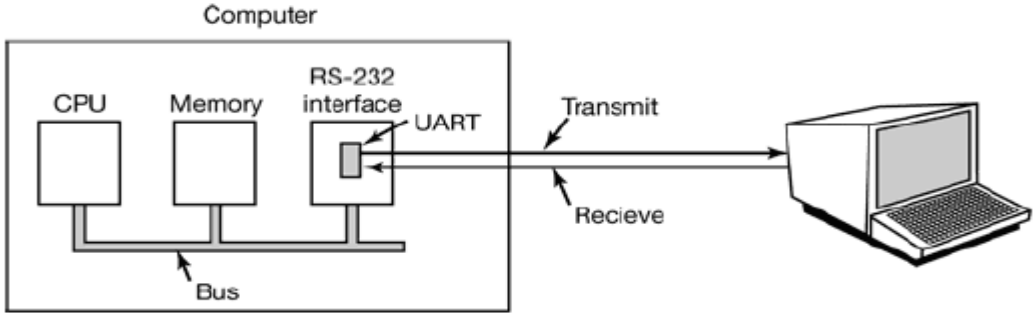


Figure: An RS-232 terminal communicates with a computer over a communication line, one bit at a time.

2. Personal computer displays with graphical user interfaces

PCs can use character-based interfaces. In fact, for years MS-DOS, which is character based, dominated the scene. However nowadays most personal computers use a **GUI (Graphical User Interface).** The acronym GUI is pronounced "gooey." A GUI has four essential elements, denoted by the characters WIMP. These letters stand for Windows, Icons, Menus, and Pointing device, respectively. Windows are rectangular blocks of screen area used to run programs. Icons are little symbols that can be clicked on to cause some action to happen. Menus are lists of actions from which one can be chosen. Finally, a pointing device is a mouse, trackball, or other hardware device used to move a cursor around the screen to select items. The GUI software can be implemented in either user-level code, as is done in UNIX systems, or in the operating system itself, as in the case in Windows.

3. Network terminals

Network terminals are used to connect a remote user to computer over a network, either a local area network or a wide area network. There are two different philosophies of how network terminals should work. In one view, the terminal should have a large amount of computing power and memory in order to run complex protocols to compress the amount of data sent over the network. (A **protocol** is a set of requests and responses that a sender and receiver agree upon in order to communicate over a network or other interface.) In the other view, the terminal should be extremely simple, basically displaying pixels and not doing much thinking in order to make it very cheap.

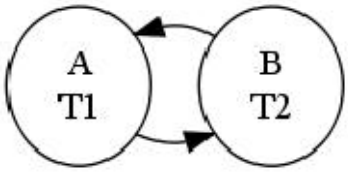
Chapter 4: Deadlocks (4 hrs)

4.1 Deadlock Detection and algorithm

Deadlock is a significant problem that can arise in a community of co-operating or competing processes. Deadlock that can occur among a group of two or more processes whereby each process holds as least one resource while making a request on another. A set of two or more processes are deadlocked if they are blocked (i.e., in the waiting state) each holding a resource and waiting to acquire a resource held by another process in the set.

OR

A process is deadlocked if it is waiting for an event which is never going to happen. A process in a multiprogramming system is said to be in dead lock if it is waiting for a particular event that will never occur.



Example 1: Suppose there are two tape drive each of the process have one one tape and requesting for another tape is deadlock.

4.1 Resources of Deadlock

Finite number of resource is available in the system by different processes. There are generally two type of resource. They are

1. Resuable resource: Resuable resource is one that can be safely used by only one process at a time and is not depleted by that use. Process after finishing their task release the resource which can be reused by another processes. Example I/O device, files, database etc….

2. Consumable resource: Consumable resource is one that can be created and destroyed that is it cannot be reused by another processes. There is no limit on the number of consumable resource of a particular type. Example interrupts, signals etc…

- A process must request a resource before using it, and must release the resource after using it. The number of resources requested may not exceed the total number of resources available in the system. If the system have 4 printers then the request for printer is equal to or less than 4.
- A process may utilize a resource in only the following sequece.
  1. **Request :** If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resources.
  2. **Use :** The process can operate the resource.
  3. **Release : The process releases the resources.**
- The three processes might have put the system in the state shown below by executing as follows :

Process 1	Process 2	Process 3
- - - - -	- - - - -	- - - - -
- - - - -	- - - - -	- - - - -
request (resource 1); /* Holding res 1 */	request (resource 2); /* Holding res 2 */	request (resource 3); /* Holding res 3 */
- - - - -	- - - - -	- - - - -
- - - - -	- - - - -	- - - - -
request (resource 2);	request (resource 3);	request (resource 1);

- Process 1 is holding resource 1 and requesting resource 2; Process 2 is holding resource 2 and requesting resource 3; Process 3 is holding resource 3 and requesting resource 1.
- None of the processes can proceed because all are waiting for a resource held by another blocked process. Unless one of the processes detects the situation and is able to withdraw its request for a resource and release the one resource allocated to it, none of the processes will ever be able to run.
- Fig. 2.1 shows the deadlock with three processes. Pictorially process is represented by circle and resource by square. Request is shown by arrow from process to resource. Holding resource by process is shown by arrow from resource to process.

- Deadlock is a global condition rather than a local one. Deadlock detection must be handled by the operating system.

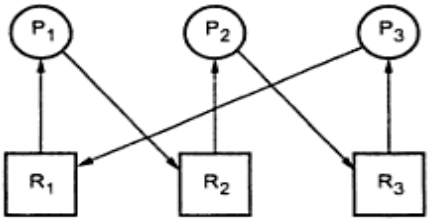


Fig. 2.1 Three deadlocked processes

2.3 Deadlock Characterization

In a deadlock, processes never finish executing and system resources are tied up, preventing other jobs from starting.

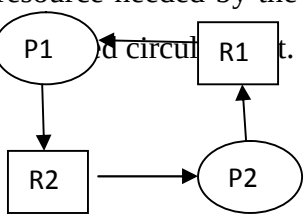
4.2 Principles of deadlock

Necessary condition

Suppose the following condition hold regarding the way a process uses resources:

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

1. Mutual Exclusion: only one process may use a resource at a time. Once a process has been allocated a particular resource, it has exclusive use of the resource. No other process can use a resource while it is allocated to a process.
2. Hold and wait: A process may hold a resource at the same time it requests another one.
3. Circular waiting: A situation can arise in which process, P1 holds resource R1 while it request resource R2 and process P2 holds R2 while it request resource R1. Each process holds at least one resource needed by the next process in the chain. There may be more than two processes in a circular set.



4. No preemption: No resource can be forcibly removed from a process holding it. Resource can be released only by the explicit action of the process, rather than by the action of an external authority.

A deadlock is possible only if all four of these conditions simultaneously hold in the community of process. These conditions are necessary for a deadlock to exist.

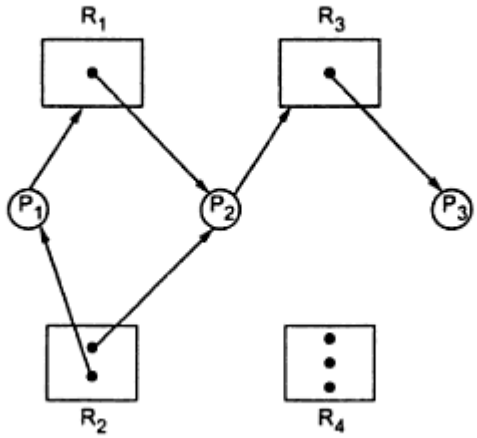
Resource Allocation Graph

Resource allocation graph is used to describe the deadlock. It is also called system resource allocation graph. Graph consist of a set of vertices (V) and set of edges (E). All the active processes in the system denoted by P={P_1, P_2,…….P_n} and set consisting of all resource type in the system is denoted by R= {R_1,R_2,R_3,……R_m}. Request edge is an edge from process to resource and denoted by P_i->R_j. An assignment edge is an edge from resource to process and denoted by R_j->P_i. holding of resource by process is denoted by assignment edge. Requesting of resource by process is denoted by request edge. For representing process and resource in the resource allocation graph is shown by square and circle. Each process is represented by circle and resource by square. Dot within the square represent the number of instances.

In the figure below of resource allocation graph the system consist of three processes i.e. P1,P2 and P3 and four resources i.e. R1,R2,R3 and R4. Resource R1 and R3 have one instance and R2 has two instances and R4 has three instances.

1) The sets P, R and E consists
   * $P = \{P_1, P_2, P_3\}$
   * $R = \{R_1, R_2, R_3, R_4\}$
   * $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
2) Resource instances
   * Resource R_1 – One instance
   * Resource R_2 – Two instances
   * Resource R_3 – One instance
   * Resource R_4 – Three instances
3) Process states
   * Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
   * Process P_2 is holding an instance of R_1 and R_2 and waiting for an instance of resource type R_3.
   * Process P_3 is holding an instance of R_3.

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist.

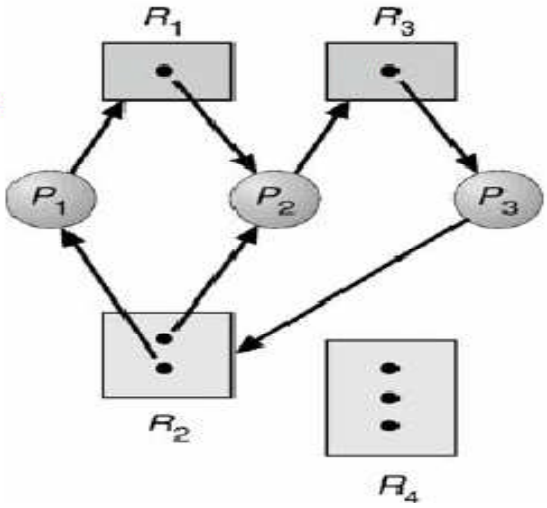


Resource-allocation graph

- **Suppose that process P_3 requests an instance of resource type R_2. Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. This is shown in Fig. 2.3. Resource-allocation graph with deadlock. At this point, two minimal cycles exist in the system.**

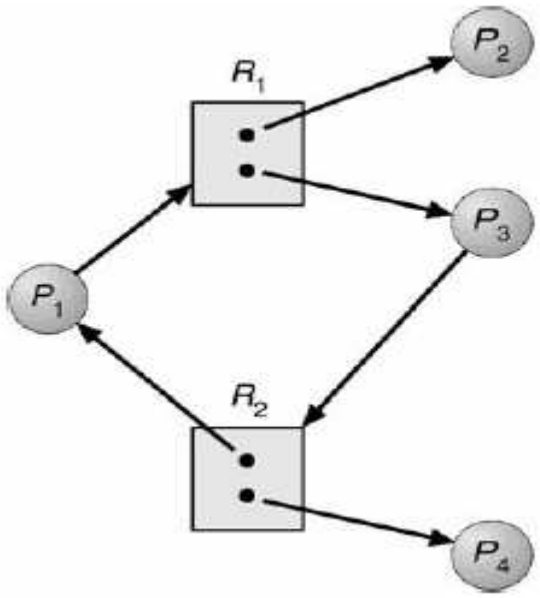
Resource Allocation Graph With A Deadlock

- Two minimal cycles exist in the system:
  - P1→R1→P2→R3→P3→R2→P1
  - P2→R3→P3→R2→P2
- Processes P1, P2, P3 are deadlocked.
  - P2 is waiting for R3, which is held by P3.
  - P3 is waiting for P1 or P2 to release R2.
  - P1 is waiting for P2 to release R1.



Resource Allocation Graph with A Cycle But No Deadlock

- We have a cycle:
  P1→ R1→ P3→R2→ P1
- There is no deadlock.
- P4 may release its instance of R2 and that resource can then be allocated to P3 breaking the cycle.
- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

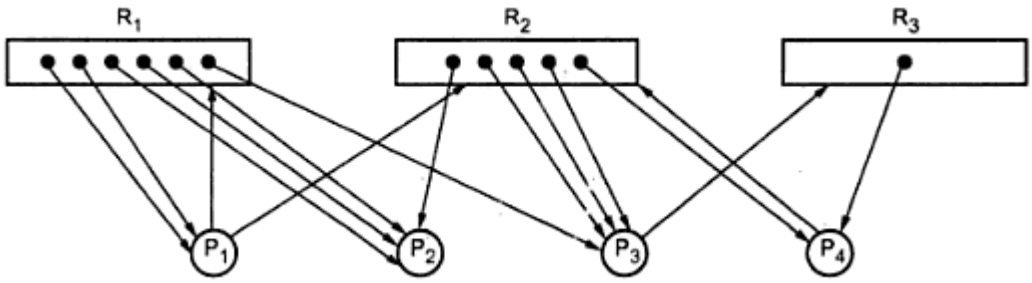


There is a cycle but no deadlock. Because the process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

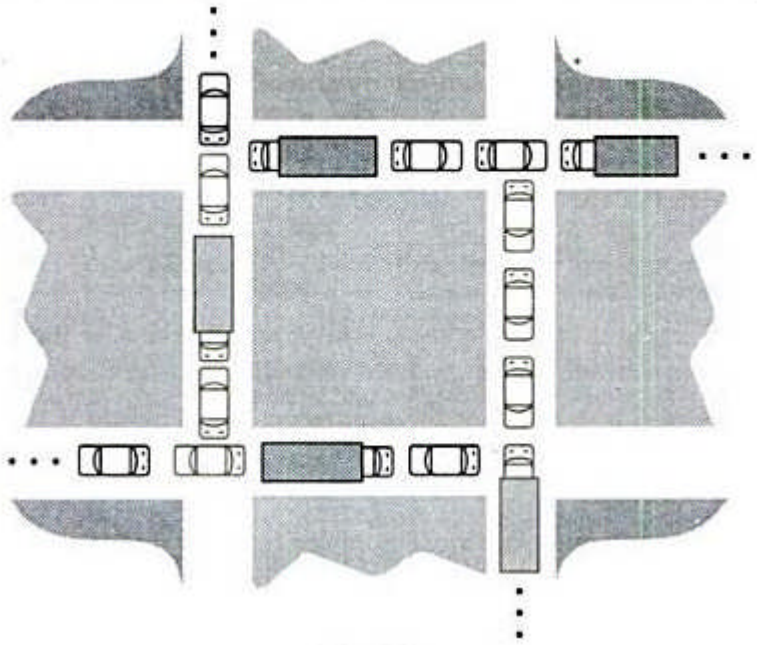
➠ **Example 2.1 :** *Given the process resource usage and availability, draw the resource-allocation graph.*

Process	Current allocation			Outstanding requests			Available resource		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	0	0	1	1	0	0	0	0
P_2	3	1	0	0	0	0			
P_3	1	3	0	0	0	1			
P_4	0	1	1	0	1	0			

Solution. : Resource-allocation graph :



Example : *Consider the traffic deadlock shown in the following figure Show that the four necessary conditions for deadlocks indeed hold in this example.*



Solution : 1. Mutual exclusion : Only one car may be occupying a particular spot on the road at any instant.

2. Hold and wait : No car ever backs up.

3. No preemption : No car is permitted to push another car out of the way.

4. Circular wait : Each corner of the city block contains vehicles whose movement depends on the vehicles blocking the next intersection.

4.3 Deadlock detection and algorithm

Methods for Handling Deadlocks or deadlock detection

There are three methods how deadlock can be handled. They are

1. Ensure that the system will *never* enter a deadlock state.
   - The system can use either a deadlock prevention or avoidance.
2. Allow the system to enter a deadlock state and then recover.
3. Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX (need to restart your computer if a deadlock occur).

- Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. If a system does not ensure that a deadlock will never occur, and also does not provide a mechanism for deadlock detection and recovery, then the system is in a deadlock state.

2.5 Deadlock Prevention

- Methods for preventing deadlock are of two classes : Indirect method and direct method.
- An indirect method is to prevent the occurrence of one of the three necessary conditions i.e. mutual exclusion, hold and wait and no preemption.
- A direct method is to prevent the occurrence of a circular wait.

2.5.1 Mutual Exclusion

- Mutual exclusion condition must hold for nonsharable resources. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the operating system.
- Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes. In this case, deadlock can occur if more than one process requires write permission.

2.5.2 Hold and Wait

- The hold and wait condition can be eliminated by forcing a process to release all resources held by it whenever it requests a resource that is not available. For example, process copies data from a floppy disk to a hard disk, sort a disk file and then prints the results to a printer.
- If all the resources must be requested at the beginning of the process, then the process must initially request the floppy disk, hard disk and a printer. It will hold the printer for its entire execution, eventhough it needs the printer only at the end.
- In these two method, resource utilization is low in the first method and second method is affected by the starvation.

2.5.3 No Preemption

This condition is also caused by the nature of the resource. This condition can be prevented in several ways.

1) If a process holding certain resources is denied a further request, that process must release its original resources and if necessary request them again, together with additional resource.
2) If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.
3) In general, sequential I/O devices cannot be preempted.
4) Preemption is possible for certain types of resources, such as CPU and main memory.

