Table of Contents

Chapter 1: Principle of Operating System

Introduction, Operations System Concepts: Processes, files, shell, system calls, security and Operating System structure: Monolithic systems, Layered, Virtual Machines, Client – Server and Evolution of Operating Systems: User driven, operator driven, simple batch system, off – line batch system, directly coupled off – line system, multi- programmed spooling system, on-line timesharing system, multiprocessor systems, multi-computer/ distributed systems, Real time Operating Systems

Operating System Introduction:

Computer Software can roughly be divided into two types:

a). Application Software: Which perform the actual work the user wants.

b). System Software: Which manage the operation of the computer itself.

The most fundamental system program is the operating system, whose job is to control all the computer's resources and provide a base upon which the application program can be written. Operating system acts as an intermediary between a user of a computer and the computer hardware.

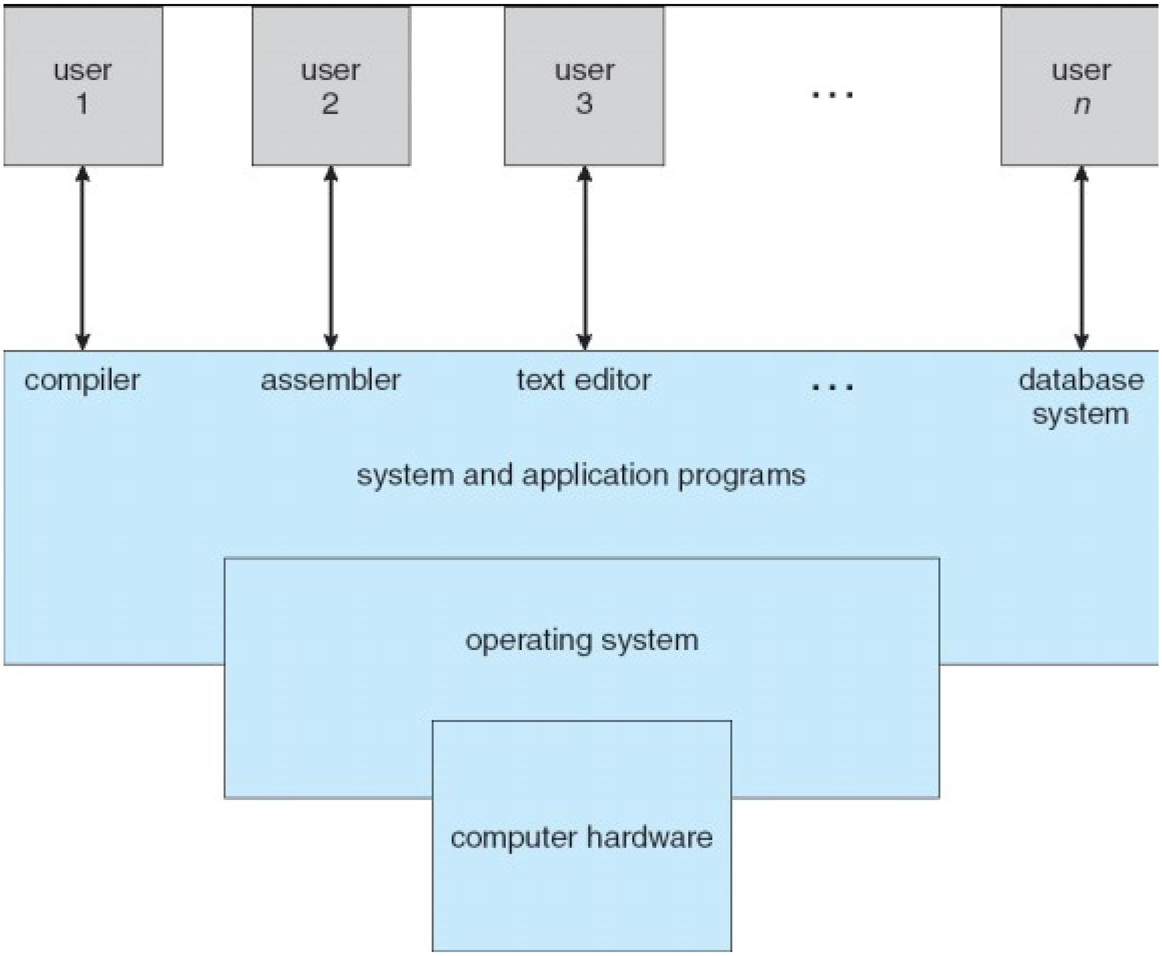


Fig 1.1 Abstract view of the components of a Computer System.

A computer system can be divided roughly into four components: *the hardware, the operating system, the application program, and the users* as shown in the fig 1.1

An operating system is similar to a **government.** Like a government it performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

Two views of the Operating System:

Operating System as an Extended Machine or Virtual Machine(or As a User/computer interface)

The operating system masks or hides the details of the Hardware form the programmers and general users and provides a convenient interface for using the system. The program that hides the truth about the hardware from the user and presents a nice simple view of named files that can be read and written is of course the operating system. In this view the function of OS is to present the user with the equivalent of an extended machine or virtual machine that is easier to program than underlying hardware. Just as the operating system shields the user from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers,memory management and other low level features.
The placement of OS is as shown in fig1.2 A major function of OS is to hide all the complexity presented by the underlying hardware and gives the programmer a more convenient set of instructions to work with.

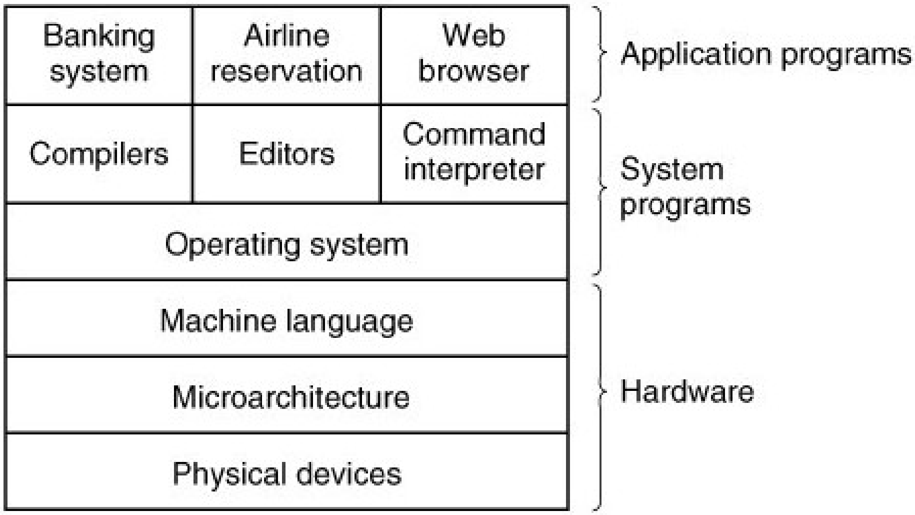


Fig1.2: Computer system consists of Hardware, system program and application program

Operating System as a Resource Manager

A computer system has many resources. 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 I/O devices among the various programs competing for them.
Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next

few from program 2, then some from program 3, and so forth. The result would be chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored to the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

Figure 1.3 suggests the main resources that are managed by the OS. A portion of the OS is in main memory. This includes Kernel or nucleus. The remainder of main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the OS and memory management hardware in the processor

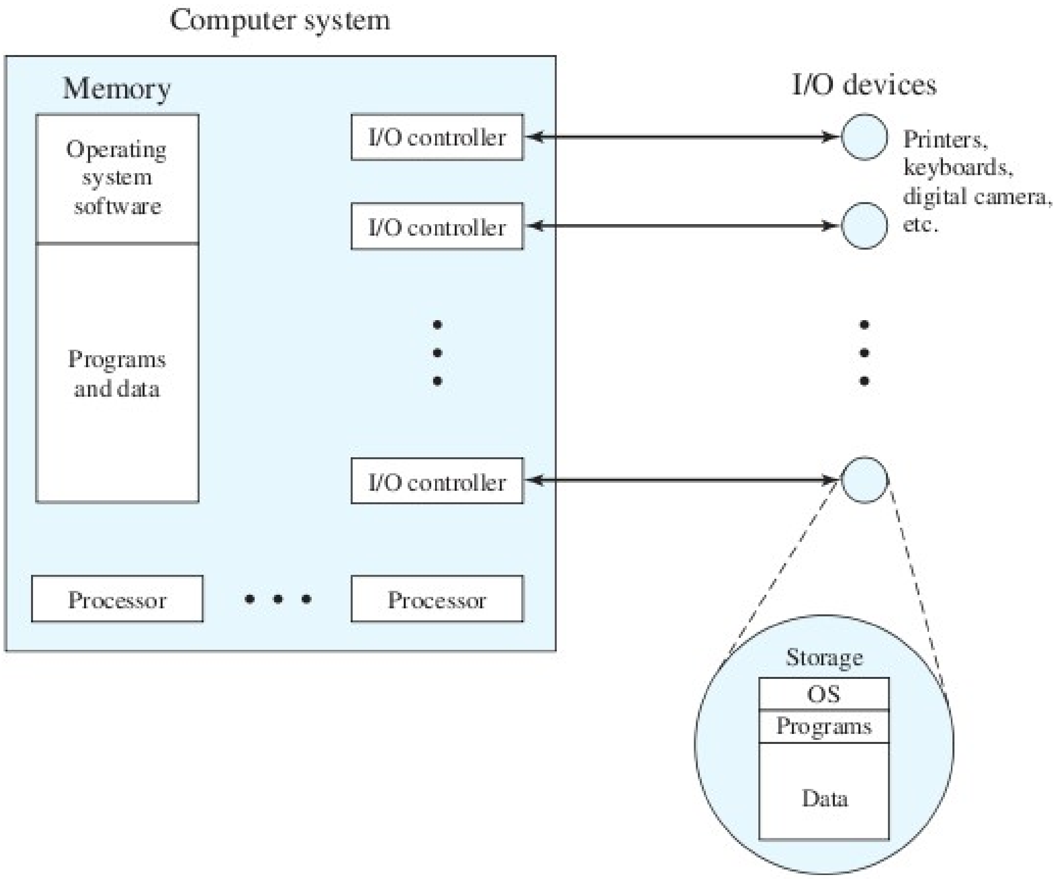


Fig1.3: The Operating system as Resource manger

computer System organization:

A modern general purpose computer system consists of one or more cpus and a number of device controllers connected through a common bus that provides access to shared memory

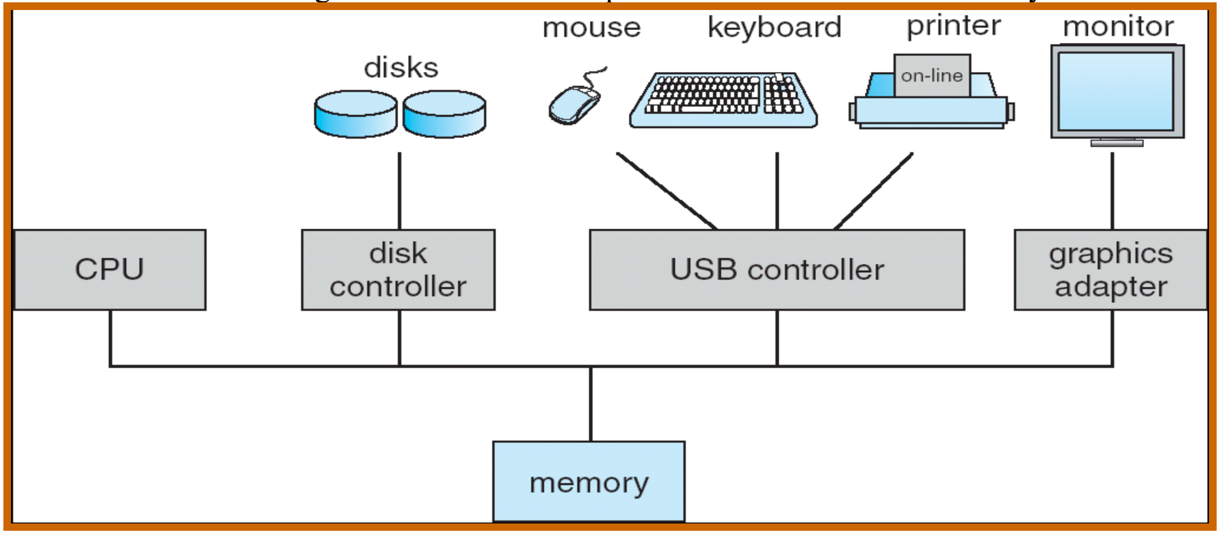
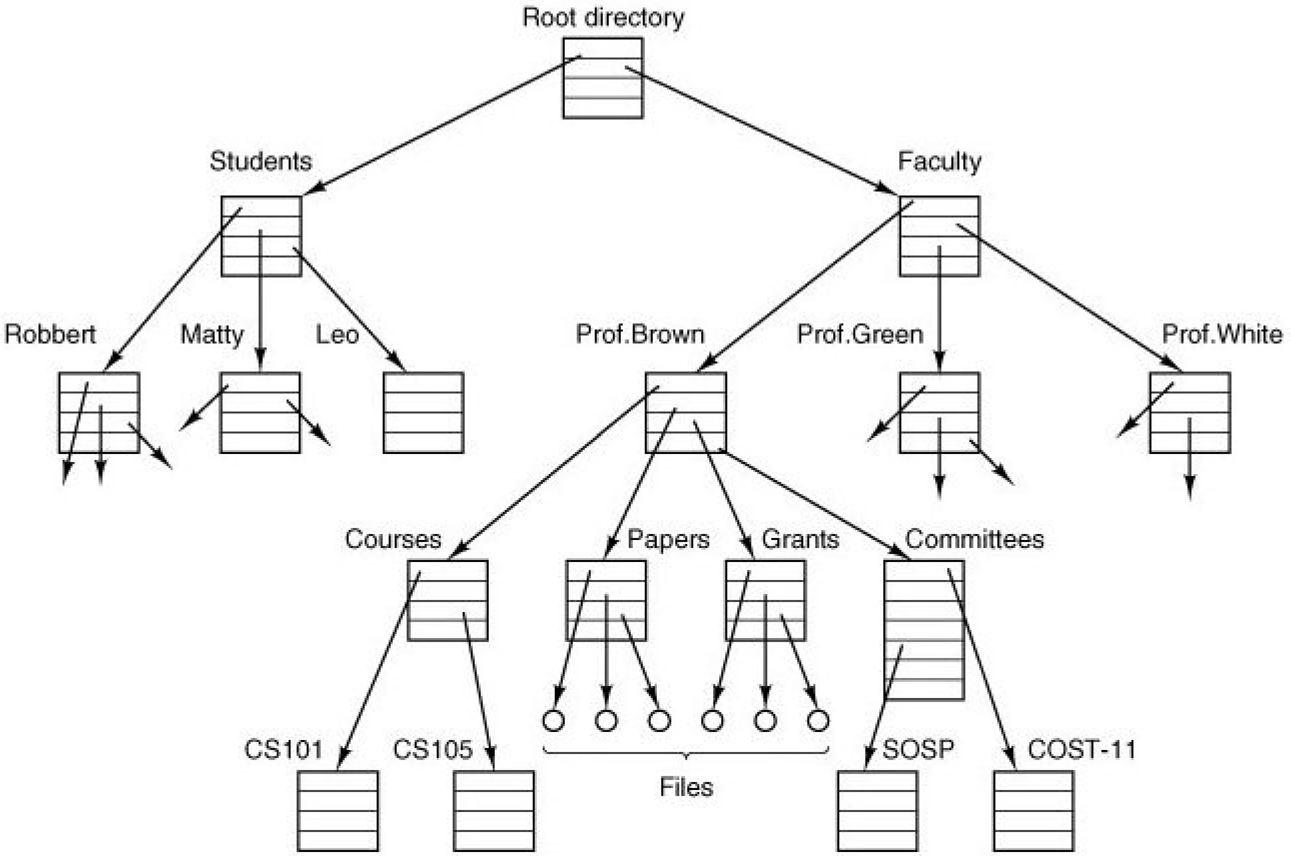