2.5.4 Circular-Wait

- One way to prevent the circular-wait condition is by linear ordering of different types of sytem resources. In this, system resources are divided into different classes. If a process has been allocated resources of type R, then it may subsequently request only those resource types following R in the ordering.
- For example, process hold the resource of class C_i, then it can only request resource of class i+1 or higher thereafter.
- Linear ordering of resource classes eliminates the possibility of circular waiting, since a process P_i holding a resource in class C_i cannot possibly wait for any process that is itself waiting for a resource in class C_i or lower.
- As with hold and wait prevention, circular wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

4.4 Deadlock Avoidance

Deadlock Avoidance

- Deadlock avoidance allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. Deadlock avoidance therefore allows more concurrency than prevention does.
- Deadlock avoidance requires additional information about how resources are to be requested. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request could, if granted, potentially lead to a deadlock.
- Two approaches are used to avoid the deadlock.
  1) Do not start a process if its demands might lead to deadlock.
  2) Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

- Fig. 2.7 shows the relationship between safe, unsafe state and a deadlock state.

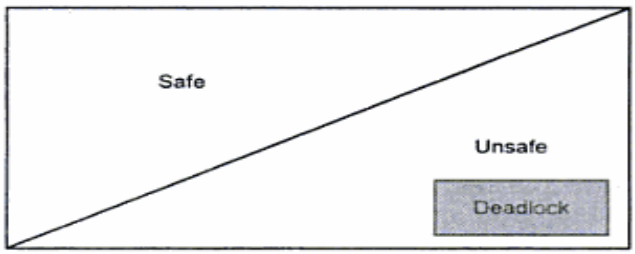


Fig. 2.7 Relationship between safe, unsafe and a deadlock state

- A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resource and the maximum demands of processes.

2.6.1 Safe State

- A **safe state** is a state in which there is at least one order in which all the processes can be run to completion without resulting in a deadlock.
- A sequence of processes $< P_1, P_2, ... P_n >$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$.
- A safe state is not a deadlock state. Conversely, a deadlock state is an unsafe state. Not all unsafe states are deadlocks. An unsafe state may lead to a deadlock. As long as the state is safe, an operating system can avoid unsafe state.
- In an unsafe state, the operating system cannot prevent processes from requesting resource such that a deadlock occurs.
- For example : A system consists of three processes P_1, P_2 and P_3 and one resource R_1. Number of units for R_1 is 12.
- Consider following case :

* Process P_1 requires 10 R_1 resource.

* Process P_2 requires 4 resource of R_1.

* Process P_3 need upto 9 resource of R_1.

At time t_0, operating system is allocated resource R_1 to the all three processes as follows.

* Process P_1 is holding 5 resource of R_1.

* Process P_2 is holding 2 resource of R_1.

* Process P_3 is holding 2 resource of R_1.

Total allocated resource is 9 and 3 resource (R_1) is free. Table 2.1 represents this situation.

Processes	Maximum request	Current allocation	Available resource (R_1)
P_1	10	5	3
P_2	4	2	
P_3	9	2	

Table 2.1

- At time t_0, the system is in a safe state. The safe sequence for this is $< P_2, P_1, P_3 >$. This sequence satisfies the safety condition. Safe sequence is calculated as follows :
  1) Process P_2 can immediately be allocated all its resource (R_1) and then return to the system.
  2) Available resource (R_1) becomes 5 after returning process P_2.
  3) Process P_1 can get all its R_1 and return them to the system. Need of process P_1 is 5 and in system R_1 is available with 5.
  4) Available resource (R_1) then becomes 10.
  5) Finally process P_3 could get all its R_1 resource and return them to the system.
  6) At last, system have all 12 R_1 resources available.

A system may go from safe state to the unsafe state.

- **Deadlock avoidance** has the advantage that it is not necessary to preempt and rollback processes as in deadlock detection. It is less restrictive than deadlock prevention.

2.6.2 Disadvantages of Deadlock Avoidances

1. The maximum resource requirement for each process must be stated in advance.
2. There must be a fixed number of resources to allocate and a fixed number of processes.
3. The processes under consideration must be independent.

2.6.3 Resource-Allocation Graph Algorithm

- In resource-allocation graph, normally request edge and assignment edge is used. In addition to the request and assignment edge, new edge called claim edge is added.
- For example, a claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This claim is shown by dotted lines in the figure. Fig.2.8 shows this deadlock avoidance with resource allocation graph.

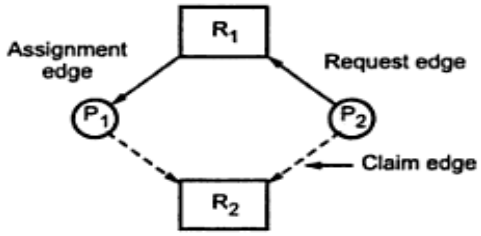


Fig. 2.8 Resource allocation graph for deadlock avoidance

a) When process P_i request resource R_j, the claim edge is converted to a request edge.

b) When a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

- Cycle detection algorithm is used for detecting cycle in the graph. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of processes in the system. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation puts the system in an unsafe state.

Banker's Algorithm

- The Banker's algorithm is the best known of the avoidance strategies. The strategy is modelled after the leading policies employed in banking system. The resource-allocation graph algorithm is suitable to a resource allocation system with single instances of each resource type. It is not suitable for multiple instance of each resource type.
- Banker's algorithm is suitable to a resource allocation system with multiple instances of each resource type. The Banker's algorithm makes decisions on granting a resource based on whether or not granting the request will put the system into an unsafe state.
- Several data structures must be maintained to implement the banker's algorithm. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures :

1. Available
2. Max
3. Allocation
4. Need

1) **Available :** A vector of length m indicates the number of available resources of each type. If available [j] = k, there are k instances of resource type R_j available.
2) **Max :** An n × m matrix defines the maximum demand of each process. If Max [i, j] = k, then process P_i may request at most k instances of resource type R_j.
3) **Allocation :** An n × m matrix defines the number of resources of each type currently allocated to each process. If allocation [i, j] = k, then process P_i is currently allocated k instances of resource type R_j.
4) **Need :** An n × m matrix indicates the remaining resource need of each process. If Need [i, j] = k, then process P_i may need k more instances of resource type R_j to complete its task. Need [i, j] – Allocation [i, j].

Safety Algorithm

- Safety algorithm is used to find the state of the system, that is, system may be in safe state or unsafe state. Method for this is as follows :
  1) Let work and finish be vector of length m and n respectively. Initialise work = Available and Finish [i] = False for i = 1, 2, 3, 4, ... n.
  2) Find an i such that both
     a. Finish [i] = False
     b. Need i ≤ work
     If no such i exist, go to step 4.
  3) Work : = Work + Allocation i
     Finish [i] = true
     go to step 2
  4) If Finish [i] = true for all i, then the system is in a safe state.

Resource-Request Algorithm

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

1) If $Request_i \leq Need_i$, then go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i \leq$ Available, then go to step 3. Otherwise, P_i must wait, since the resources are not available.

3) Available : = Available – $Request_i$;

   $Allocation_i$: = $Allocation_i$ + $Request_i$;

   $Need_i$: = $Need_i$ – $Request_i$;

If the resulting resource-allocation state is safe, the transaction is completed and process P_i is allocated its resources. If the new state is unsafe, then P_i must wait for the $Request_i$ and the old resource-allocation state is restored.

Examples on Banker's Algorithm

1. System consists of five processes $(P_1, P_2, P_3, P_4, P_5)$ and three resources (R_1, R_2, R_3). Resource type R_1 has 10 instances, resource type R_2 has 5 instances and R_3 has 7 instances. The following snapshot of the system has been taken :

Process	Allocation			Max			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	1	0	7	5	3	3	3	2
P_2	2	0	0	3	2	2			
P_3	3	0	2	9	0	2			
P_4	2	1	1	2	2	2			
P_5	0	0	2	4	3	3			

Content of the matrix need is calculated as Need = Max – Allocation.

Process	Need		
	R_1	R_2	R_3
P_1	7	4	3
P_2	1	2	2
P_3	6	0	0
P_4	0	1	1
P_5	4	3	1

Currently the system is in safe state.

Safe sequence : Safe sequence is calculated as follows :

1) Need of each process is compared with available. If $need_i \leq available_i$, then the resources are allocated to that process and process will release the resource.
2) If need is greater than available, next process need is taken for comparison.
3) In the above example, need of process P_1 is (7, 4, 3) and available is (3, 3, 2).

need ≥ available → False

So system will move for next process.

4) Need for process P_2 is (1, 2, 2) and available (3, 3, 2), so

need ≤ available (work)

(1, 2, 2) ≤ (3, 3, 2) = True

then Finish [i] = True

Request of P_2 is granted and process P_2 releases the resources.

Work = Work + Allocation

Work = (3, 3, 2) + (2, 0, 0)

= (5, 3, 2)

This procedure is continued for all processes.

5) Next process P_3 need (6, 0, 0) is compared with new available (5, 3, 2).

Need > Available = False

(6 0 0) > (5 3 2)

6) Process P_4 need (0, 1, 1) is compared with available (5, 3, 2).

Need < Available

(0 1 1) < (5 3 2) = True

Available = Available + Allocation

= (5 3 2) + (2 1 1)

= (7 4 3) (New available)

7) Then process P_5 need (4 3 1) is compared with available (7 4 3).

Need < Available

(4 3 1) < (7 4 3) = True

Available = Available + Allocation

= (7 4 3) + (0 0 2)

= (7 4 5) (New available)

8) Process P_1 need (7 4 3) and available (7 4 5). If this request is granted then the system may be in the deadlock state. After granting the request, available resource is (0 0 2) so the system is in unsafe state.

9) Process P_3 need is (6 0 0) and available (7 4 5)

$\therefore$ Need < Available

(6 0 0) < (7 4 5) = True

Available = Available + Allocation

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

= (10 4 7) = (New available)

10) Last the remaining process P_1 need (7 4 3) and available is (10 4 7)

$\therefore$ Need < Available

(7 4 3) < (10 4 7) = True

Available = Available + Allocation

= (10 4 7) + (0 1 0)

= (10 5 7)

Safe sequence is < P_2 P_4 P_5 P_6 P_1 >

Example *Consider the following snapshot of a system.*

Processes	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Answer the following questions using the banker's algorithm.

a) What is the content of the matrix need ?

b) Is the system in a safe state ?

c) If the request from process P_1 arrives for (0, 4, 2, 0) can the request be granted immediately ?

Solution : a) Content of the needed matrix is

Process	A	B	C	D
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

b) System is in safe state because resources are available (1, 5, 2, 0).

c) Request from process P_1 can be granted immediately. Request is (0, 4, 2, 0) and available resource is (1, 5, 2, 0).

➠ **Example 2.4** *The operating system contains 3 resources, the number of instance of each resource type are 7, 7, 10. The current resource allocation state is as shown below.*

Process	Current allocation			Maximum need		
	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	2	3	3	6	8
P_2	2	0	3	4	3	3
P_3	1	2	4	3	4	4

i) Is the current allocation in a safe state ?

ii) Can the request made by process P_1 (1 1 0) be granted ?

Solution : i) First find the available resource in the system.

Available = Number of instance – Sum of allocation

Process	Current allocation			
	R_1	R_2	R_3	
P_1	2	2	3	
P_2	2	0	3	
P_3	1	2	4	
	5	4	10	⇐ Sum

Available = (7 7 10) – (5 4 10)

Available Resources = (2 3 0)

Content of need matrix is

Process	Need		
	R_1	R_2	R_3
P_1	1	4	5
P_2	2	3	0
P_3	2	2	0

Safe sequence < P_3, P_2, P_1 >.

The system is in safe state.

ii) Process P_1 request (1 1 0), this request is less than need. Need for process P_1 is (1 4 5). Available resource is (2, 3, 0) and request is (1 1 0).

∴ Request < Available

(1 1 0) < (2 3 0)

After allocating (1 1 0) to process P_1 the need becomes as follows.

Process	Need		
	R_1	R_2	R_3
P_1	0	3	5
P_2	2	3	0
P_3	2	2	0

And available resource is (1 2 0).

Need of any process is never satisfied after granting the process P_1 request (1 1 0). So the system will be blocked. Therefore request of process P_1 (1 1 0) cannot be granted.

Example *Consider following snapshot.*

Process	Allocation		Max		Available	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	1	2	4	2	1	1
P_2	0	1	1	2		
P_3	1	0	1	3		
P_4	2	0	3	2		

Whether the system is safe or unsafe ?

Solution : First calculate Need

Need = Max – Allocation

Process	Need	
	R_1	R_2
P_1	3	0
P_2	1	1
P_3	0	3
P_4	1	2

System is in safe state with safe sequence < P_2, P_4, P_1, P_3 >. System will complete its operation in this sequence.

Example *Consider following snapshot.*

Process	Allocation		Max		Available	
	R_1	R_2	R_1	R_2	R_1	R_2
P_1	7	2	9	5	2	1
P_2	1	3	2	6		
P_3	1	1	2	2		
P_4	3	0	5	0		

i) Calculate content of need matrix.

ii) System is safe or unsafe.

Solution : i) Need = Max – Allocation

Process	Need	
	R_1	R_2
P_1	2	3
P_2	1	3
P_3	1	1
P_4	2	0

ii) System is unsafe state. Resources are not available for processes P_1 and P_2 to complete their operation.

Weakness of Banker's Algorithm

1. It requires that there be a fixed number of resources to allocate.
2. The algorithm requires that users state their maximum needs (request) in advance.
3. Number of users must remain fixed.
4. The algorithm requires that the bankers grant all requests within a finite time.
5. Algorithm requires that process returns all resource within a finite time.

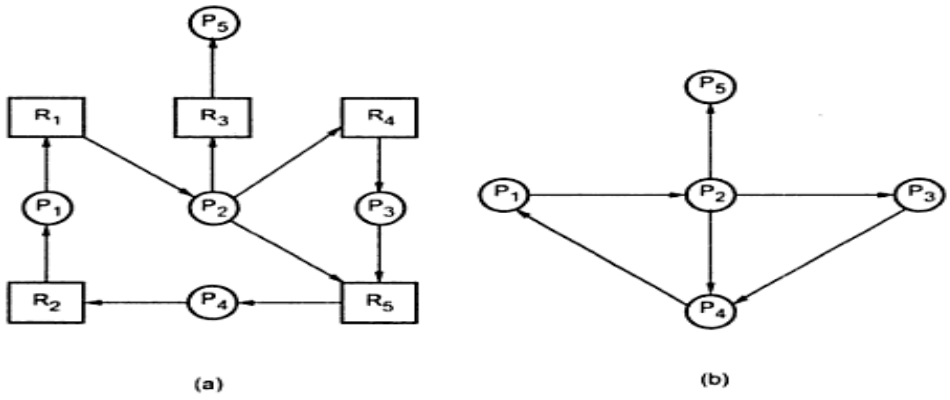
Deadlock Detection

- If the system is not using any deadlock avoidance and deadlock prevention, then a deadlock situation may occur. Deadlock detection approach do not limit resource access or restrict process actions.
- With deadlock detection request resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition. We discuss the algorithm for deadlock detection.

Single Instance of Each Resource Type

- If all resources have only a single instance, then deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for graph.**
- Wait-for graph is obtained from resource-allocation graph. Nodes of resource is removed and collapsing the appropriate edge. i.e. assignment and request edge.
- Fig. shows the resource allocation graph with corresponding wait-for graph. The assumption made in the wait-for graph is as follows.

1) 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.
2) 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 $R_q \rightarrow P_i$ for some resource R_q.



Resource-allocation graph with corresponding wait for graph

- In the figure of resource-allocation graph, process P_1 is holding resource R_2 and requesting resource R_1. So this is shown in the following figure. In this, process P_1 is request edge with process P_2.

Request edge

- If the wait for graph contains the cycle, then there is a deadlock. To detect deadlock, the system needs to maintain the wait for graph and periodically to invoke an algorithm that searches for a cycle in the graph. For detecting a cycle in the graph, algorithm requires n^2 operation. Where n is the number of vertices in the graph. This method is suitable for single instance of resource.

Several Instances of a Resource Type

- In deadlock detection approaches, the resource allocator simply grants each request for an available resource. For checking for deadlock of the system, the algorithm is as follows.
  1) Unmark all active processes from allocation, Max and Available in accordance with the system state.
  2) Find an unmarked process i such that

     $$Max_i \leq \text{Available}$$

     If found, mark process i, update Available

     $$\text{Available} := \text{Available} + \text{Allocation}$$

     and repeat this step.

     If no process is found, then go to next step.
  3) If all processes are marked, the system is not deadlocked.

Otherwise system is in deadlock state and the set of unmarked processes is deadlocked.

- Deadlock detection is only a part of the deadlock handling task. The system break the deadlock to reclaim resources held by blocked processes and to ensure that the affected processes can eventually be completed.

Deadlock Recovery

- Once deadlock has been detected, some strategy is needed for recovery. Following are the solutions to recover the system from deadlock.
  1. Process termination
  2. Resource preemption

1 Process Termination

- All deadlocked processes are aborted. Most of the operating system use this type of solution
  1. Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense.
  2. Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead. The order in which processes are selected for abortion should be on the basic of some criterion of minimum cost.
- Many factors may affect which process is chosen, including :
  a. Least amount of processor time consumed so far.
  b. Least amount of output produced so far.

c. Lowest priority

d. Most estimated time remaining.

e. Least total resources allocated so far.