Fig 1.4: A Modern Computer System

Files:

A major function of the operating system is to hide the peculiarities of the disks and other I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be opened, and after it has been read it should be closed, so calls are provided to do these things.

To provide a place to keep files, most operating system has the concept of a directory as a way of grouping files together. A student, for example, might have one directory for each course he is taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. System calls are then needed to create and remove directories. Calls are also provided to put an existing file into a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy of the file system as shown in fig.



Every file within the directory hierarchy can be specified by giving its path name from the top of the directory hierarchy, the root directory. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 1-6, the path for file CS101 is /Faculty/Prof.Brown/Courses/CS101. The leading slash indicates that the path is absolute, that is, starting at the root directory. As an aside, in Windows, the backslash (\) character is used as the separator instead of the slash (/) character, so the file path given above would be written as \Faculty\Prof.Brown\Courses\CS101.

System Call:

In computing, a **system call** is how a program requests a service from an operating system's kernel. This may include hardware related services (e.g. accessing the hard disk), creating and executing new processes, and communicating with integral kernel services (like scheduling). System calls provide the interface between a process and the operating system.
On Unix, Unix-like and other POSIX-compatible operating systems, popular system calls are open, read, write, close, wait, execve, fork, exit, and kill. Many of today's operating systems have hundreds of system calls. For example, Linux has over 300 different calls.

See the figure in book page 51

System calls can be roughly grouped into five major categories:

1. Process Control.

System Called --> Police Force CID

- load
- execute
- create process

    - terminate process
    - get/set process attributes for all
    - wait for time, wait event, signal event
    - allocate, free memory
2. File management.
    - create file, delete file
    - open, close
    - read, write, reposition
    - get/set file attributes
3. Device Management.
    - request device, release device
    - read, write, reposition
    - get/set device attributes
    - logically attach or detach devices
4. Information Maintenance.
    - get/set time or date
    - get/set system data
    - get/set process, file, or device attributes
5. Communication.
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices

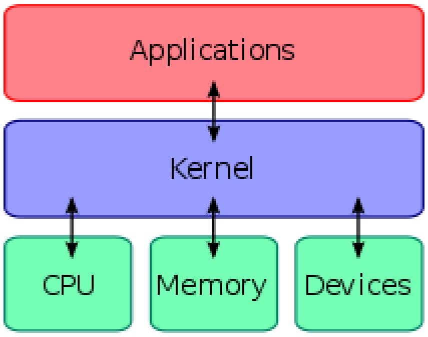
Shell:

A shell is a program that provides the traditional text only user interface for Linux and other Unix operating system. Its primary function is to read commands typed into a console or terminal window and then execute it. The term shell derives its name form the fact that it is an outer layer of the OS. A shell is an interface between the user and the internal part of the operating system.
A user is in shell(i.e interacting with the shell) as soon as the user has logged into the system. A shell is the most fundamental way that user can interact with the system and the shell hides the detail of the underlying system from the user.
Example:
Bourne Shell
Bash shell
Korn Shell
C shell

Kernel:

In computing, the **kernel** is the main component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources (the communication between hardware and software components). Usually as a basic component of an operating system, a kernel can provide the lowest-level abstraction layer for the resources (especially processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

Operating system tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels execute all the operating system code in the same address space to increase the performance of the system, microkernels run most of the operating system services in user space as servers, aiming to improve maintainability and modularity of the operating system. A range of possibilities exists between these two extremes.



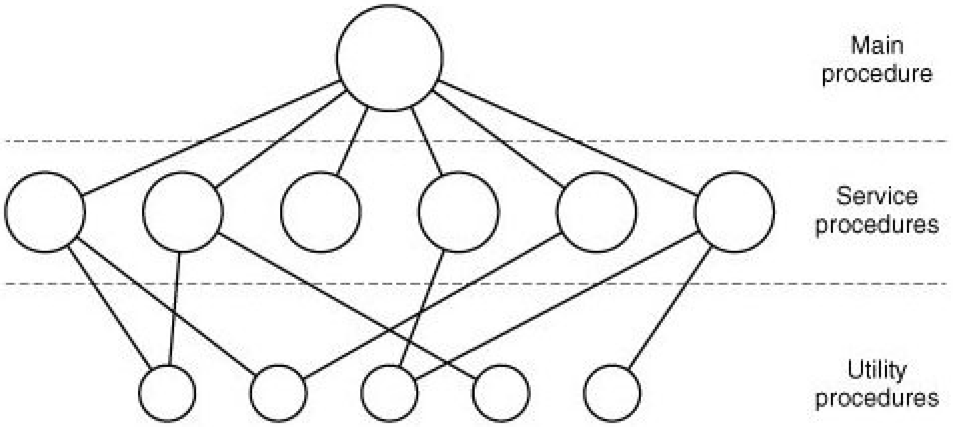
Operating System Structure:

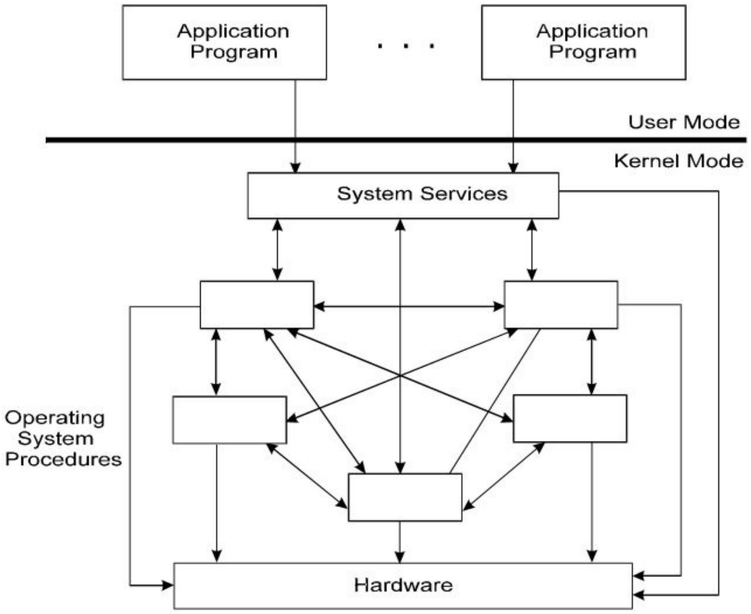
The structure of an operating system is dictated by the model employed in building them. An operating system model is a broad framework that unifies the many features and services the operating system provides and tasks it performs. Operating systems are broadly classified into following categories, based on the their structuring mechanism as follows:

a. Monolithic System
b. Layered System
c. Virtual Machine
d. Exokernels
e. Client-Server Model

Monolithic System

The components of monolithic operating system are organized haphazardly and any module can call any other module without any reservation. Similar to the other operating systems, applications in monolithic OS are separated from the operating system itself. That is, the operating system code runs in a privileged processor mode (referred to as kernel mode), with access to system data and to the hardware; applications run in a non-privileged processor mode (called the user mode), with a limited set of interfaces available and with limited access to system data. The monolithic operating system structure with separate user and kernel processor mode is shown in Figure.





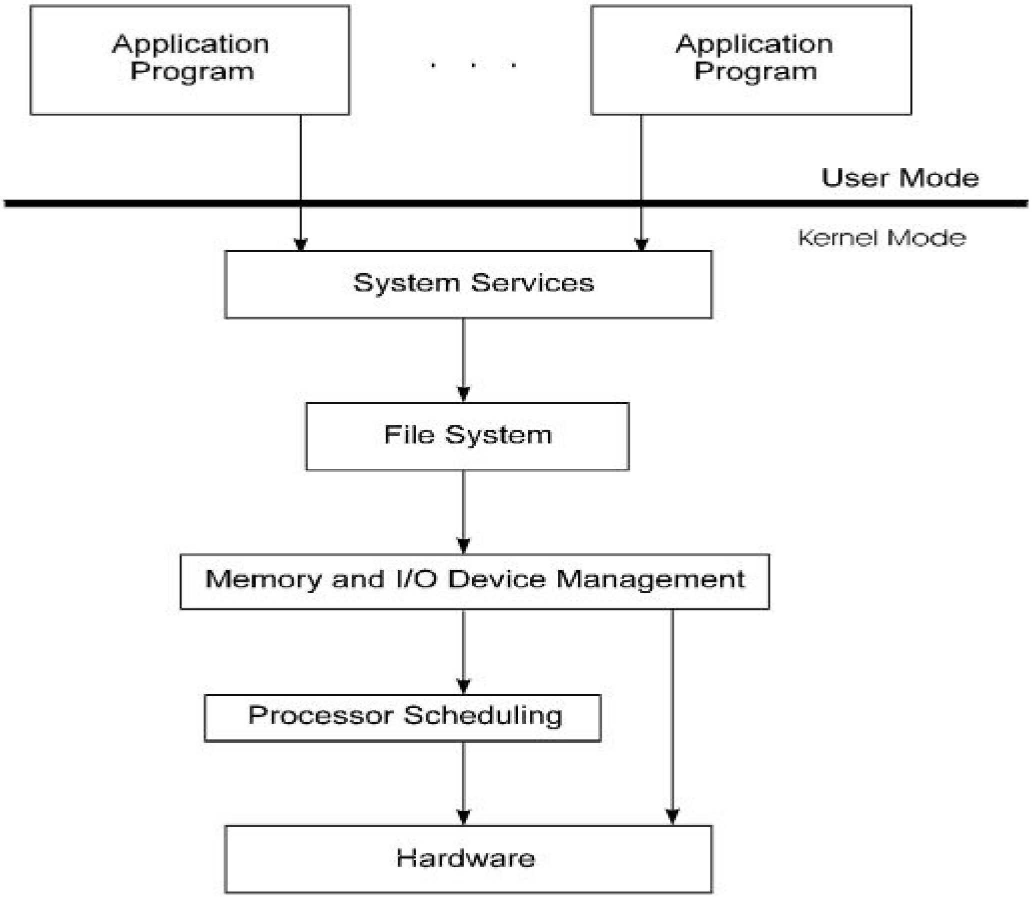
This approach might well be subtitled "The Big Mess." The structure is that there is no structure. The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

Example Systems: CP/M and MS-DOS

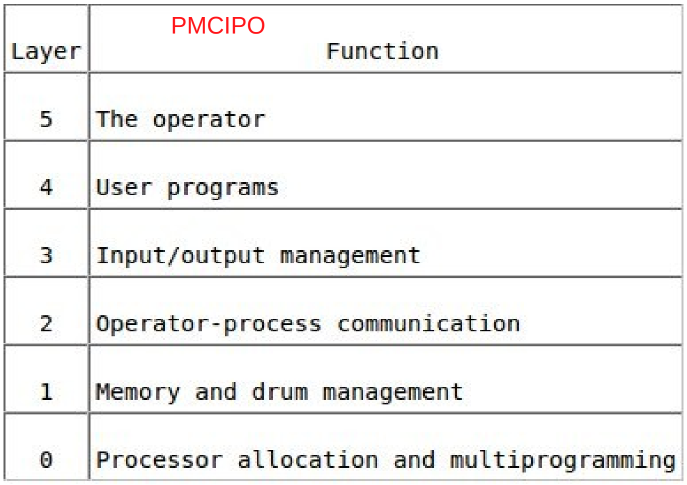
Layered Operating System

The layered approach consists of breaking the operating system into the number of layers(level), each built on the top of lower layers. The bottom layer (layer 0) is the hardware layer; the highest layer is the user interface.

The main advantages of the layered approach is modularity. The layers are selected such that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verifications. That is in this approach, the Nth layer can access services provided by the (N-1)th layer and provide services to the (N+1)th layer. This structure also allows the operating system to be debugged starting at the lowest layer, adding one layer at a time until the whole system works correctly. Layering also makes it easier to enhance the operating system; one entire layer can be replaced without affecting other parts of the system.



The layer approach to design was first used in the THE operating system at the Technische Hogeschool Eindhoven. The THE system was defined in the six layers , as shown in the fig below.



Layer	PMCIPO Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Example Systems: VAX/VMS, Multics, UNIX

Virtual Machines:

Virtual machine approach provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer as shown in the fig. The resources of the physical computer are shared to create the virtual machine. CPU scheduling can be used to share the CPU and to create the appearance that users have their own processors.

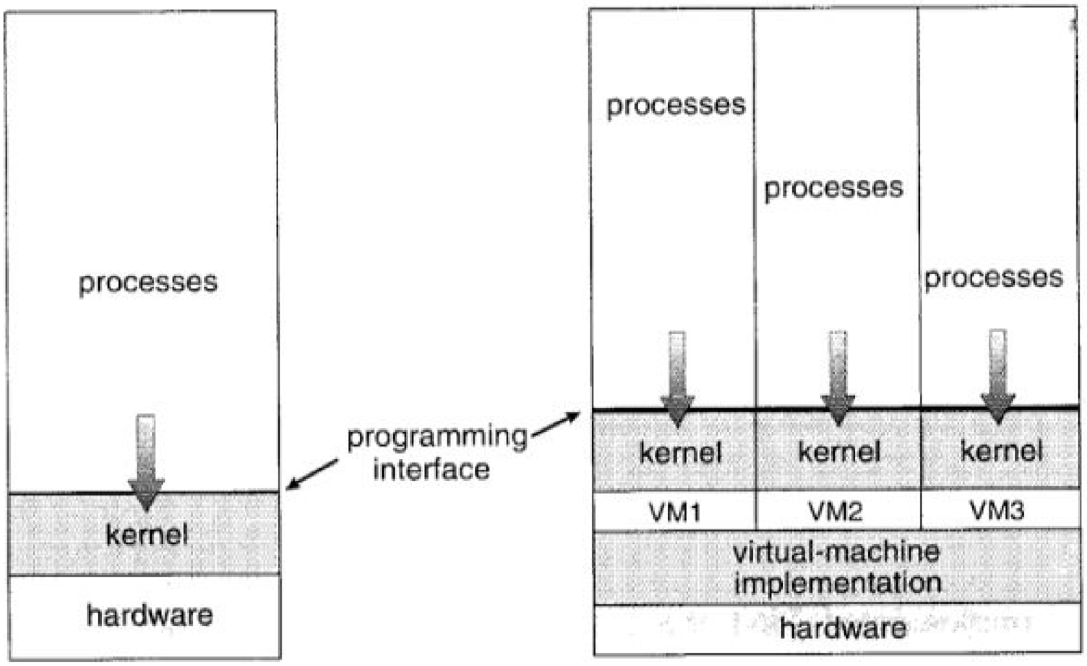


Fig: a). Non Virtual Machine b). Virtual Machine.

Although the virtual machine concept is useful, it is difficult to implement. Much effort is required to provide an exact duplicate of the underlying machine.

Example. Java

Client-Server or Microkernel

The advent of new concepts in operating system design, microkernel, is aimed at migrating traditional services of an operating system out of the monolithic kernel into the user-level process. The idea is to divide the operating system into several processes, each of which implements a single set of services - for example, I/O servers, memory server, process server, threads interface system. Each server runs in user mode, provides services to the requested client. The client, which can be either another operating system component or application program, requests a service by sending a message to the server. An OS kernel (or microkernel) running in kernel mode delivers the message to the appropriate server; the server performs the operation; and microkernel delivers the results to the client in another message, as illustrated in Figure.

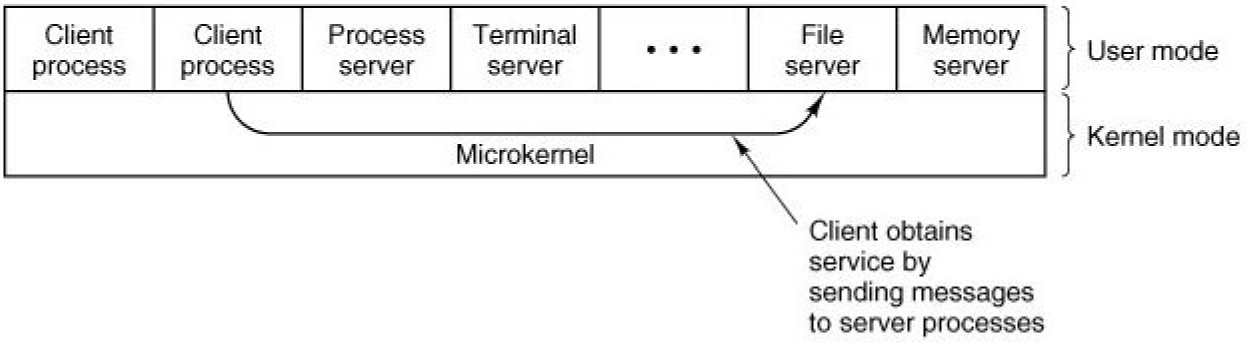


Fig: The client-server model.

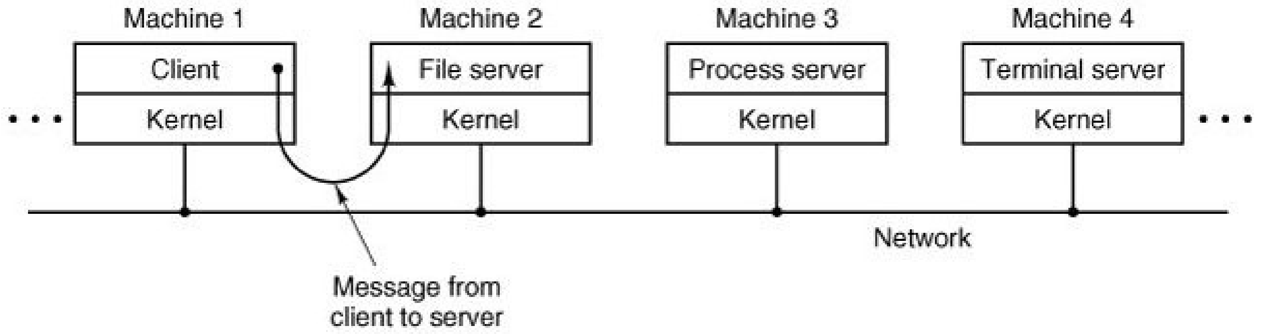


Fig: The client-server model in a distributed system.

Function of Operating system:

1. Memory management function
2. processors management function
3. I/O Device management function
4. File management function

Evolution of Operating System:

Evolution of Operating Systems: User driven, operator driven, simple batch system, off – line batch system, directly coupled off – line system, multi- programmed spooling system, online timesharing system, multiprocessor systems, multi-computer/ distributed systems, Real time Operating Systems.

1. Serial processing
2. Batch processing
3. Multiprogramming
4. Multitasking or time sharing System

5. Network Operating system
6. Distributed Operating system
7. Multiprocessor Operating System
8. Real Time Operating System
9. Modern Operating system

Serial Processing:

- Early computer from late 1940 to the mid 1950.
- The programmer interacted directly with the computer hardware.
- These machine are called bare machine as they don't have OS.
- Every computer system is programmed in its machine language.
- Uses Punch Card, paper tapes and language translator

These system presented two major problems.

1. Scheduling
2. Set up time:

Scheduling:

Used sign up sheet to reserve machine time. A user may sign up for an hour but finishes his job in 45 minutes. This would result in wasted computer idle time, also the user might run into the problem not finish his job in alloted time.

Set up time:

A single program involves:

- Loading compiler and source program in memory
- Saving the compiled program (object code)
- Loading and linking together object program and common function

Each of these steps involves the mounting or dismounting tapes on setting up punch cards. If an error occur user had to go the beginning of the set up sequence. Thus, a considerable amount of time is spent in setting up the program to run.

This mode of operation is turned as serial processing ,reflecting the fact that users access the computer in series.