2 Resource Preemption

- If preemption is required to deal with deadlocks then three issues need to be addressed :
  1. Selecting a victim. Which resources & Which processes are to be preempted.
  2. Rollback. Backup each deadlocked process to some previously defined check point and restart all processes. This requires that rollback & restart mechanism are build in to the system.
  3. Starvation. How do we ensure that starvation will not occur?

Advantages and Disadvantages

Advantages of Deadlock Prevention

1) No Preemption necessary.
2) Works well for processes that perform a single burst of activity.
3) Needs no run-time computation.
4) Feasible to enforce via compile time checks.

Disadvantages of Deadlock Prevention

1) Inefficient.
2) Delays process initiation.
3) Subject to cyclic restart.
4) Disallows incremental resource requests.

Advantages of Deadlock Detection

1) Never delays process initiation.
2) Facilitates online handling.

Disadvantages of Deadlock Detection

1) Inherent preemption losses.

Advantages of Deadlock Avoidance

1) No preemption necessary.

Disadvantages of Deadlock Avoidance

1) Future resource requirements must be known.
2) Processes can be blocked for long periods.

An Integrated Deadlock Strategy

- All the deadlock strategies have their own strengths and weakness. Instead of using one strategy for designing operating system, all the deadlock strategy are used according to the situation.
- Some of the approaches are :
  1) Group resources into a number of different resource classes.
  2) Within a resource class, use the algorithm that is most appropriate for that class.
- Resources are classified as belows :
  1. Swappable space
  2. Process resource
  3. Main memory
  4. Internal resource i.e. I/O channels.
- Swappable space means blocks of memory on secondary storage for use in swapping process. Tape drive and files are process resources. Assignable in pages or segments to processes are the main memory resources.

Chapter 5: Memory Management (5hrs)

Memory is an important resource that must be carefully managed. Most computers have a **memory hierarchy**, with a small amount of very fast, expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, and volatile main memory (RAM), and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used. The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and de-allocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

Memory Management Requirements

Memory management is intended to satisfy the following requirements:

1. Relocation: The ability to load and execute a given program into an arbitrary place in a memory is relocation. Relocation is a mechanism to covert logical or virtual address into physical address. The address generated by CPU is logical or virtual address and address that is generated by memory manager is known as physical address.
2. Protection: Protection means providing security from unauthorized usage of memory. The OS can protect the memory with the help of base and limit register. Base register consist of the starting address of the next process and limit register specifies the boundary of that job.
3. Sharing: the process of accessing the same portion of the main memory by a number of processes is known as sharing.
4. Logical organization

   The main memory of a computer is organized as a linear or 1-dimensional address apace consisting of sequence of bytes of words. At physical level, the secondary memory is similarly organized. This organization closely mirrors the actual machine hardware but it does not correspond to the way in which program are typically constructed. Most programs are organized into module some of which are modifiable and some are non-modifiable. If the OS and computer hardware can effectively deal with the user program and data in the form of modules of some sort then a number of advantage can be realized.

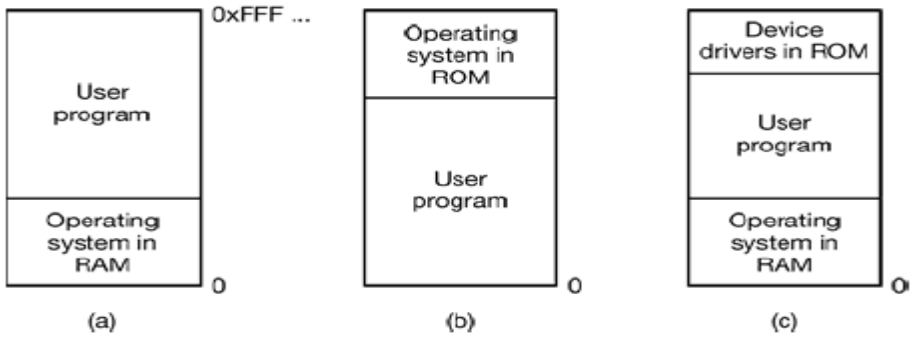
5. Physical organization

   Main memory provides fast access at relatively high cost. The main memory is volatile; it does not provide permanent storage. Secondary memory is slower and cheaper than main memory and is usually non-volatile. Secondary memory of large capacity can be provides to allow for long term storage of program and data.

Mono-programming without swapping and paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in Figure. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Figure (a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Figure (b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Figure (c). The first model was formerly used on mainframes and minicomputers but is rarely used any more. The second model is used

on some palmtop computers and embedded systems. The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the **BIOS** (Basic Input Output System).



When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

Multiprogramming

Most modern systems allow multiple processes to run at the same time. Having multiple processes running at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization.

In multiprogramming, there are two Memory allocation techniques. They are

1. Contiguous memory allocation
2. Non-contiguous memory allocation

1. Contiguous memory allocation

   In contiguous memory allocation a memory resident program occupies a single contiguous block of physical memory. The memory is partitioned into blocks of different sizes to accommodate the program. The partitioned allocation technique was introduced to solve the problem of wasted time and memory. The partitioning is of two types

   a. Fixed-partition allocation
   b. Variable partition allocation

Fixed-partitioning allocation

In partition allocation, main memory is divided into a number of partitions, whereas in fixed size of partitioning allocation, partition can be of equal or different sizes. The number and size of partitions are fixed, thus also fixing the degree of multiprogramming.

Partitioned Allocation

The partitioned allocation technique was introduced to solve the problems of wasted time and memory. This approach is based upon partitioning of memory to accommodate more programs in the main memory. There are two ways to multiprogramme storage management in contiguous storage allocation systems.

1. Fixed Size Partitioning Allocation
2. Variable Size Partitioning Allocation (Relocation Allocation)

1. Fixed Size Partitioning Allocation

In partitioning allocation, main memory is divided into a number of partitions, whereas in fixed size partitioning allocation, partitions can be of equal or different sizes. The number and sizes of partitions are fixed, thus also fixing the degree of multiprogramming. The operator can change the size and the number of partitions only when no programs are executing in the affected partitions. All job step programs of a job run in the same partition. Hence, a user has to be aware of partition sizes and sizes of his various job step programs while specifying the partition in which his program will run. Fig.4.12 show the equal and unequal fixed-size partitioning.

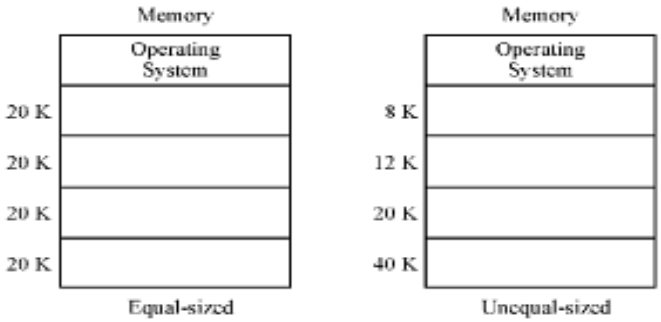


Fig.4.12 Fixed size partitioning allocation

Case 1: Equal Sized Partitions: In case of equal size partitions, if a program size is 30 K then it cannot reside in the main memory and therefore, cannot be executed. Similarly, if a program size is 12K then 8KB of memory will be wasted.

Case 2: Unequal Sized Partitions: In case of unequal size partitions, for instance, if a program size is 10KB, then it can be executed if any of the partition except 8KB. If the 10KB program is executed in 12K, or 20K, or 40K, then 2KB or 10KB or 30KB of memory will be wasted, respectively. To overcome this problem, variable sized partitioning is introduced.

2. Variable Size Partitioning Allocation (Relocation Allocation)

In variable size partitioning allocation, a memory of exactly the same size is allocated when a program is brought into the memory. Thus, there is no wastage of memory. As jobs terminate, the system keeps track of the released storage. When a new job is to be initiated, it creates a partition to suit its storage requirements. Fig.4.13 shows that in due course of time, memory becomes fragmented.

It can be observed in Fig.4.13 that there are four jobs in memory (Fig.4.13(*a*)), when fragmentation prevents a new job from being loaded. The job that remains in the memory can be compacted by relocating them upward. Note that when Job A and Job D are finished

(Fig.4.13(*b*)), Job B and Job C are moved upward so that Job E can be fitted into the contiguous area formed (Fig.4.13(*c*)).

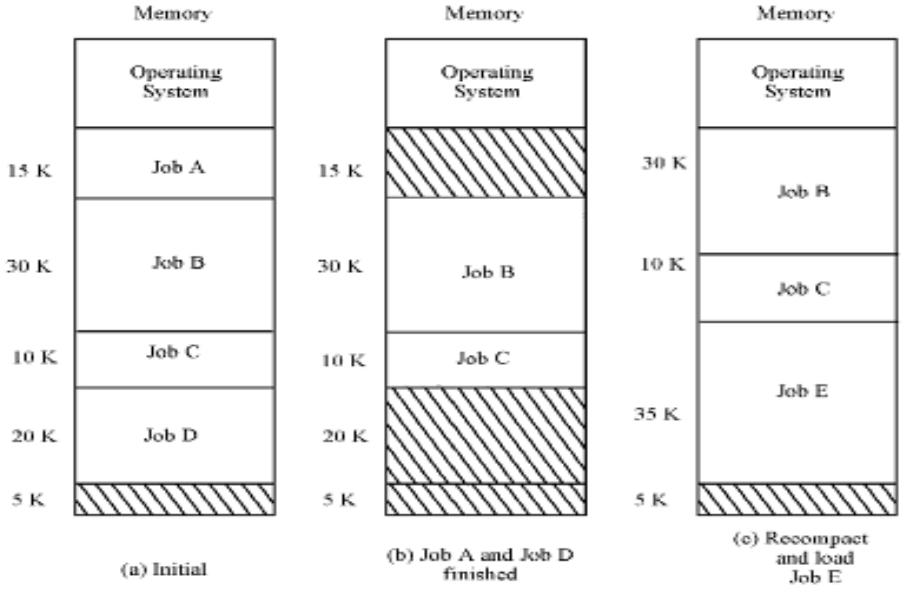


Fig.4.13 Variable Sized Partitions (dynamically relocation)

2. Non-contiguous memory allocation
    a. Paging
    b. swapping

Memory management

In operating system memory management is the essential task that should be managed in the proper way. There are two method used in memory management. They are

1. Bit map
2. Linked list

Bit maps

When memory is assigned dynamically, the operating system must manage it. With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure shows part of memory and the corresponding bitmap.

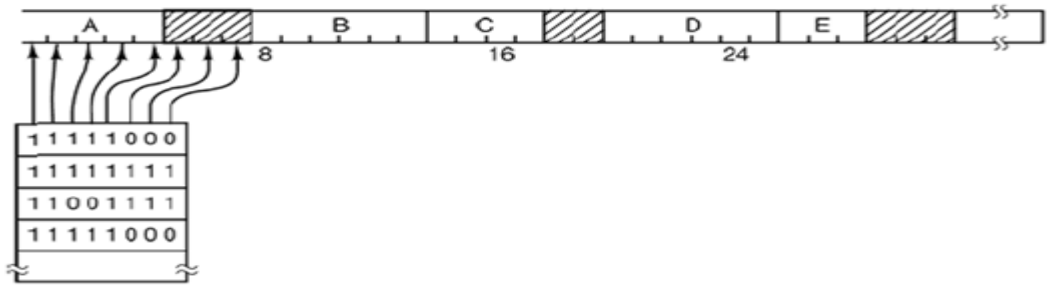
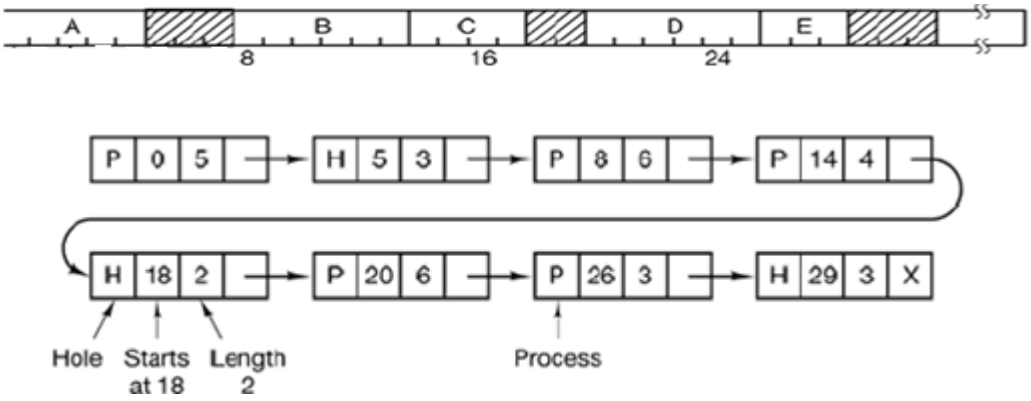


Figure: A part of memory with five processes and three holes, the tick marks show the memory allocation units and the shaded regions (0 in the bitmap) are free with the corresponding bitmap.

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem with it is that when it has been decided to bring a *k* unit process into memory, the memory manager must search the bitmap to find a run of *k* consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

Linked list (First fit, best fit. Next fit, quick fit and buddy system)



Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. The memory is managed using linked list of segments. Each entry in the list specifies a Hole (H) or Process (P), the address at which it starts, the length, and a pointer to the next entry. In this

example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations of Figure.

The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does.

Another well-known algorithm is **best fit**. Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

Another allocation algorithm is **quick fit**, which maintains separate lists for some of the more common sizes requested. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbor to see if a merge is possible is expensive. If merging is done, memory will quickly fragment into a large number of small holes into which no process fit.

Buddy system: Both fixed and dynamic partition have drawback. A fixed partition limits the number of active process and may use space inefficiently if there is a poor match between available partition size and process sizes. A dynamic is more complex to maintain and includes the overhead of compaction. But in buddy system, memory block are available of size 2^K words, L<=K<=U where,

2^L = smallest size block that is allocated

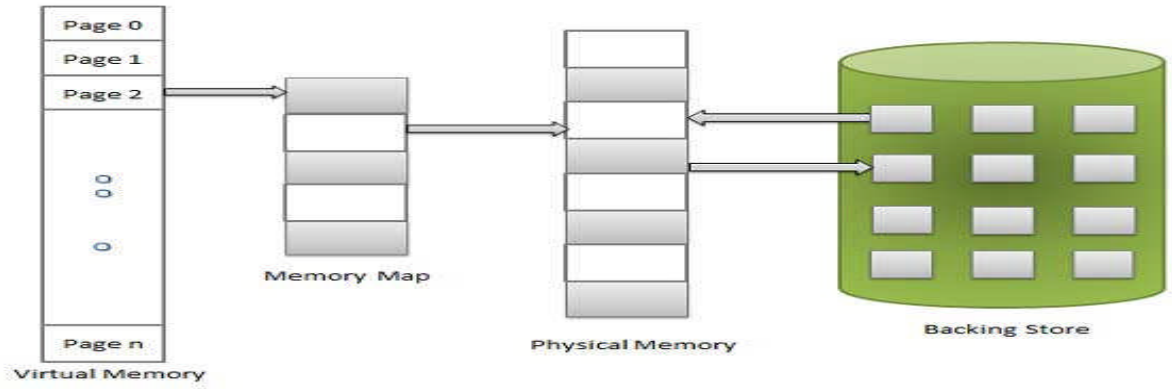
2^U= Largest size block hat is allocated; generally 2U is the size of the entire memory available for allocation.

5.3 Multiprogramming memory management techniques

5.4 Virtual Memory

Virtual memory is a feature of an operating system (OS) that allows a computer to compensate for shortages of physical memory by temporarily transferring pages of data from random access memory (RAM) to disk storage. Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory.

Virtual memory management techniques split a program into a number of pieces and perform swapping. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.



Advantage

1. The size of user's program would be no longer constrained by the available size of physical memory.

2. Since each user utilizes less physical memory, more users can keep their program in virtual memory so increase the CPU utilization and throughput.

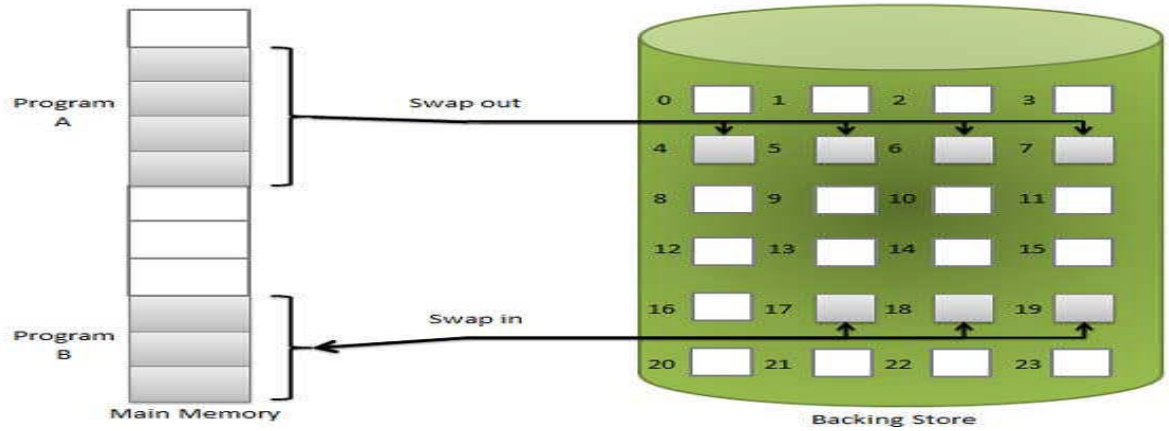
3. Degree of programming can be varied over a large range.

4. Since a process may be loaded into a space of arbitrary size, which in turn reduces external fragmentation without the need to change the scheduled order of process execution?

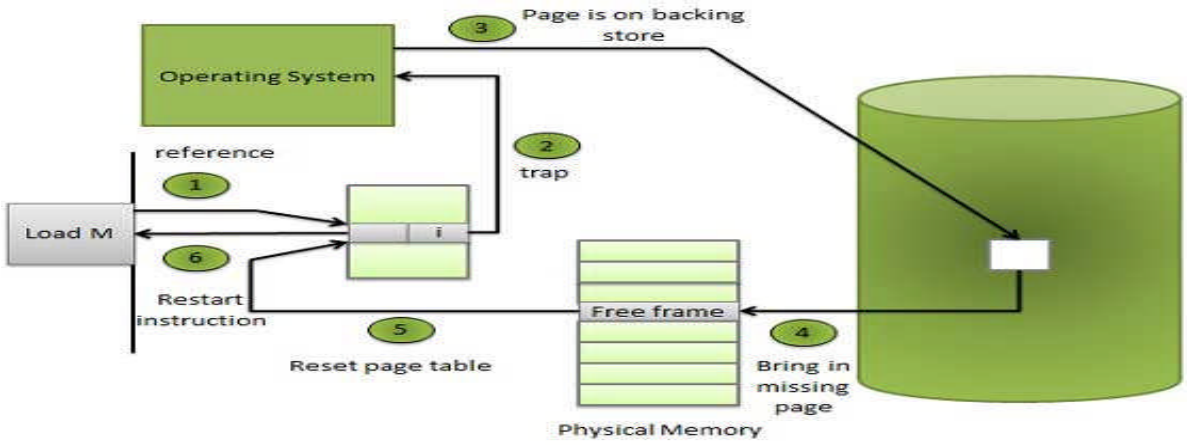
Demand Paging

A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. It is a schema in which a page is not loaded into the main memory from the secondary memory, until it is needed. So in demand paging, pages are loaded only demand not in advance. When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme, where valid and invalid pages can be checked by checking the bit. Marking a page will have no effect if the process never attempts to access the page. While the process executes and accesses pages that are memory resident, execution proceeds normally.



Access to a page marked invalid causes a **page-fault trap**. This trap is the result of the operating system's failure to bring the desired page into memory. But page fault can be handled as following



Step	Description
Step 1	Check an internal table for this process, to determine whether the reference was a valid or it was an invalid memory access.
Step 2	If the reference was invalid, terminate the process. If it was valid, but page have not yet brought in, page in the latter.
Step 3	Find a free frame.
Step 4	Schedule a disk operation to read the desired page into the newly allocated frame.
Step 5	When the disk read is complete, modify the internal table kept with the process and the page table to indicate that the page is now in memory.
Step 6	Restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory. Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

Advantages

Following are the advantages of Demand Paging

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

Disadvantages

Following are the disadvantages of Demand Paging

- Number of tables and amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.
- Due to the lack of explicit constraints on jobs address space size.

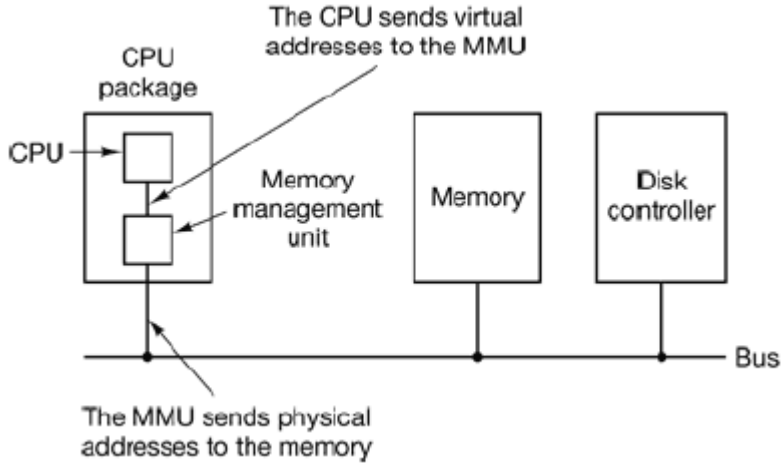
5.5.1 **Paging and segmentation**

Paging

Paging is a memory management scheme that allows processes physical memory to be discontinuous, and which eliminates problems with fragmentation by allocating memory in equal sized blocks known as **pages**. Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today. Most virtual memory systems use a technique called **paging**. On any computer, there exists a set of memory addresses that programs can produce. When a program uses an instruction like

MOV REG, 1000

it does this to copy the contents of memory address 1000 to REG (or vice versa, depending on the computer). Addresses can be generated using indexing, base registers, segment registers, and other ways.



These program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU (Memory Management Unit)** that maps the virtual addresses onto the physical memory addresses as illustrated in above Figure.

The virtual address space is divided up into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are always the same size. In this example they are 4 KB, but page sizes from 512 bytes to 64 KB have been used in real systems. With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames. Transfers between RAM and disk are always in units of a page.

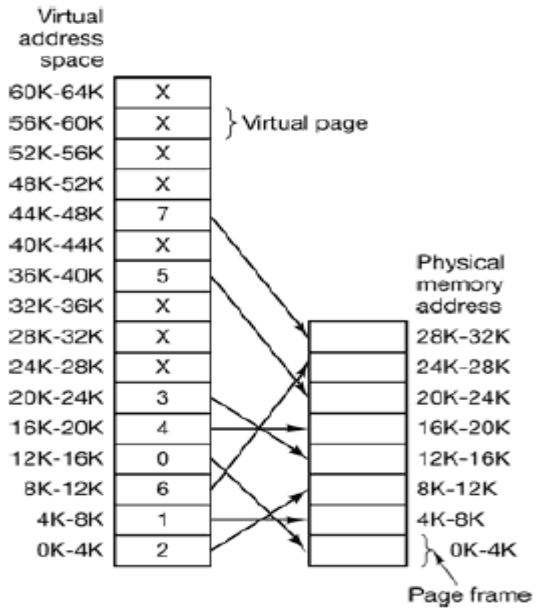
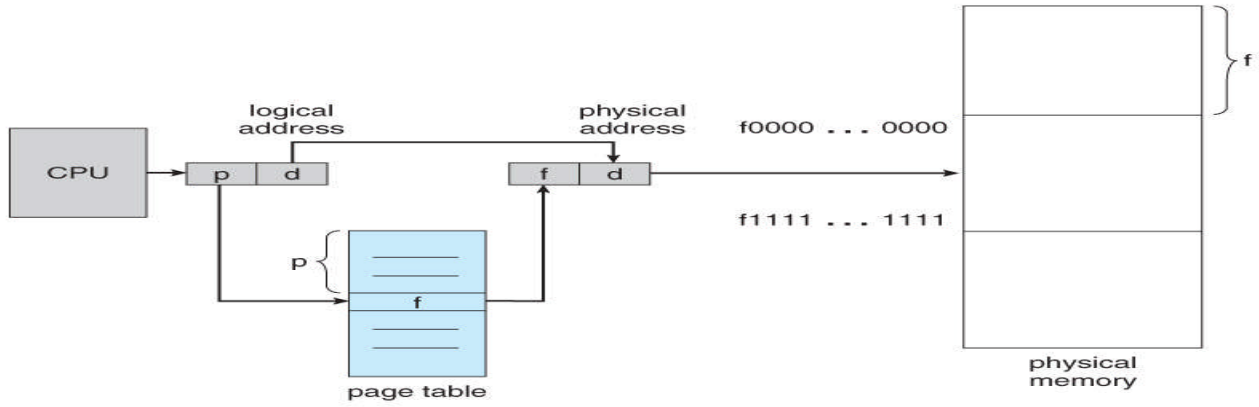


Figure: The relation between virtual addresses and physical memory addresses is given by the page table.

Basic Method

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

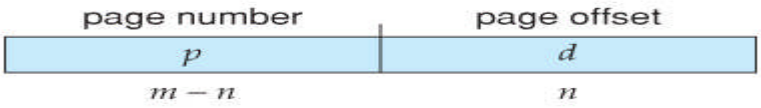


Paging model of logical and physical memory

A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the maximum size of each page, and should correspond to the system frame size.)

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

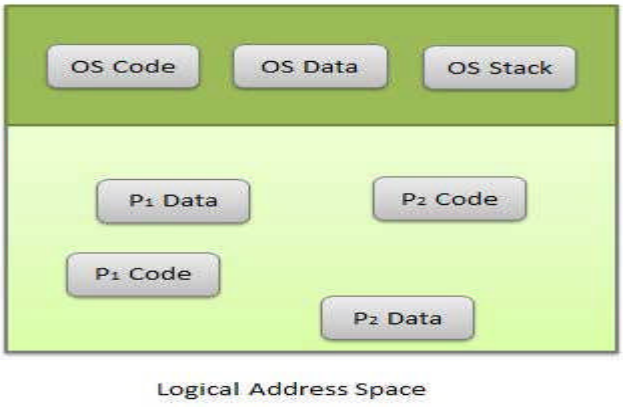
page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.



Segmentation

Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information. For example, data segments or code segment for each process, data segment for operating system and so on. Segmentation can be implemented using or without using paging.

Unlike paging, segments are having varying sizes and thus eliminate internal fragmentation. External fragmentation still exists but to lesser extent.



Logical Address Space

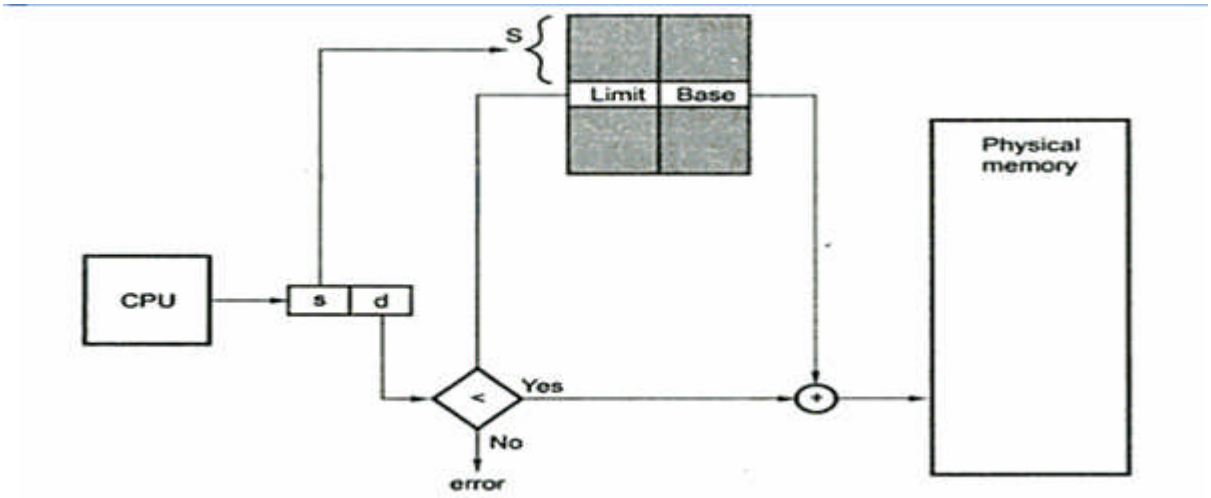
- Code segment, where the program code is stored
- Data segment, where variables defined by the code are stored whilst the application is running.
- Stack segment, starts from the top of the segment and grows downwards. The stack is used by sub-routines and interrupts service routines to hold temporary data and addresses.

Segment table is used by memory management unit which contain base and segment size columns. Base contains starting physical address where segment resides in memory and size

gives the size of segment. CPU generates logical or virtual address. Address generated by CPU is divided into

1. **Segment number (s)** -- segment number is used as an index into a segment table which contains base address of each segment in physical memory and a limit of segment.

2. **Segment offset (o)** -- segment offset is first checked against limit and then is combined with base address to define the physical memory address. If the offset is not within the range it is trapped by the operating system otherwise the offset is added to the base address of the segment to produce the physical address of the desired segment.



Advantage and disadvantage

Advantage

1. Sharing of code or data is possible. So there is no need of sharing the data or code twice in physical memory.
2. Protection bits are associated with segments. It provides protection by not allowing writing to a segment that is read-only.
3. Segment can grow dynamically also.
4. Segmentation supports virtual memory.

Disadvantage

1. As with paging, this mapping requires two memory references per logical address. This slows the system by a factor of two.
2. When the number of segment is large then the size of the segment table will also grow.
3. Segmentation is prone to external fragmentations.

Difference between Segmentation and Paging

Sr. No.	Segmentation	Paging
1.	Program is divided into variable size segments.	Program is divided into fixed size pages.
2.	User (or compiler) is responsible for dividing the program into segments.	Division into pages is performed by the operating system.
3.	Segmentation is slower than paging.	Paging is faster than segmentation.
4.	Segmentation is visible to the user.	Paging is invisible to the user.
5.	Segmentation eliminates internal fragmentation.	Paging suffers from internal fragmentation.
6.	Segmentation suffers from external fragmentation.	There is no external fragmentation.
7.	Processor uses page number, offset to calculate absolute address.	Processor uses segment number, offset to calculate absolute address.
8.	OS maintain a list of free holes in main memory.	OS must maintain a free frame list.

5.5.2 Swapping and page replacement

Swapping

With a batch system, organizing memory into fixed partitions is simple and effective. Each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated. With timesharing systems or graphically oriented personal computers, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must he kept on disk and brought in to run dynamically.

The simplest strategy, called **swapping**, consists of bringing in each process in its entirety, running it for a while, and then putting it back on the disk.

Swapping is a technique of temporarily removing inactive program from the memory of a system. it removes the process from the primary memory when it is blocked and de-allocating the memory. This free memory is allocated to other processes. Figure below shows the swapping of process. For example, when process P1 request an I/O operation, it becomes blocked and will not return to the ready state for relatively long period of time. process manage places the process P1 into a blocked state, then the memory manager will decide whether to swap the process from primary memory to secondary memory. Process P2 changes the state from swap the address space back into primary memory either immediately if primary memory is available or at least as soon as it becomes available.

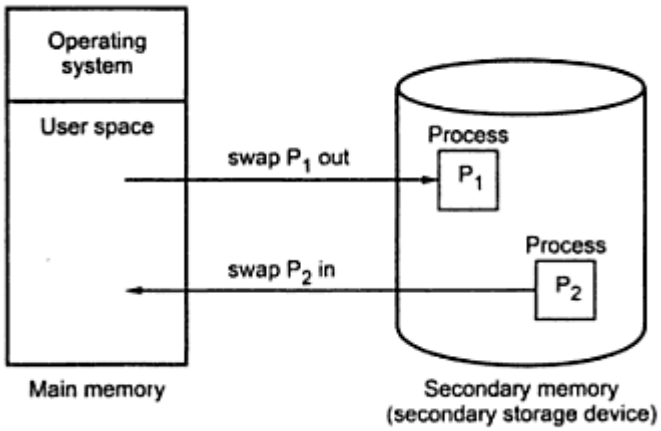


Fig. Swapping

When process is swapped out, its executable image is copied to secondary memory. When the process is swapped back into available primary memory, the executable image that was swapped out is copied into the new block allocated by the memory manager. Swapped out process will be swapped back into the same memory space that is occupied previously. It also depends upon method of binding.

1. If the address binding is done at load time, then the process is moved to same location of previously one.
2. If the address binding is done at execution time, then a process can be swapped into a different memory space. This is because of the physical address which is computed during execution time.

Page replacement:

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page is to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted. While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. An example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except for the fact that the Web pages are never modified in the cache, so there is always a fresh copy on disk. In a virtual memory system, pages in main memory may be either clean or dirty.

When a page in a memory is needed to be replaced with a new page then the page replacement algorithm is required.