Simple Batch Processing:

- Early computers were very expensive, and therefore it was important to maximize processor utilization.
- The wasted time due to scheduling and setup time in Serial Processing was unacceptable.
- To improve utilization, the concept of a batch operating system was developed.
- Batch is defined as a group of jobs with similar needs. The operating system allows users to form batches. Computer executes each batch sequentially, processing all jobs of a batch considering them as a single process called batch processing.

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor**. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

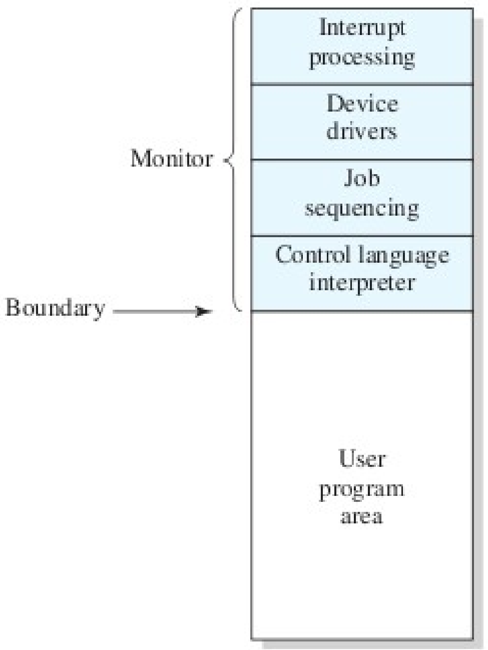


Fig.1.5: Memory Layout for resident memory

With a batch operating system, processor time alternates between execution of user programs and execution of the monitor. There have been two sacrifices: Some main memory is now given over to the monitor and some processor time is consumed by the monitor. Both of these are forms of overhead.

Multiprogrammed Batch System:

A single program cannot keep either CPU or I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs in such a manner that CPU has always one job to execute. If computer is required to run several programs at the same time, the processor could be kept busy for the most of the time by switching its attention from one program to the next. Additionally I/O transfer could overlap the processor activity i.e, while one program is awaiting for an I/O transfer, another program can use the processor. So CPU never sits idle or if comes in idle state then after a very small time it is again busy. This is illustrated in fig below.

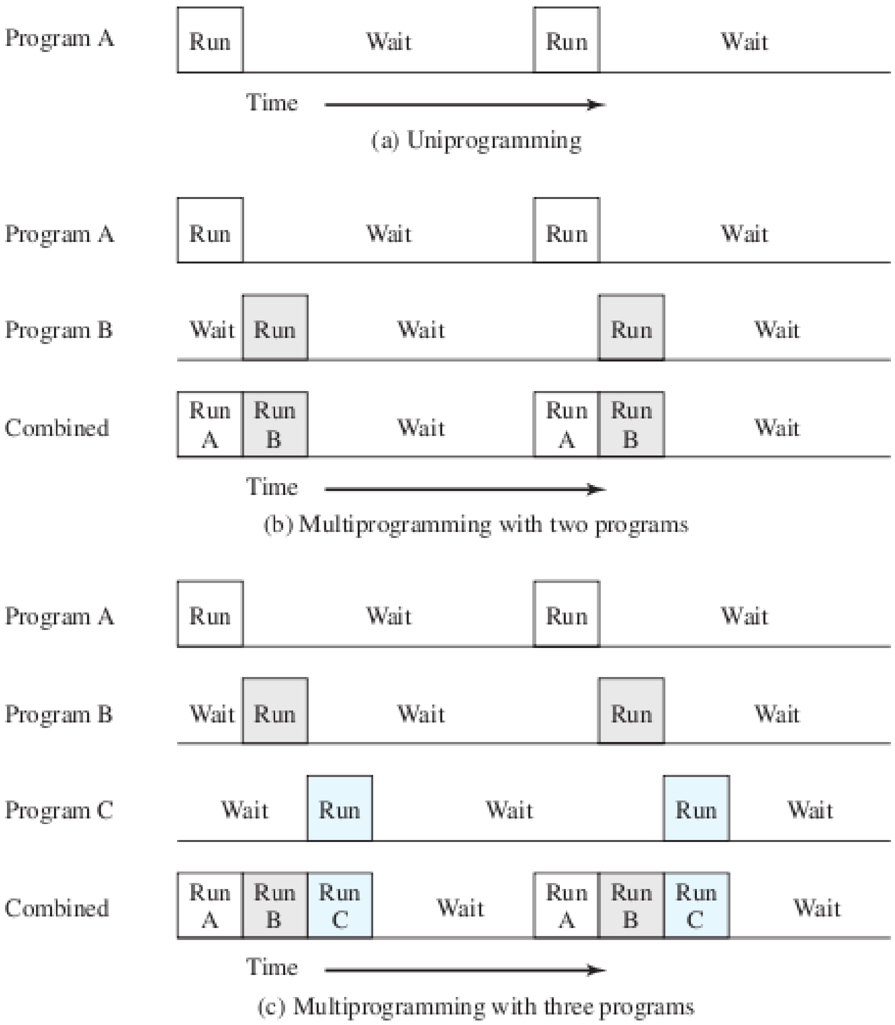


Fig 1.6. Multiprogramming example

Multitasking or Time Sharing System:

- Multiprogramming didn't provide the user interaction with the computer system.
- Time sharing or Multitasking is a logical extension of Multiprogramming that provides user interaction.
- There are more than one user interacting the system at the same time
- The switching of CPU between two users is so fast that it gives the impression to user that he is only working on the system but actually it is shared among different users.
- CPU bound is divided into different time slots depending upon the number of users using the system.
- just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as time sharing, because processor time is shared among multiple users
- A multitasking system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared computer. Each user has at least one separate program in

memory.

- Multitasking are more complex than multiprogramming and must provide a mechanism for jobs synchronization and communication and it may ensure that system does not go in deadlock.

Although batch processing is still in use but most of the system today available uses the concept of multitasking and Multiprogramming.

Distributed System:

Multiprocessor system:

- General term for the use of two or more CPUs for a computer system.
- Can vary with context, mostly as a function of how CPUs are defined.
- The term multiprocessing is sometimes used to refer to the execution of multiple concurrent software processes in a system as opposed to a single process at any one instant.

Multiprogramming:

Multiprogramming is more appropriate to describe concept which is implemented mostly in softwares, whereas multiprocessing is more appropriate to describe the use of multiple hardware CPUs.

A system can be both multiprocessing and multiprogramming, only one of the two or neither of the two.

Processor Coupling:

Its the logical connection of the CPUs. Multiprocessor system have more than one processing unit sharing memory/peripherals devices. They have greater computing power and higher reliability. Multiprocessor system can be classified into two types:

1. Tightly coupled
2. Losely coupled(distributed). Each processor has its own memory and copy of the OS.

Tightly Coupled(Multiprocessor System): Tightly coupled multiprocessor system contain multiple CPUs that are connected at the bus level. Each processor is assigned a specific duty but processor work in close association, possibly sharing one memory module.

chip multiprocessors also known as multi-core computing involves more than one processor placed on a single chip and can be thought as the most extreme form of tightly coupled multiprogramming.

Dual core, Core-2 Duo, Intel Core I5 etc are the brand name used for various mid-range to high end consumers and business multiprocessor made by Intel.

Loosely Coupled(Distributed System):

Loosely coupled system often referred to as clusters are based on multiple stand-alone single or dual processors commodity computers interconnected via a high speed communication system. distributed system are connected via a distributed operating system.