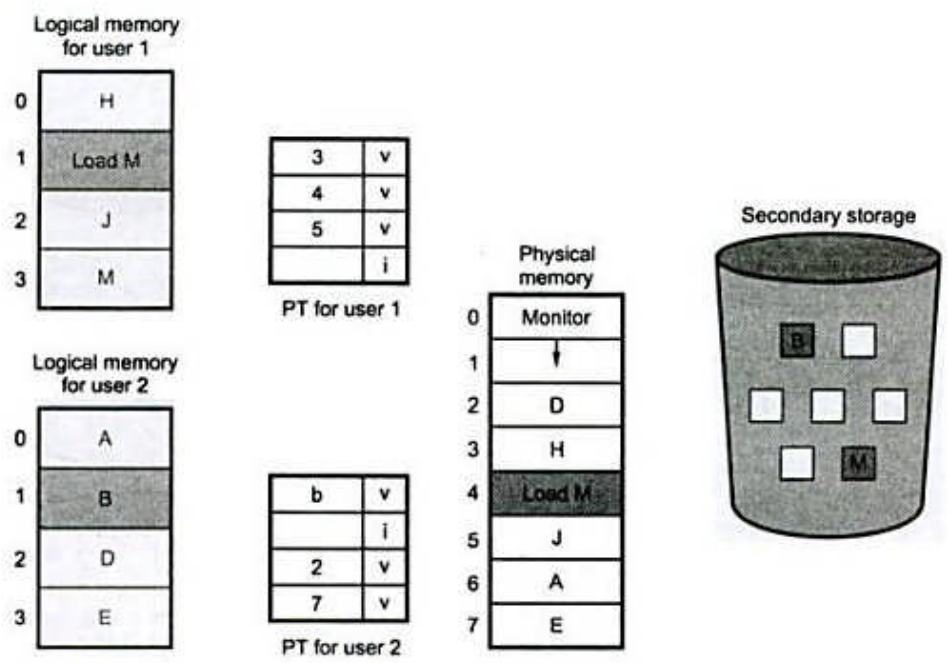
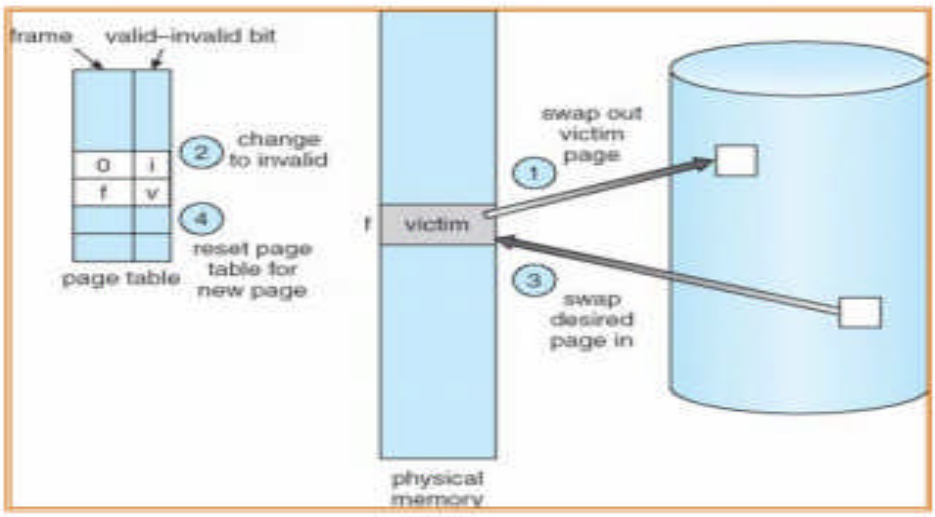


Fig. 8.7 Need for page replacement

Algorithm for page replacement algorithm

The basic algorithms for page replacement are as follows

1. Find the location of the desired page on the disk.
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 a victim frame.
    c. Write a victim page to the disk, change the page and frame tables accordingly
3. Read the desired page into the free frame, change the page and frame tables.
4. Restart the user process.



b. If there is no free frame, use a page replacement algorithm to select a victim frame.

c. Write a victim page to the disk, change the page and frame tables accordingly.

3. Read the desired page into the free frame, change the page and frame tables.

4. Restart the user process.

Fig. 8.8 shows the page replacement.

Please refer Fig. 8.8 on next page.

8.3.2 Memory Reference String

The string of memory references is called a reference string. A succession of memory references made by a program executing on a computer with 1 MB of memory is given below in hex notation.

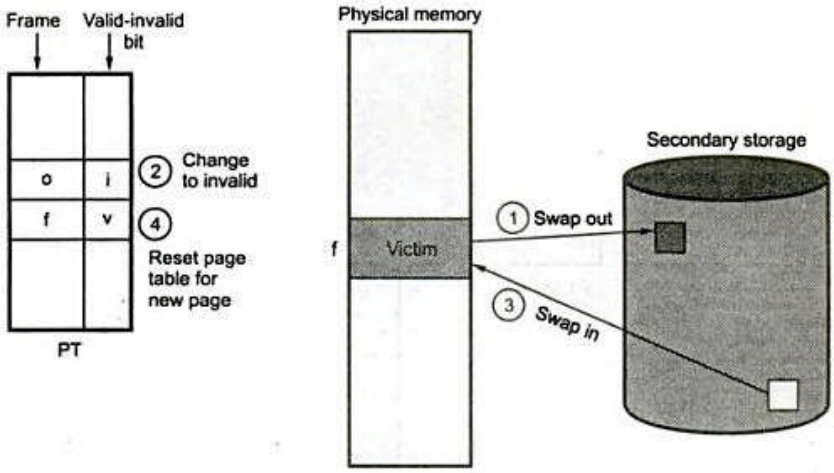


Fig. 8.8 Page replacement

....., 14489, 1448B, 14494, 14496, A1F8, 14497, 14499, 2638E, 1449A, ...

When analyzing page replacement algorithms, we are interested only in the pages being referenced. Assuming a 256 byte page size the referenced pages are obtained simply by omitting the two least significant hex digits.

... 144, 144, 144, 144, A1, 144, 144, 263, 144, ...

The pattern of page reference above can be compared into a reference string for page replacement analysis as follows :

... 144, A1, 144, 263, 144, ...

A reference string obtained in this way is used to illustrate most of the following replacement algorithms.

There are different types of page replacement algorithm in the memory management in operating system.

1. First In First Out Page Replacement Algorithm

FIFO is the simplest page replacement algorithm, the basic idea behind this is “Replace a page that page is the oldest page of all the page of the main memory” or “Replace the page that has been in memory longest”.

Let us consider the reference string

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

Frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	5	5	5	5	3	3	3	3
1		3	3	3	3	2	2	2	2	2	5	5
2				1	1	1	4	4	4	4	4	2
Fault	*	*		*	*	*	*		*		*	*

Number of fault= 9

5 is inserted in the place of 2 as 2 has been for the longest period and 2 is the oldest one.

2 is inserted in the place of 3 as 3 is in the frame for the maximum time.

2. The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced. The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	1	1	1	4	4	4	7
1		1	1	1	1	1	2	2	2	2	2	2	2	5	5	5
2			2	3	3	3	3	3	3	3	3	3	3	3	6	6
Fault	*	*	*	*			*			*			*	*	*	*

3 is inserted in 2 as 2 is not used for longest period

2 is inserted in 1 as 1 is far from 2. After 2 there is 3 and 0 which is already in the frame.1 is replaced by 2

4 is inserted in the place of 1 as 1 2 3 is not been used after this. So 4 is inserted in the 1st position and after that 5 is inserted in the 2nd frame and 6 is inserted in 3rd frame. And again 7 is inserted in the 1st position.

Number is fault=10

3. Least Recently Used (LRU)

The basic idea for this algorithm is "Replace a page that has been not been used for the longest period". In least recently used algorithm we look backward to see which is far from the current string that is to be updated.

Let us consider reference string

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

Frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	2	2	2	2	3	3	3	3
1		3	3	3	5	5	5	5	5	5	5	5
2				1	1	1	4	4	4	2	2	2
Fault	*	*		*	*		*		*	*		

5 is inserted in the place of 3. As we look backward before 5 there is 1 and before 1 there is 2 so 3 is far from the 5 which is replaced by the 5.

Number of page fault=7

Example 8.3 : *Consider the following page-reference string :*
2 3 2 1 5 2 4 5 3 2 5 2
How many page faults occur for the following replacement algorithms, assuming three frames. i) FIFO ii) LRU iii) Optimal

Ans. : i) FIFO

frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	5	5	5	5	3	3	3	3
1		3	3	3	3	2	2	2	2	2	5	5
2				1	1	1	4	4	4	4	4	2
Page fault	*	*		*	*	*	*		*		*	*

Number of page fault = 9

ii) LRU

frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	2	2	2	2	3	3	3	3
1		3	3	3	5	5	5	5	5	5	5	5
2				1	1	1	4	4	4	2	2	2
Page fault	*	*		*	*		*		*	*		

Number of page fault = 7

iii) Optimal

frame	2	3	2	1	5	2	4	5	3	2	5	2
0	2	2	2	2	2	2	4	4	4	2	2	2
1		3	3	3	3	3	3	3	3	3	3	3
2				1	5	5	5	5	5	5	5	5
	*	*		*	*		*			*		

Number of page fault = 6

Example 8.4 : *Consider the following page reference. Indicate page faults and calculate total number of page faults for optimal and LRU. The total number of available frames are 4. 1, 2, 3,2, 5, 6, 3, 4, 6, 3, 7, 3, 1, 5, 3, 6, 3, 4, 2, 4, 3, 4, 5, 1.*

Ans. : i) Optimal (frames = 4)

Reference string	0	1	2	3	Page fault
1	1	–	–	–	*
2	1	2	–	–	*
3	1	2	3	–	*
2	1	2	3	–	
5	1	2	3	5	*
6	1	6	3	5	*
3	1	6	3	5	
4	1	6	3	4	*
6	1	6	3	4	
3	1	6	3	4	
7	1	6	3	7	*
3	1	6	3	7	
1	1	6	3	7	
5	1	6	3	5	*
3	1	6	3	5	
6	1	6	3	5	
3	1	6	3	5	
4	1	4	3	5	*
2	2	4	3	5	*
4	2	4	3	5	
3	2	4	3	5	
4	2	4	3	5	
5	2	4	3	5	
1	1	4	3	5	*

Number of page fault = 11

ii) LRU

Reference string	0	1	2	3	Page fault
1	1	–	–	–	*
2	1	2	_	–	*
3	1	2	3	–	*
2	1	2	3	–	
5	1	2	3	5	*
6	6	2	3	5	*
3	6	2	3	5	
4	6	4	3	5	*
6	6	4	3	5	
3	6	4	3	5	
7	6	4	3	7	*
3	6	4	3	7	
1	6	1	3	7	*
5	5	1	3	7	*
3	5	1	3	7	
6	5	1	3	6	*
3	5	1	3	6	
4	5	4	3	6	*
2	2	4	3	6	*
4	2	4	3	6	
3	2	4	3	6	
4	2	4	3	6	
5	2	4	3	5	*
1	1	4	3	5	*

Number of page fault = 14

4. The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

In this algorithm **second chance** is given to the string.

Let us consider reference string

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

Frame	2	3	2	1	5	2	4	5	3	2	5	2
	2	2	(1)2	2	(0)2	(1)2	(0)2	2	3	3	3	3
		3	3	3	5	5	5	(1)5	(0)5	5	(1)5	(1)5
				1	1	1	4	4	4	2	2	(1)2
Fault	*	*		*	*		*		*	*		

2 is inserted again so its reference is increased to 1.

Though 2 is in the frame for longer period it cannot be removed as it is given second chance be making reference to 1 so 5 should be removed in the place of 3 and 1. 3 is in the frame for longer period it is removed. And 2 that have referenced 1 set to 0.

Chapter 6: Real Time Operating System (2 hrs)

6.1 Introduction and Example

Introduction

In general, an operating system (OS) is responsible for managing the hardware resources of a computer and hosting applications that run on the computer. An RTOS performs these tasks, but is also specially designed to run applications with very precise timing and a high degree of reliability. This can be especially important in measurement and automation systems where downtime is costly or a program delay could cause a safety hazard. Real time Operating Systems are very fast and quick respondent systems. These systems are used in an environment where a large number of events (generally external) must be accepted and processed in a short time. Real time processing requires quick transaction and characterized by supplying immediate response. For example, a measurement from a petroleum refinery indicating that temperature is getting too high and might demand for immediate attention to avoid an explosion.

Real time system is defines as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. Real time processing is always on line whereas on line system need not be real time. The time taken by the system to respond to an input and display of required updated information is termed as response time. So in this method response time is very less as compared to the online processing. Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. Real-time operating system has well-defined, fixed time constraints otherwise system will fail. For example Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, and home-appliance controllers, Air traffic control system etc.

The primary functions of the real time operating system are to:

1. Manage the processor and other system resources to meet the requirements of an application.

2. Synchronize with and respond to the system events.

3. Move the data efficiently among processes and to perform coordination among these processes.

The Real Time systems are used in the environments where a large number of events (generally external to the computer system) is required to be accepted and is to be processed in the form of quick response. Such systems have to be the multitasking. So the primary function of the real time operating system is to manage certain system resources, such as the CPU, memory, and time. Each resource must be shared among the competing processes to accomplish the overall function of the system Apart from these primary functions of the real time operating system there are certain secondary functions that are not mandatory but are included to enhance the performance:

1. To provide an efficient management of RAM.
2. To provide an exclusive access to the computer resources.

The term real time refers to the technique of updating files with the transaction data immediately just after the event that it relates with.

Few more examples of real time processing are:

1. Airlines reservation system.
2. Air traffic control system.
3. Systems that provide immediate updating.
4. Systems that provide up to the minute information on stock prices.
5. Defense application systems like as RADAR.

Real time operating systems mostly use the preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with such policies, such as the time-slice in Round Robin scheduling where each task in the task queue is scheduled up to a maximum time, set by the time-slice parameter, in a round robin manner. Hundred of the priority levels are commonly available for scheduling. Some specific tasks can also be indicated to be non-preemptive.

Features of RTOS

An RTOS must be designed in a way that it should strike a balance between supporting a rich feature set for development and deployment of real time applications and not compromising on the deadlines and predictability.

The following points describe the features of an RTOS (Note that this list is not exhaustive) :

- Context switching latency should be short. This means that the time taken while saving the context of current task and then switching over to another task should be short.
- The time taken between executing the last instruction of an interrupted task and executing the first instruction of interrupt handler should be predictable and short. This is also known as interrupt latency.
- Similarly the time taken between executing the last instruction of the interrupt handler and executing the next task should also be short and predictable. This is also known as interrupt dispatch latency.
- Reliable and time bound inter process mechanisms should be in place for processes to communicate with each other in a timely manner.
- An RTOS should have support for multitasking and task preemption. Preemption means to switch from a currently executing task to a high priority task ready and waiting to be executed.
- Real time Operating systems but support kernel preemption where-in a process in kernel can be preempted by some other process.

A Real time operating system, which were originally used to control autonomous system such satellites, robots and hydroelectric dams. A real time operating system is one that must react to input and responds to them quickly. A real time system cannot afford to be late with a response to an event.

A real time system has well defined, fixed time constraints. Deterministic (Periodic) scheduling algorithms are used in real time system. Real time systems are divided into two groups:

1. Hard real time system
2. Soft real time system

1. A hard real time system guarantees that the critical task be completed on time. This goal requires that all delay in the system be bounded. Soft real time system is a less restrictive type. In this, a critical real time task gets priority over other tasks, and retains that priority until it completes.

Real time operating system used priority scheduling algorithm to meet the response requirement of a real time application.

Real Time Application	Example
Detection	Rader system, Burglar alarm
Process monitoring and control	Petroleum, Paper mill
Communication	Telephone switching system
Flight simulation and control	Auto pilot shuttle mission simulator
Transportation	Traffic light system, Air traffic control

Memory management in real time system is comparatively less demanding than in other types of multiprogramming system. Time-critical device management is one of the main characteristic of real time system. The primary objective of file management in real time system is usually speed of access, rather than efficient utilization of secondary storage.

The ability of the operating system to provide a required level of service in a bounded response time is real time operating system. A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period.

i. The correctness depends not only on the logical result but also the time it was delivered
ii. Failure to respond is as bad as the wrong response.

Examples:

A common example of an RTOS is an HDTV receiver and display. It needs to read a digital signal, decode it and display it as the data comes in. Any delay would be noticeable as jerky or pixilated video and/or garbled audio.

Some of the best known, most widely deployed, real-time operating systems are:

- LynxOS
- OSE
- QNX
- RTLinux
- VxWorks
- Windows CE

Compare RTOSs and general-purpose operating systems

1. Deterministic: Formally deterministic timing means that operating system services consume only known and expected amounts of time. Random elements in service times could cause random delays in application software and could then make the application randomly miss which the real-time system to the deadlines. In real time system the timing of occurrence should be perodicatiable or should be determinate. But general-computing non-real-time operating systems are often quite non-deterministic. Their services can inject random delays into application software and thus cause slow responsiveness of an application at unexpected times.
2. **Task scheduling:** Most RTOSs do their scheduling of tasks using a scheme called "priority-based preemptive scheduling."Each task in a software application must be assigned a priority, with higher priority values representing the need for quicker responsiveness. Very quick responsiveness is made possible by the "preemptive" nature of the task scheduling. "Preemptive" means that the scheduler is allowed to stop any task at any point in its execution, if it determines that another task needs to run immediately. The basic rule that governs priority-based preemptive scheduling is that at every moment in time, "The Highest Priority Task that is Ready to Run, will be the Task that Must be Running." In other words, if both a low-priority task and a higher-priority task are ready to run, the scheduler will allow the higher priority task to run first. The low-priority task will only get to run after the higher-priority task has finished with its current work. What if a low-priority task has already begun to run, and then a higher-priority task becomes ready? This might occur because of an external world trigger such as a switch closing. A priority-based preemptive scheduler will behave as follows: It will allow the low-priority task to complete the current assembly language instruction that it is executing. [But it won't allow it to complete an entire line of high-level language code; nor will it allow it to continue running until the next clock tick.] It will then immediately stop the execution of the low-priority task, and allow the higher-priority task to run. After the higher priority task has finished its current work, the low-priority task will be allowed to continue running. This is shown in Figure 3, where the higher-priority task is called "Mid-Priority Task." Of course, while the mid-priority task is running, an even higher-priority task might become ready. This is represented in Figure 3 by "Trigger_2" causing the "High-Priority Task" to become ready. In that case, the running task ("Mid-Priority Task") would be preempted to allow the high-priority task to run. When the high-priority task has finished its current work, the mid-priority task would be allowed to

continue. And after both the high-priority task and the mid-priority task complete their work, the low-priority task would be allowed to continue running. This situation might be called "nested preemption."

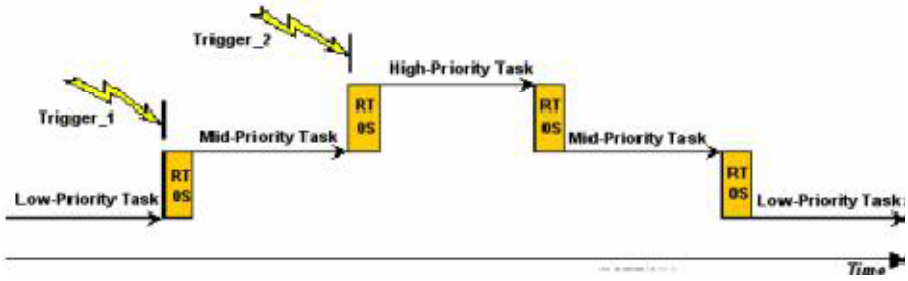


Figure 3: Timeline for Priority-based Preemptive Scheduling Examples

Each time the priority-based preemptive scheduler is alerted by an external world trigger (such as a switch closing) or a software trigger (such as a message arrival), it must go through the following 5 steps:

1. Determine whether the currently running task should continue to run. If not …
2. Determine which task should run next.
3. Save the environment of the task that was stopped (so it can continue later).
4. Set up the running environment of the task that will run next.
5. Allow this task to run.

These 5 steps together are called "task switching."

3. **Fixed-time task switching:** The time it takes to do task switching is of interest when evaluating an operating system. A simple general-computing (non-preemptive) operating system might do task switching only at timer tick times, which might for example be ten milliseconds apart. Then if the need for a task switch arises anywhere within a 10-millisecond timeframe, the actual task switch would occur only at the end of the current 10-millisecond period. Such a delay would be unacceptable in most real-time embedded systems. In more sophisticated preemptive task schedulers, the scheduler may need to search through arrays of tasks to determine which task should be made to run next. If there are more tasks to search through, the search will take longer. Such searches are often done by general-computing operating systems, thus making them non-deterministic. Real-time operating systems, on the other hand, avoid such searches by using incrementally updated tables that allow the task scheduler to identify the task that should run next in a rapid fixed-time fashion. These two types of timing behavior for task switching can be seen in Figure 4.

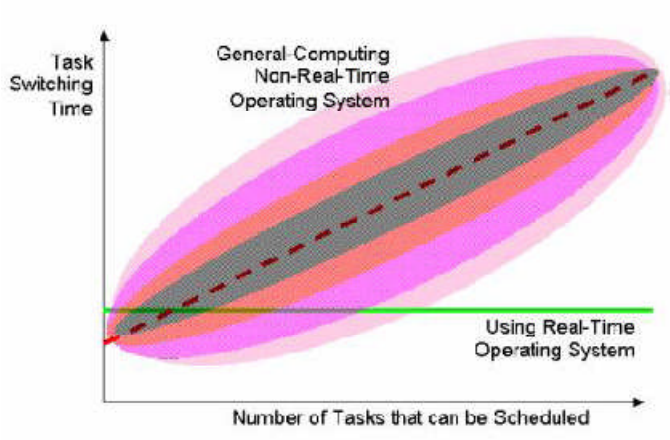


Figure 4: Task Switching Timing

In this figure, we see that for a general-computing (non-real-time) operating system, the task switching time generally rises as a software system includes more tasks that can be scheduled. However, the actual time for a task switch is not the time shown by the dashed red line. Instead, in any given task switch instance, it might be well above or well below the time shown by the dashed red line. The shaded regions surrounding the dashed red line simply show the likelihood of the actual task switch time being that far above or below the dashed red line.

On the other hand, the horizontal solid green line shows the task switching time characteristic of a real time operating system. It is constant, independent of any load factor such as the number of tasks in a software system.

Please note that in some instances, such as the leftmost area of the graph, the task switching time might in special cases be quicker for a general-computing non-real-time operating system, than for a real-time operating system. This does not detract from the appropriateness of a real-time operating system for real time embedded applications. For, in fact, the term "real-time" does not mean "as fast as possible" but rather "real-time" demands consistent, repeatable, known timing performance. Although a non-real-time operating system might do some faster task switching for small numbers of tasks, it might equally well introduce a long time delay the next time it does the same task switch. The strength of a real-time operating system is in its known, repeatable timing performance, which is also typically faster than that of a non-deterministic task scheduler in situations of large numbers of tasks in a software system. Most often, the real-time operating system will exhibit task-switching times much faster than its non-real-time competitor when the number of tasks grows above 5 or 10.

4. **Intertask communication and synchronization:** Most operating systems, including RTOSs, offer a variety of mechanisms for communication and

synchronization between tasks. These mechanisms are necessary in a preemptive environment of many tasks, because without them the tasks might well communicate corrupted information or otherwise interfere with each other. For instance, a task might be preempted when it is in the middle of updating a table of data. If a second task that preempts it reads from that table, it will read a combination of some areas of newly-updated data plus some areas of data that have not yet been updated. These updated and old data areas together may be incorrect in combination, or may not even make sense. An example is a data table containing temperature measurements that begins with the contents "10 C." A task begins updating this table with the new value "99 F", writing into the table character-by-character. If that task is preempted in the middle of the update, a second task that preempts it could possibly read a value like "90 C" or "99 C." or "99 F", depending on precisely when the preemption took place. The partially updated values are clearly incorrect, and are caused by delicate timing coincidences that are very hard to debug or reproduce consistently. An RTOS's mechanisms for communication and synchronization between tasks are provided to avoid these kinds of errors. Most RTOSs provide several mechanisms, with each mechanism optimized for reliably passing a different kind of information from task to task. Probably the most popular kind of communication between tasks in embedded systems is the passing of data from one task to another. Most RTOSs offer a message passing mechanism for doing this, as seen in Figure 5. Each message can contain an array or buffer of data.

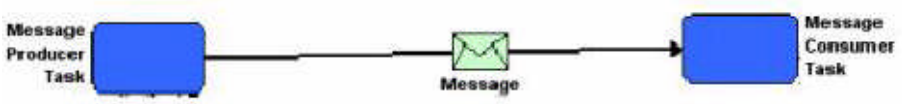


Figure 5: Intertask Message Communication

If messages can be sent more quickly than they can be handled, the RTOS will provide message queues for holding the messages until they can be processed. This is shown in Figure6. Another kind of communication between tasks in embedded systems is the passing of what might be called "synchronization information" from one task to another. "Synchronization information" is like a command, where some commands could be positive, and some negative. For example, a negative command to a task would be something like "Please don't print right now, because my task is using the printer." Or more generally, "I want to lock the . . . for my own use only." A positive command would be something like

"I've detected a cardiac emergency, and I want you to help me handle it." Or more generally, "Please join me in handling . . ."

Most RTOSs offer a semaphore or mutex mechanism for handling negative synchronization (sometimes called "mutual exclusion"). These mechanisms allow tasks to lock certain embedded system resources for their use only, and subsequently to unlock the resource when they're done. For positive synchronization, different RTOSs offer different mechanisms. Some RTOSs offer eventflags, while others offer signals. And yet others rely on message passing for positive synchronization as well as data passing duties.

5. **Determinism and high-speed message passing:** Intertask message communication is another area where different operating systems show different timing characteristics. Most operating systems actually copy messages twice as they transfer them from task to task via a message queue. See Figure 6. The first copying is from the message-sender task to an operating system-owned "secret" area of RAM memory (implementing the "message queue"); and the second copying is from the operating system's "secret" RAM area to the message-receiver task. Clearly this is non-deterministic in its timing, as these copying activities take longer as message length increases.

Figure 6: Message Transfer via Message Queue

An approach that avoids this non-determinism and also accelerates performance is to have the operating system copy a pointer to the message and deliver that pointer to the message-receiver task without moving the message contents at all. In order to avoid access collisions, the operating system then needs to go back to the message-sender task and obliterate its copy of the pointer to the message. For large messages, this eliminates the need for lengthy copying and eliminates non-determinism.

6. **Dynamic memory allocation:** Determinism of service times is also an issue in the area of dynamic allocation of RAM memory. Many general-computing non-real-time operating systems offer memory allocation services from what is termed a "Heap." The famous "malloc" and "free" services known to C-language programmers work from a heap. Tasks can temporarily borrow some memory from the operating system's heap by calling "malloc", and specifying the size of memory buffer needed. When this task (or another task) is finished with this memory buffer it can return the buffer to the operating system by calling "free." The operating system will then return

the buffer to the heap, where its memory might be used again, perhaps as part of a larger buffer. Or perhaps it may in the future be broken into several smaller buffers. Heaps suffer from a phenomenon called "External Memory Fragmentation" that may cause the heap services to degrade. This fragmentation is caused by the fact that when a buffer is returned to the heap, it may in the future be broken into smaller buffers when "malloc" requests for smaller buffer sizes occur. After a heap undergoes many cycles of "malloc"s and "free"s, small slivers of memory may appear between memory buffers that are being used by tasks. These slivers are so small that they are useless to tasks. But they are trapped between buffers that are being used by tasks, so they can't be coagulated ("glued") together into bigger, useful buffer sizes. Over time, a heap will have more and more of these slivers. This will eventually result in situations where tasks will ask for memory buffers ("malloc") of a certain size, and they will be refused by the operating system --- even though the operating system has enough available memory in its heap. The problem: That memory is scattered in small slivers distributed in various separate parts of the heap. In operating system terminology, the slivers are called "fragments", and this problem is called "external memory fragmentation." This fragmentation problem can be solved by so-called "garbage collection" (defragmentation) software. Unfortunately, "garbage collection" algorithms are often wildly non-deterministic – injecting randomly appearing random-duration delays into heap services. These are often seen in the memory allocation services of general-computing non-real-time operating systems. This puts the embedded system developer who wants to use a general-computing non-real-time operating system into a quandry: Should the embedded system be allowed to suffer occasional randomly-appearing random-duration delays if / when "garbage collection" kicks in?... Or, alternatively, should the embedded system be allowed to fragment its memory until application software "malloc" requests to the heap are refused even though a sufficient total amount of free memory is still available? Neither alternative is acceptable for embedded systems that need to provide service continually for long periods of time. Real-time operating systems, on the other hand, solve this quandry by altogether avoiding both memory fragmentation and "garbage collection", and their consequences. RTOSs offer non-fragmenting memory allocation techniques instead of heaps. They do this by limiting the variety of memory chunk sizes they make available to application software. While this approach is less flexible than the approach taken by memory heaps, they do avoid external memory fragmentation and

avoid the need for defragmentation. For example, the "Pools" memory allocation mechanism allows application software to allocate chunks of memory of perhaps 4 or 8 different buffer sizes per pool. Pools totally avoid external memory fragmentation, by not permitting a buffer that is returned to the pool to be broken into smaller buffers in the future. Instead, when a buffer is returned the pool, it is put onto a "free buffer list" of buffers of its own size that are available for future re-use at their original buffer size. This is shown in Figure 7.

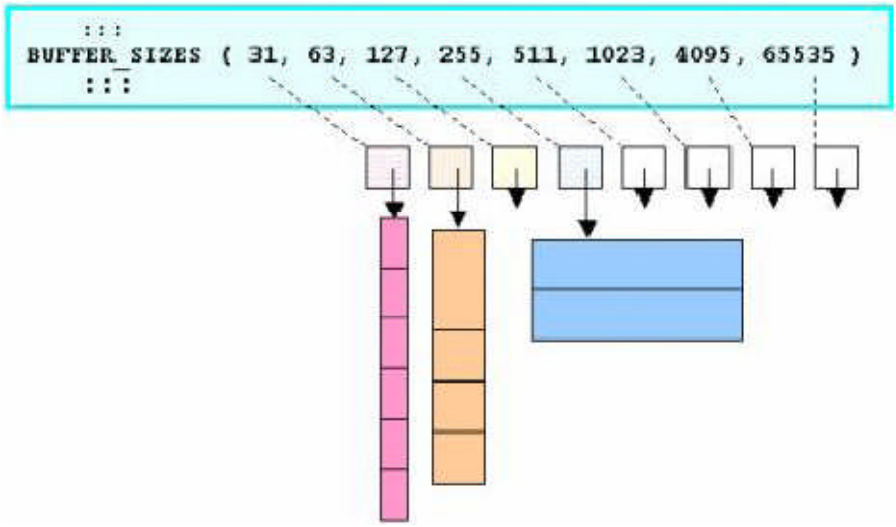


Figure 7: A Memory Pool's Free Buffer Lists

Memory is allocated and de-allocated from a pool with deterministic, often constant, timing.

Hard real-time - Systems where it is absolutely imperative that responses occur within the required deadline. E.g. Flight control systems.

Soft real-time - Systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Data acquisition system.

Real real-time - Systems which are hard real-time and which the response times are very short. E.g. Missile guidance system.

Firm real-time - Systems which are soft real-time but in which there is no benefit from late delivery of service.

6.1.1 HARD REAL-TIME SYSTEMS

Definition: A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure. Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems secondary storage is limited or missing with data stored in ROM. In these systems virtual memory is almost never found. Hard real time system is purely deterministic and time constraint system for example users

expected the output for the given input in 10sec then system should process the input data and give the output exactly by 10^{th} second. Here in the above example 10 sec. is the deadline to complete process for given data. Hard real systems should complete the process and give the output by 10^{th} second. It should not give the output by 11^{th} second or by 9^{th} second, exactly by 10^{th} second it should give the output. In the hard real time system meeting the deadline is very important if deadline is not met the system performance will fail.

Another example is defense system if a country launched a missile to another country the missile system should reach the destiny at 4:00 to touch the ground what if missile is launched at correct time but it reached the destination ground by 4:05 because of performance of the system, with 5 minutes of difference destination is changed from one place to another place or even to another country. Here system should meet the deadline.

SOFT REAL-TIME SYSTEMS

Definition: A Soft Real-Time System Is One In Which Performance Is Degraded But Not Destroyed By Failure To Meet Response-Time Constraints. Soft real time systems are less restrictive. Critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, Multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers etc …In soft real time system, the meeting of deadline is not compulsory for every time for every task but process should get processed and give the result. Even the soft real time systems cannot miss the deadline for every task or process according to the priority it should meet the deadline or can miss the deadline. If system is missing the deadline for every time the performance of the system will be worse and cannot be used by the users.

Examples:

- multimedia transmission and reception,
- networking, telecom (cellular) networks,
- web sites and services
- Computer games.

6.2.3 FIRM REAL TIME

Definition: A firm real-time system is one in which a few missed deadlines will not lead to total failure, but missing more than a few may lead to complete and catastrophic system failure. Firm real time systems can miss some deadlines, but eventually performance will

degrade if too many are missed. A good example is the sound system in your computer. If you miss a few bits, no big deal, but miss too many and you're going to eventually degrade the system. For a Firm real-time system**,** even if the system fails to meet the deadline, possibly more than once (i.e. for multiple requests), the system is not considered to have failed. Also, the responses for the requests (replies to a query, result of a task, etc.) are worthless once the deadline for that particular request has passed (The usefulness of a result is zero after its deadline). A hypothetical example can be a storm forecast system (if a storm is predicted before arrival, then the system has done its job, prediction after the event has already happened or when it is happening is of no value).