Multiprocessor operating system:

Multiprocessor operating system aims to support high performance through the use of multiple CPUs. It consists of a set of processors that share a set of physical memory blocks over an interconnected

network. An important goal is to make the number of CPUs transparent to the application. Achieving such transparency is relatively easy because the communication between different (parts of) application uses the same primitives as those in uni-processor OS.
The idea is that all communication is done by manipulating data at the shared memory locations and that we only have to protect that data segment against simultaneous access. Protection is done through synchronization primitives like semaphores and monitors.

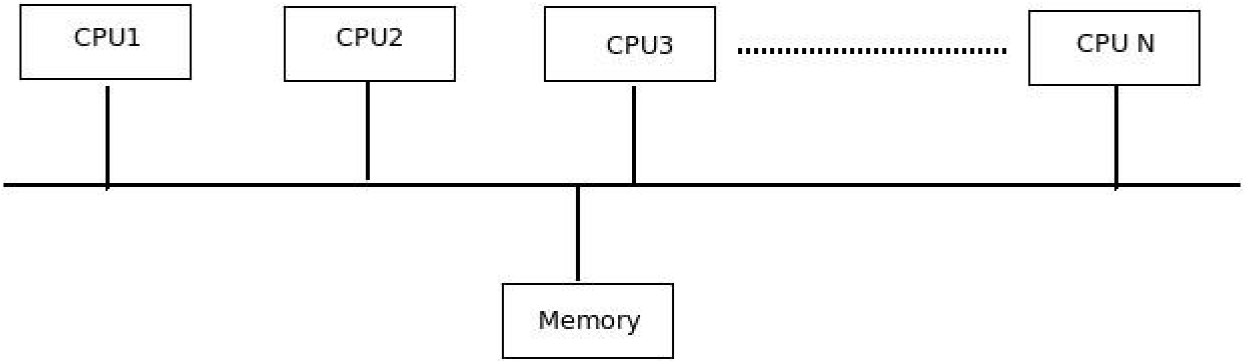


Fig: Multiprocessor System.

Distributed Operating System:

A recent trend in computer system is to distribute computation among several processors. In contrasts to the tightly coupled system the processors do not share memory or a clock. Instead, each processor has its own local memory. The processors communicate with one another through various communication lines such as computer network. Distributed operating system are the operating system for a distributed system(a network of autonomous computers connected by a communication network through a message passing mechanisms). A distributed operating system controls and manages the hardware and software resources of a distributed system. When a program is executed on a distributed system, user is not aware of where the program is executed or the location of the resources accessed.

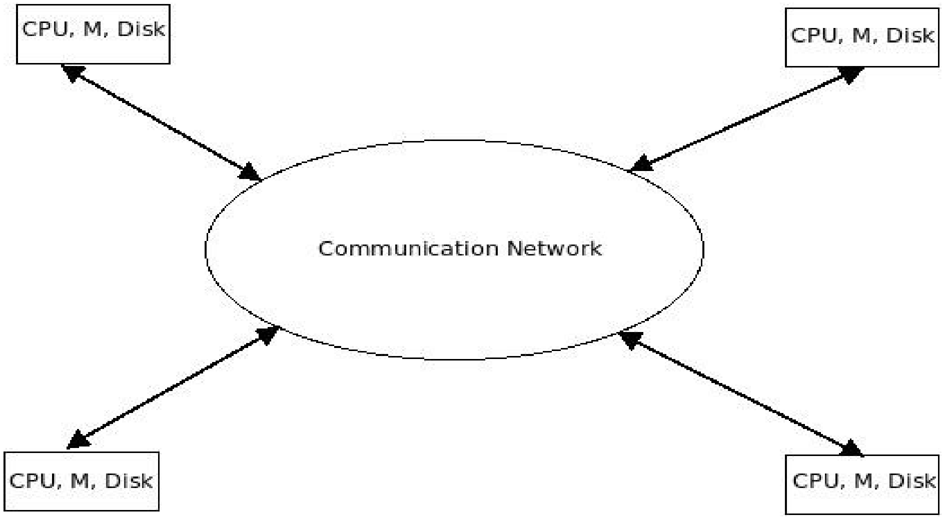


Fig: Architecture of a Distributed system.

Example of Distributed OS: Amoeba, Chorus, Alpha Kernel.

Real Time Operating System:

Primary objective of Real Time Operating System is to provide quick response time and thus to meet a scheduling deadline. User convenience and resource utilization are secondary concern to these systems. Real time systems has many events that must be accepted and processed in a short time or within certain deadline. Such applications include:
Rocket launching, flight control, robotics, real time simulation, telephone switching equipments etc.

Real time systems are classified into two categories:
a). Soft Real time System: If certain deadlines are missed then system continues its working with no failure but its performance degrade.

b). Hard Real time System: If any deadline is missed then system will fail to work or does not work properly. This system gurantees that critical task is completed on time.

Chapter-2 Processes and Threads

Process Concepts: Introduction, Definition of Process, Process states and transition, PCB (Process Control Block), Concurrent Process: Introduction, Parallel Processing, IPC (Inter-process Communication), Critical Regions and conditions, Mutual Exclusion, Mutual Exclusion Primitives and Implementation, Dekker's Algorithm, Peterson's Algorithm, TSL (test and set lock), Locks, Producer and consumer problem, monitors, Message Passing, Classical IPC problems, Deadlock and Indefinite Postponement: Introduction, Preemptable and Nonpreemptable Resources, Conditions for deadlock, deadlock modeling, deadlock prevention, deadlock avoidance, dadlock detection and recovery, Starvation, Threads: Introduction, thread model, thread usage, advantages of threads.

Introduction to process:

A process is an instance of a program in execution. A program by itself is not a process; a program is a ***passive entity***, such as a file containing a list of instructions stored on disks. (often called an executable file), whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory.

Program: A set of instructions a computer can interpret and execute.

Process:

- Dynamic
- Part of a program in execution
- a live entity, it can be created, executed and terminated.
- It goes through different states
  wait
  running
  Ready etc
- Requires resources to be allocated by the OS
- one or more processes may be executing the same code.

Program:

- static
- no states

This example illustrate the difference between a process and a program:

```
main ()
 {
        int i , prod =1;
        for (i=0;i<100;i++)
        prod = pord*i;
 }
```

It is a program containing one multiplication statement (prod = prod * i) but the process will execute

100 multiplication, one at a time through the 'for' loop.
Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance several users may be running different copies of mail program, or the same user may invoke many copies of web browser program. Each of these is a separate process, and although the text sections are equivalent, the data, heap and stack section may vary.