6.2.4 REALREAL TIME

(Combination of all Types)

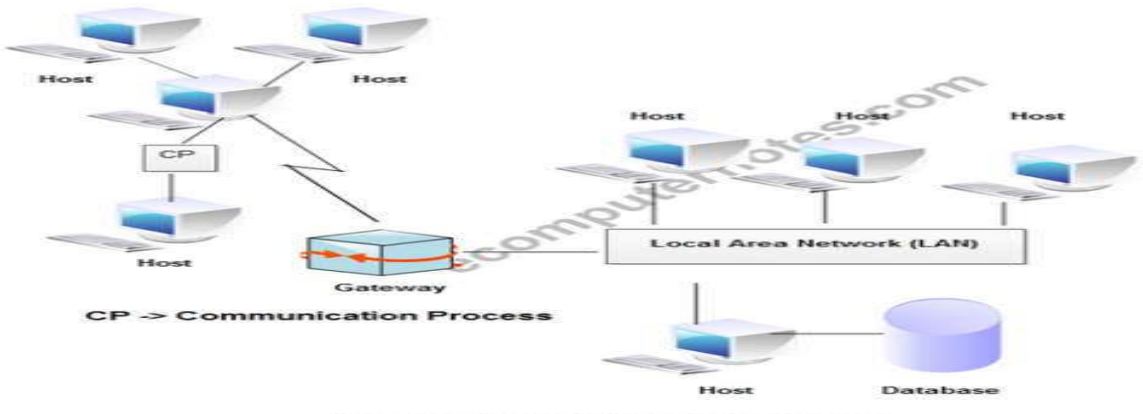
Chapter 7: Distributed Operating System (3 hrs)

7.1 Introduction

Definition of distributed system

A distributed system is a collection of independent computers that appear to the users of the system as a single computer. A distributed system consists of a collection of autonomous computers linked to a computer network and equipped with distributed system software. A distributed system is a collection of processors that do not share memory or a clock. Distributed systems are a term used to define a wide range of computer systems from a weakly-coupled system such as wide area networks, to very strongly coupled systems such as multi-processor systems. A distributed system is a set of computers that communicate each other using software and hardware interconnecting components. Multiprocessors (MIMD computers using shared memory architecture), multi-computers connected through static or dynamic interconnection networks (MIMD computers using message passing architecture) and workstations connected through local area network are examples of such distributed systems.

Distributed systems are potentially more reliable than a central system because if a system has only one instance of some critical component, such as a CPU, disk, or network interface, and that component fails, the system will go down. When there are multiple instances, the system may be able to continue in spite of occasional failures. In addition to hardware failures, one can also consider software failures. Distributed systems allow both hardware and software errors to be dealt with. This system looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs).



A Typical View of Distributed System

Distributed operating system

Distributed operating system is a model where distributed applications are running on multiple computer linked by communication. Distributed operating system is an extension of the network operating system that supports higher level of communication and integration of the machine on the network. These systems are referred as loosely coupled system where each processor has it own local memory and processors communicate with one another through various communication lines, as high speed buses or telephone lines.

A distributed system is managed by a distributed operating system. A distributed operating system manages the system shared resources used by multiple processes, the process scheduling activity (how processes are allocating on available processors), the communication and synchronization between running processes and so on. The software for parallel computers could be also tightly coupled or loosely coupled. The loosely coupled software allows computers and users of a distributed system to be independent each other but having a limited possibility to cooperate. An example of such a system is a group of computers connected through a local network. Every computer has its own memory, hard disk. There are some shared resources such files and printers.

The Distributed Os involves a collection of autonomous computer systems, capable of communicating and cooperating with each other through a LAN / WAN. A Distributed Os provides a virtual machine abstraction to its users and wide sharing of resources like as computational capacity, I/O and files etc. The structure shown in fig contains a set of individual computer systems and workstations connected via communication systems, but by this structure we cannot say it is a distributed system because it is the software, not the hardware, that determines whether a system is distributed or not.

The users of a true distributed system should not know, on which machine their programs are running and where their files are stored. LOCUS and MICROS are the best examples of distributed operating systems.

Using LOCUS operating system it was possible to access local and distant files in uniform manner. This feature enabled a user to log on any node of the network and to utilize the resources in a network without the reference of his/her location. MICROS provided sharing of resources in an automatic manner. The jobs were assigned to different nodes of the whole system to balance the load on different nodes.

Advantages and Disadvantages

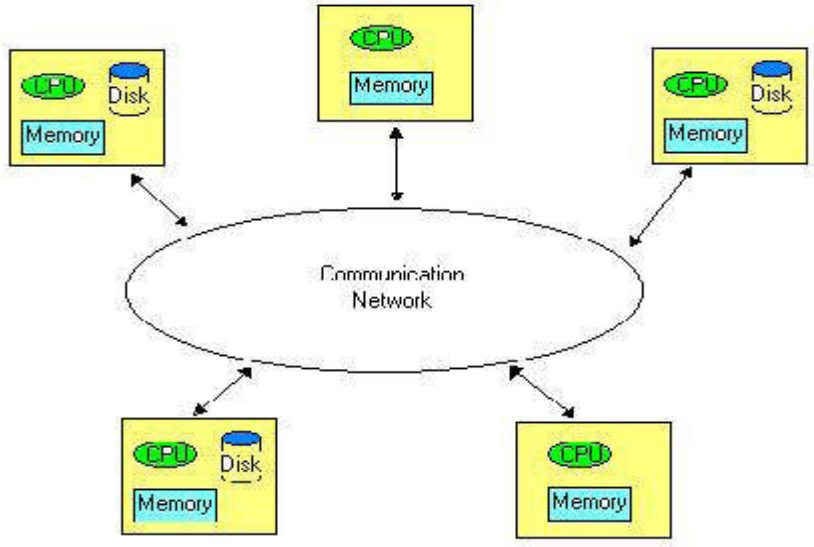
• Advantages

- Communication and resource sharing possible
- Economics – price-performance ratio
- Reliability, scalability
- Potential for incremental growth

• Disadvantages

- Distribution-aware OSs and applications
- Network connectivity essential
- Security and privacy

7.2 Communication and Synchronization



Synchronization

Issues in synchronization

1. How do processes cooperate and synchronizes with each other?
    - Each computer's internal clock can be different
    - No shared memory to synchronize on semaphores
    - Delay in data communications
    - Detecting computer/process crashes
2. How are critical regions implemented?
    - When the processes are located on different computers
3. How are resources allocated?
    - When processes on different computers require the same resource
    - Deadlock

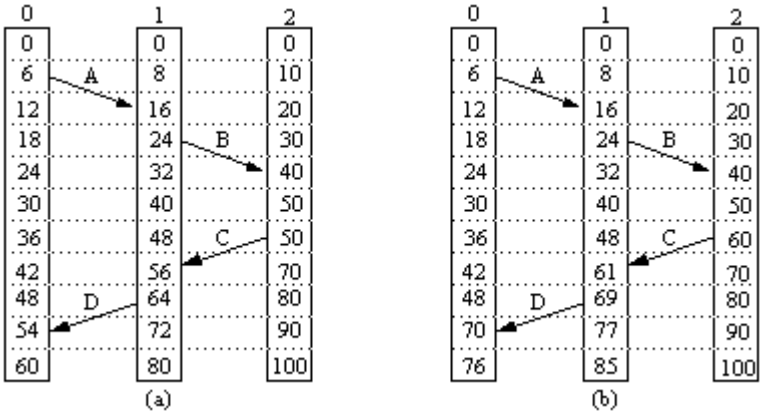
Clock Synchronizations

Synchronizing processes in a distributed environment is difficult because:

- Relevant information can be anywhere on the system
- Processes can make decisions based only on local information
- Want to avoid having a centralized coordinator
- No shared system clock

Logical Clocks

- Not always necessary for each computer to know the exact time
- Sufficient for computers to know the order in which events occur
- Computers can update their logical clocks based on when events occurred on cooperating computers



- Additionally, between two events the clock must tick at least once

Physical Clocks

- Some processes are dependent on real time.
- The problems are to synchronize physical clocks with real-world clocks and to synchronize physical clocks
- Universal Coordinated Time: Atomic Clocks are used to keep accurate track of time. The National Institute of Standard Time broadcasts using the call sign WWV on SW Radio and computers can receive this (expensive)

Synchronizing Physical Clocks

- Time server connected to WWV receiver
- Computers ask the time server for the current time, and reset their clocks
- Problems

- Time is not allowed to run backwards - client must "slow down" time if its clock is running fast.
- The time taken for the server to send the response and the client to receive it (and generate an interrupt, etc.) takes... a variable amount of time!
- Client can ask server for time repeatedly and calculate average delay and take this into account when resetting its clock

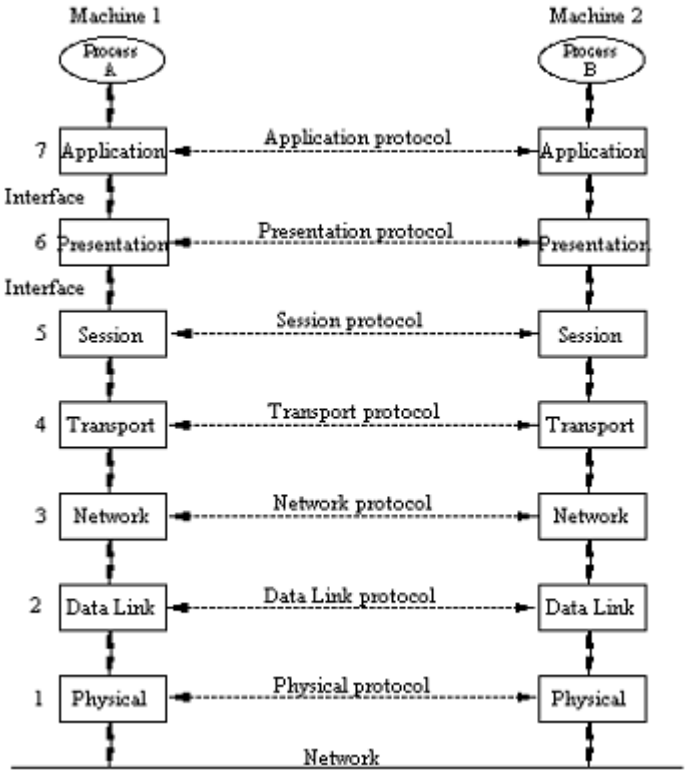
COMMUNICATION IN DISTRIBUTED OPERATING SYSTEMS

Issues

- Processes which execute on CPUs sharing memory can easily communicate, but what about processes executing on CPUs which don't share memory?
- How to implement message passing (communication)
- Speed of communication
- Reliability of communication
- Transparency of communication
- Locating the right process to communicate with
- Consistency of Communication

OSI in a nutshell

- Defines a standard for open systems to communicate
- 2 Generic Types
  - Connection-oriented
  - Connectionless
- OSI generally gives users/applications the impression that connection-oriented communication is provided
- What happens in lower layers in the model is hidden from users/applications



7. The Application Layer

- High-level application protocols, e.g., e-mail, video conferencing, file transfer, etc.

6. The Presentation Layer

- Concerned with the meaning of bits in the message
- Notifies receiver that message contains a particular record in a certain format

5. The Session Layer

- Provides dialog control and synchronization facilities
- Check pointing can be used so that after recovering from a crash transmission can resume from the point just before the crash
- Rarely used

4. The Transport Layer

- Provides a mechanism to assure the Session Layer that messages sent are all received without any data corruption or loss

- Breaks message from Session Layer into appropriate chunks (e.g., IP Packets), numbers them and sends them all
- Communicates with receiver to ensure that all have been received, how many more the receiver can receive, etc.

3. The Network Layer

- Determines route (next hop) message will take to bring it closer to its destination

2. The Data Link Layer

- Detects and corrects data transmission errors (data corruption, missing data, etc.)
- Gathers bits into frames and ensures that each frame is received correctly
- Sender puts special bit pattern at the start and end of each frame + a checksum + a frame number

1. The Physical Layer

- Concerned with Transmitting Bits
- Standardizes the electrical, mechanical and signaling interfaces

There are different way of communicating **way**

1. **Asynchronous Transfer Mode (ATM) Networks in a nutshell**

- Advances in network technology introduce high-speed networks capable of transferring multimedia data at rates of 155 Mbps and up, using either point-to-point and/or broadcast communications
- Can now implement high-bandwidth distributed systems
- ATM grew out of a need for a single network capable of carrying data in low constant data rates (e.g., voice) and burst data traffic
- ATM carries fixed-sized blocks (cells) over virtual (connection-oriented) circuits
- When two or more computers need to communicate, the sender first establishes a connection. The network helps to determine the route and then future conversations are carried over the same route, until the connection is dropped
- The routing information is stored in switches. When the connection is dropped, the routing information in the switches is purged

2. The Client-Server Model in a nutshell

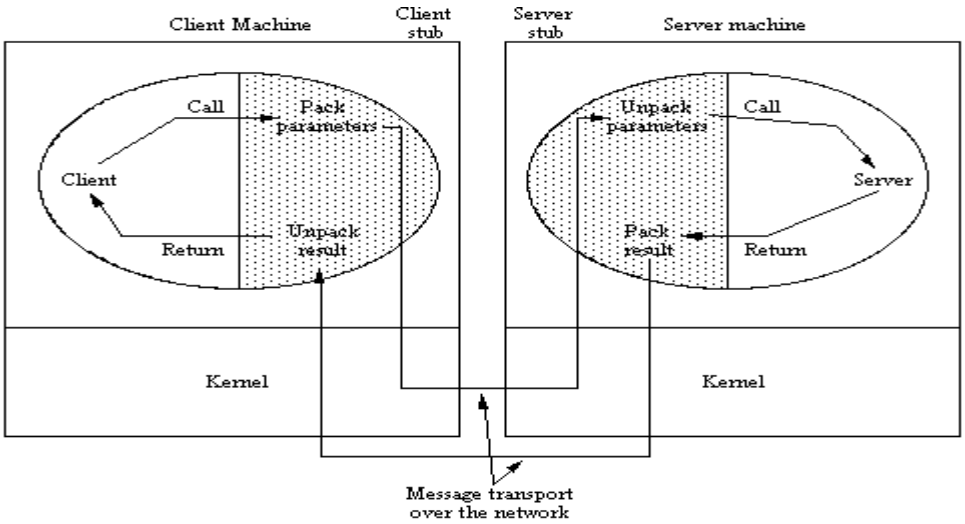
- The OS is structured as a group of cooperating processes (servers) that offer services to users (Clients)
- Clients and Servers run the same microkernel
- Connectionless request/reply protocol used
- Client requests a service from a server using *send(dest, &mptr).* Server waits for a message to be recieved using *receive(addr, &mptr),* performs the task and sends the data or error code
- No overheads of setting up/dropping a connection, passing messages through multiple layers, etc.
- A name server is used for clients to determine the machine running the process with which they want to communicate

3. Remote Procedure Call in a nutshell

- Programs on one computer are allowed to call procedures located on other computers.
- Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details.
- When program statements that use RPC are compiled into an executable program, a stub is included in the compiled code that acts as the representative of the remote procedure code. When the program is run and the procedure call is issued, the stub receives the request and forwards it to a client runtime program in the local computer. The client runtime program has the knowledge of how to address the remote computer and server application and sends the message across the network that requests the remote procedure. Similarly, the server includes a runtime program and stub that interface with the remote procedure itself. Results are returned the same way.
- Identical to one program calling another on the same machine (with parameters, return results, etc.)
- Problems...
    - Processes don't share memory
    - Locating the machine running the procedure
    - Machines involved can use different micro-kernels
    - Reference parameters

- One of the machines might crash

Example



- On machine 1, program rand is replaced by a **client stub**
- Machine 2 has both a **server stub** for rand as well as the rand procedure
- Machine 1 doesn't realize that rand is located on a different machine
- rand on machine 2 doesn't realize that it has been called by a procedure on a remote machine
- The client stub packs the parameters (including the procedure name) into a message (parameter marshalling) and passes the message to machine 2. The client then waits to receive the result
- The server stub for rand will have been waiting to receive a message. When one arrives, it unpacks the parameters and calls the rand procedure. The procedure returns the result to the server stub which sends the new message back to the client. When the client stub for rand receives the message, it unblocks, unpacks the message and passes the result to the calling procedure

Dealing with Failures

- Several types of failure
  - Client unable to locate server
  - Request message from client to server is lost
  - Reply message from server to client is lost

- Server crashes after receiving request
- Client crashes after sending request

7.3 Process and Processor in Distributed OS

- In most traditional OS, each process has an address space and a single thread of control.
- It is desirable to have multiple threads of control sharing one address space but running in quasi-parallel.

Processor Allocation

- Processor allocation determines which process is assigned to which processor it is also called load distribution.
- Two categories:
  - Static load distribution-non-migratory, once allocated, cannot move, no matter how overloaded the machine is.
  - Dynamic load distribution-migratory, can move even if the execution started. But algorithm is complex
- The goals of allocation
  - Maximize CPU utilization
  - Minimize mean response time/ Minimize response ratio

Chapter 8: File Systems (3 hrs)

All computer applications need to store and retrieve information. There are many problem occurred during storing information.

First problem while a process is running, it can store a limited amount of information within its own address space. However the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small. A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications, (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process. A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it.

The way to solve this problem is to make the information itself independent of any one process. Thus we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called **files.** Processes can then read them and write new ones if need be. Information stored in files must be **persistent,** that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the **file system**

8.1 Files and Directories

A file is a collection of similar records. The file is treated as a single entity by users and applications and may be referred by name. Files have unique file names and may be created

and deleted. A file is a container for a collection of information. The file manager provides a protection mechanism to allow user's administrator how processes executing on behalf of different users can access the information in a file. File protection is a fundamental property of files because it allows different people to store their information on a shared computer.

File Naming

When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name. The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names.

File Structure

Files can be structured in any of several ways. Three common possibilities are:

1. Byte sequence
2. Record Sequence
3. Tree

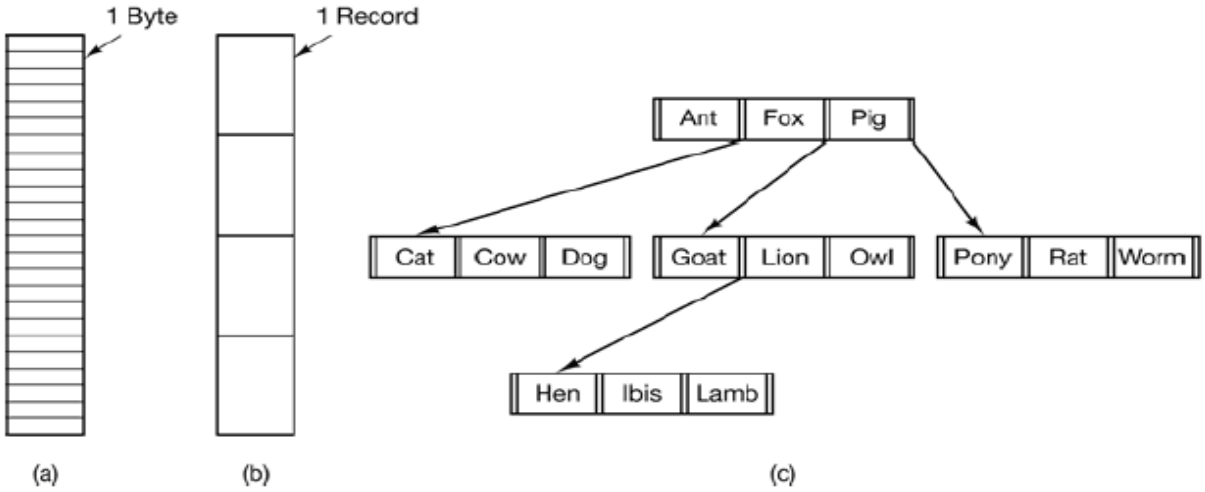


Figure: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

1. Byte sequence: The file in byte sequence is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs.
2. Record sequence: A file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one

record. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system works this way.

3. Tree: In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key. The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. New records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and Windows but is widely used on the large mainframe computers still used in some commercial data processing

File Types

Many operating systems support several types of files.

1. **Regular files** are the ones that contain user information. Directories are system files for maintaining the structure of the file system. Regular files are generally either ASCII files or binary files.
    a. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., MS-DOS) use both. Lines need not to be of the same length. The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines.
    b. Binary files are not ASCII files. Listing them on the printer gives an incomprehensible listing full of what is apparently random junk. Usually, they have some internal structure known to programs that use them.

2. **Character special files** are related to input/output and used to model serial I/O devices such as terminals, printers, and networks.
3. **Block special files** are used to model disks.

File Access

There are two types of file access. They are

1. Sequential access
2. Random access

1. Sequential access: Early operating systems provided only one kind of file access i.e. **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed.
2. Random access: When disks came into use for storing files, it became possible to read the bytes or records of a file out of order or to access records by key, rather than by position. Files whose bytes or records can be read in any order are called **random access files**. Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size, these extra items the file's attributes. The list of attributes varies considerably from system to system.

File attributes vary from one operating system to another. The common attributes are:

- **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

File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval

1. Create: The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete: When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open: Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. Close: When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
5. Read: Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.
6. Write: Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. Append: This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
8. Seek: For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position,

9. Get attributes: Processes often need to read file attributes to do their work.
10. Set attributes: Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.
11. Rename: It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

Directories

To keep track of files, file systems normally have **directories** or **folders,** which, in many systems, are themselves files.

Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory,** but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. An example of a system with one directory is given in Figure. Here the directory contains four files. The file *owners* are shown in the figure, not the file *names* (because the owners are important to the point we are about to make). The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all.

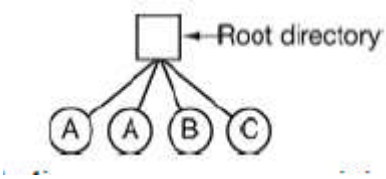


Figure: A single-level directory system containing four files, owned by three different people, *A* , *B* , and *C* .

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user *A* creates a file called *mailbox,* and then later user *B* also creates a file called *mailbox, B*'s file will overwrite *A's* file. Consequently, this scheme is not used on multiuser systems any more, but could be

used on a small embedded system, for example, a system in a car that was designed to store user profiles for a small number of drivers.

Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design could be used, for example, on a multiuser computer or on a simple network of personal computers that shared a common file server over a local area network.

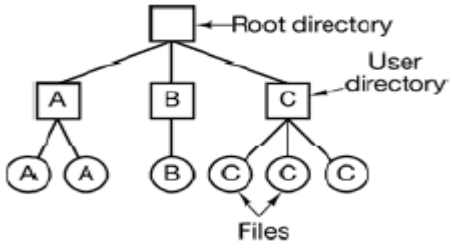


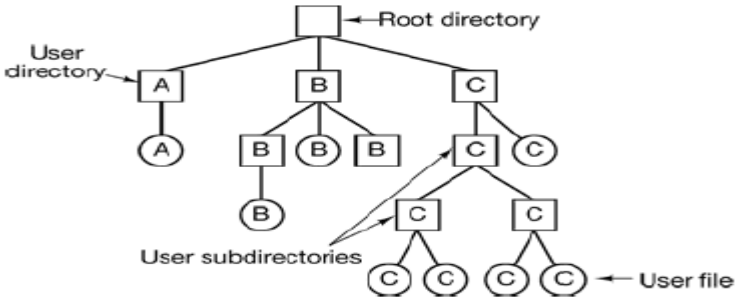
Figure: A two-level directory system. The letters indicate the owners of the directories and files.

Implicit in this design is that when a user tries to open a file, the system knows which user it is in order to know which directory to search. As a consequence, some kind of login procedure is needed, in which the user specifies a login name or identification, something not required with a single-level directory system.

Hierarchical Directory Systems

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. Even on a single-user personal computer, it is inconvenient. It is quite common for users to want to group their files together in logical ways. A professor for example, might have a collection of files that together form a book that he is writing for one course, a second collection of files containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Figure. Here, the directories *A, B,* and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.



A hierarchical directory system

Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used.

In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the root directory contains a subdirectory *usr* , which in turn contains a subdirectory *ast* , which contains the file *mailbox* . Absolute path names always start at the root directory and are unique.

The other kind of name is the **relative path name.** This is used in conjunction with the concept of the **working directory** (also called the **current directory)**. A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast* , then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox* .

Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files.

1. Create: A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).
2. Delete: A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.

3. Opendir: Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. Closedir : When a directory has been read, it should be closed to free up internal table space.
5. Readdir: This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures are being used.
6. Rename: In many respects, directories are just like files and can be renamed the same way files can be.
7. Link: Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link.**
8. Unlink: A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed.

8.2 File System Implementation

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems.

1. Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Notice that with this ordering assuming that only one job is accessing the disk, accessing block b + 1 after block b normally requires no head movement. When head movement is needed, it is only one track. Thus, the

number of disk seeks required for accessing contiguously allocated files is minimal. Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long, and starts at location!), then it occupies blocks b, b + 1, b + 2,……, b + n – 1. The directory entry for each file indicates the address of the starting block and the length of the area allocate for this file. Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b + i.

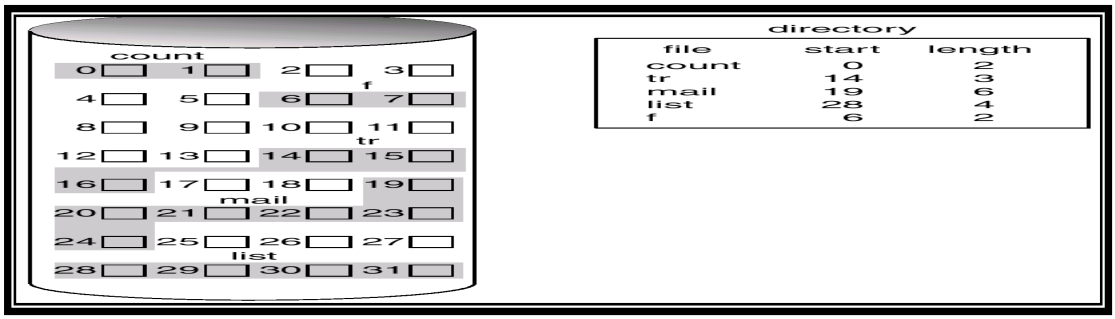


Figure: contiguous allocation

In the above figure there are different types of file implemented in the disk. While implementing file in the contiguous allocation only the start of file and length of the file is necessary. As shown in above figure file name count has started at 0 position and 2 is the length of file and after that there is another file which start from 6th position and length of 2 so it occur in the 6th position of the block freeing the block before 6th position.

Advantage

1. It is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.
2. The read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Disadvantage

1. These algorithms suffer from the problem of external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be either a minor or a major problem.

2. To prevent loss of significant amounts of disk space to external fragmentation, the user had to run repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole. These schemes effectively compress all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time.

3. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this down time, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

4. A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. The user will normally over estimate the amount of space needed, resulting in considerable wasted space.

2. **Linked list allocation**

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Figure. The first word of each block is used as a pointer to the next one. The rest of the block is for data. Linked allocation solves all problems of contiguous allocation. With link allocation, each file is a linked list disk blocks; the disk blocks may be scattered anywhere on the disk.

This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free bio to be found via the free-space management system, and this new block is the written to, and is linked to the end of the file

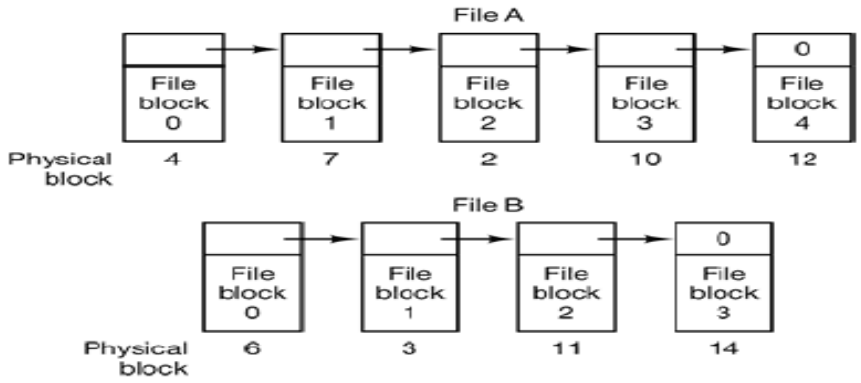


Figure: Linked list allocation

Advantage

1. No space lost to external fragmentation
2. Disk only needs to maintain first block of each file

Disadvantage

1. Random access is slow: Although reading a file sequentially is straight forward, random access is extremely slow. To get to block *n* , the operating system has to start at the beginning and read the n – 1 blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.
2. Overheads of pointers: The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying. If a pointer requires 4 bytes out of a 512 Byte block then 0.78 percent of the disk is being used for pointer, rather than for information.
3. Reliability: Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer— were lost or damaged. Partial solutions are to use doubly linked lists or to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

3. Linked list using index

Disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure shows what the table looks like for the example given in the linked list. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., –1) that is not a valid block number. Such a table in main memory is called a **FAT** (**File Allocation Table**).

Note:
FAT is simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each-partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value -as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-offline value. An illustrative example is the FAT structure of for a file consisting of disk blocks 217, 618, and 339.

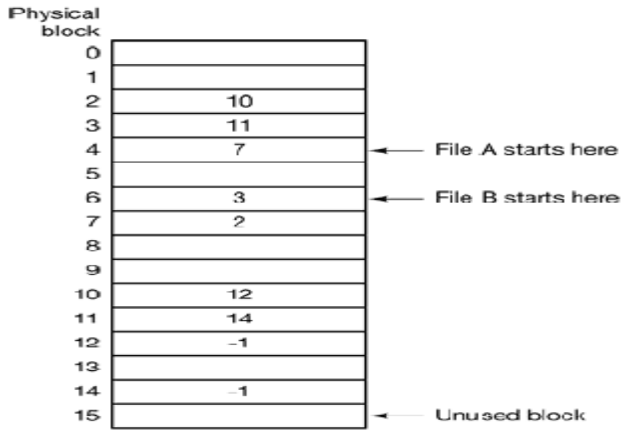


Figure: Linked list allocation using file allocation table in main memory

Advantage:

1. Using this organization, the entire block is available for data.

2. Furthermore, random access is much easier.
3. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

Disadvantage:

1. The indexed allocation suffers from wasted space. The index block may not be fully used (i.e., internal fragmentation).
2. The number of entries of an index table determines the size of a file. To overcome this problem, we can
    a. Have multiple index blocks and chain them into a linked-list ™
    b. Have multiple index blocks, but make them a tree just like the indexed access method ™
    c. A combination of both

4. I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the files blocks. A simple example is depicted in Figure. Given the i-node, it is then possible to find all the blocks of the file.

Advantage:

The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies *n* bytes and a maximum of *k* files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only *kn* bytes. Only this much space need be reserved in advance.

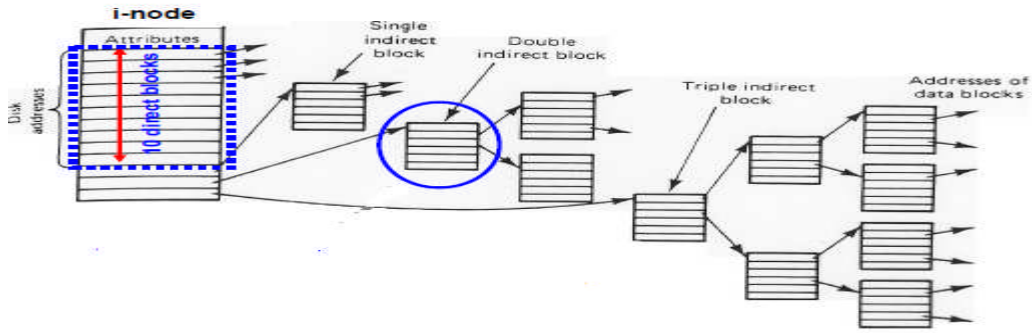


Figure: I-nodes file implementing

8.3 File Sharing and Locking

File sharing

File sharing is the practice of sharing or offering access to digital information or resources, including documents, multimedia (audio/video), graphics, computer programs, images and e-books. File sharing is the public or private sharing of computer data or space in a network with various levels of access privilege. The term file sharing almost always means sharing files in a network, even if in a small local area network. Files can easily be shared outside a network (for example, simply by handing or mailing someone your file on a diskette).

File sharing allows a number of people to use the same file or file by some combination of being able to read or view it, write to or modify it, copy it, or print it. Typically, a file sharing system has one or more administrators. Users may all have the same or may have different levels of access privilege. File sharing can also mean having an allocated amount of personal file storage in a common file system.

More usually, however, file sharing implies a system in which users write to as well as read files or in which users are allotted some amount of space for personal files on a common server, giving access to other users as they see fit. File sharing can be viewed as part of file systems and their management.

File locking

File locking is a mechanism that restricts access to a computer file by allowing only one user or process access at any specific time. Systems implement locking to prevent the

classic *interceding update* scenario, which is a typical example of race condition, by enforcing the serialization of update processes to any given file. The following example illustrates the interceding update problem:

1. Process A reads a customer record from a file containing account information, including the customer's account balance and phone number.
2. Process B now reads the same record from the same file so it has its own copy.
3. Process A changes the account balance in its copy of the customer record and writes the record back to the file.
4. Process B, which still has the original *stale* value for the account balance in its copy of the customer record, updates the account balance and writes the customer record back to the file.
5. Process B has now written its stale account-balance value to the file, causing the changes made by process A to be lost.

Most operating systems support the concept of record locking, which means that individual records within any given file may be locked, thereby increasing the number of concurrent update processes. Database maintenance uses file locking, whereby it can serialize access to the entire physical file underlying a database. Although this does prevent any other process from accessing the file, it can be more efficient than individually locking a large number of regions in the file by removing the overhead of acquiring and releasing each lock.

Poor use of file locks, like any computer lock, can result in poor performance or in deadlocks. File locking may also refer to additional security applied by a computer user either by using Windows security, NTFS permissions or by installing a third party file locking software.

So that with the help of File sharing many users can perform the operations on the Single File. But there must be Some Mechanism which is used for Controlling the Access onto the Single File. File Locking is used when a Single User is performing operations on. The file and other users can’t be able to change the contents of the File.

So that there are three types of Lock:-

1) **Read Only:** This Lock is used when all the users are Reading the Contents from the Single File and no one can be able to Change the contents on the File.

2) Linked shared: The Linked Shared Locked is used when there are many users working on a Single file. And when the various users are requesting to change the contents of the File then the Changing's will perform on the File by using some **Linear Order.**

3) **Exclusive Lock:** This is very important Lock and in this when the multiple users are working on the Single File. Then the File will be locked when any user is going to Change the data of the file, the changing's will be made by only the Single user and for the other users will be locked.