The process Model

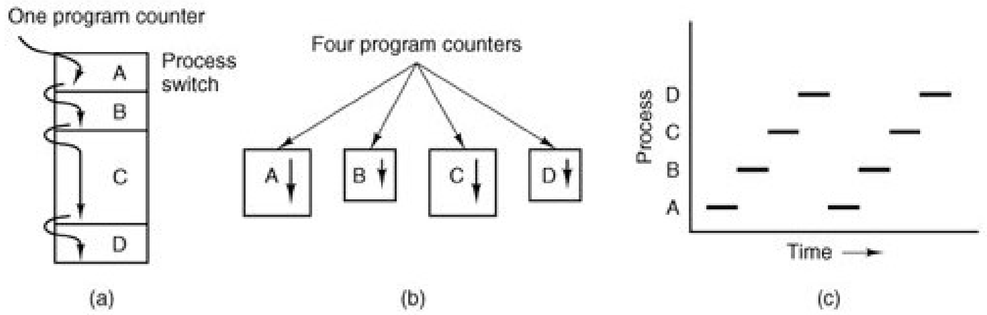


Fig:1.1: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

A process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called multiprogramming

Process Creation:

There are four principal events that cause processes to be created:
1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Parent process create children processes, which, in turn create other processes, forming a tree of processes . Generally, process identified and managed via a process identifier (pid)

When an operating system is booted, often several processes are created.

Some of these are foreground processes, that is, processes that interact with (human) users and perform work for them.

Others are background processes, which are not associated with particular users, but instead have some specific function. For example, a background process may be designed to accept incoming requests for web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as web pages, printing, and so on are called daemons
In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job.

In interactive systems, users can start a program by typing a command or double clicking an icon.
In UNIX there is only one system call to create a new process: **fork.** This call creates an exact clone of the calling process. After the fork, the two processes, the parent and the child, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes **execve** or a similar system call to change its memory image and run a new program. For example, when a user types a command, say, sort, to the shell, the shell forks off a child process and the child executes sort.

The C program shown below describes the system call, fork and exec used in UNIX. We now have two different process running a copy of the same program. The value of the PID for the child process is zero; that for the parent is an integer value greater than zero. The child process overlays its address space with the UNIX command /bin/ls using the execlp() system call. (execlp is version of the exec). The parent waits for the child process to complete with the wait() system call. When the child process completes (by either implicitly or explicitly invoking the exit()) the parent process resumes from the call to wait, where it completes using the exit() system call. This is also illustrated in the fig. below.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            printf("%d", pid);
              exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete\n");
            printf("%d",pid);
              exit(0);
    }
}
```

The last situation in which processes are created applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

UNIX System Call:

fork system call creates new process

exec system call used after a fork to replace the process' memory space with a new program

Process Control Block:

In operating system each process is represented by a process control block(PCB) or a task control block. Its a data structure that physically represent a process in the memory of a computer system. It contains many pieces of information associated with a specific process that includes the following.

- **Identifier**: A unique identifier associated with this process, to distinguish it
  from all other processes.
- **State**: If the process is currently executing, it is in the running state.
- **Priority**: Priority level relative to other processes.
- **Program counter**: The address of the next instruction in the program to be
  executed.
- **Memory pointers:** Includes pointers to the program code and data associated
  with this process, plus any memory blocks shared with other processes.
- **Context data**: These are data that are present in registers in the processor
  while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape
  drives) assigned to this process, a list of files in use by the process, and so on.

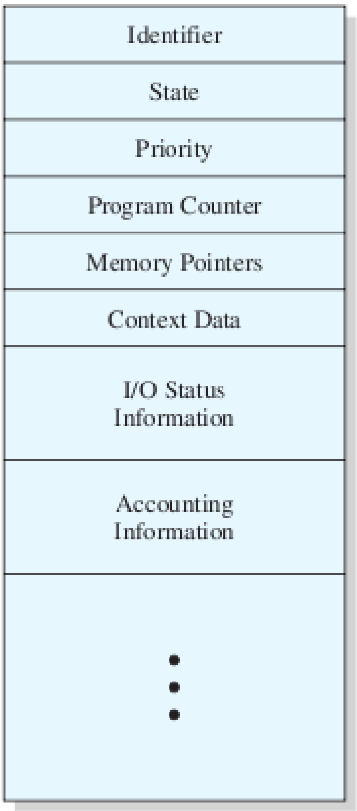


Fig1.2: Process Control Block

• **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

Process Termination:

After a process has been created, it starts running and does whatever its job is: After some time it will terminate due to one of the following conditions.

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process States:

Each process may be in one of the following states:

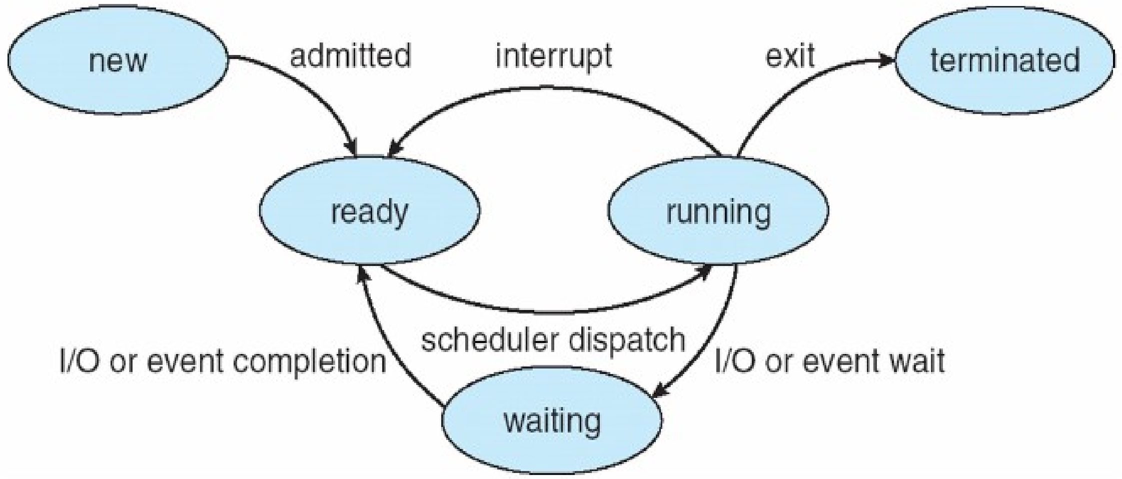


Fig1.3: Process state Transition diagram

- **New**: The process is being created.
- **Running:**Instructions are being executed.
- **Waiting:** The process is waiting for some event to occur (such as I/O completion or reception of a signal)
- **Ready:**The process is waiting to be assigned to a processor.
- **Terminated:** The process has finished execution.

Implementation of Process:

Operating system maintains a table (an array of structure) known as process table with one entry per process to implement the process. The entry contains detail about the process such as, *process state, program counter, stack pointer, memory allocation, the status of its open files, its accounting information, 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.

Process Management	**Memory Management**	**File management**
Register program counter Program status word stack pointer process state priority Scheduling parameter Process ID parent process process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment pinter to data segment pinter to stack segment	Root directory Working directory File descriptors USER ID GROUP ID

Each I/O device class is associated with a location (often near the bottom of the memory) called the ***Interrupt Vector.*** It contains the address of 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.

Interrupt handling and scheduling are summarized below.

1. Hardware stack program counter etc.
2. Hardware loads new program counter from interrupt vector
3. Assembly languages procedures save registers.
4. Assembly language procedures sets up new stack.
5. C interrupt service runs typically reads and buffer input.
6. Scheduler decides which process is to run next.
7. C procedures returns to the assembly code.
8. Assembly language procedures starts up new current process.

Fig:The above points lists the Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Context Switching:

Switching the CPU to another process requires performing a state save of the current process and a state restore restore of a different process. This task is known as context switch. When a context switch occurs, the kernel saves the the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied etc.

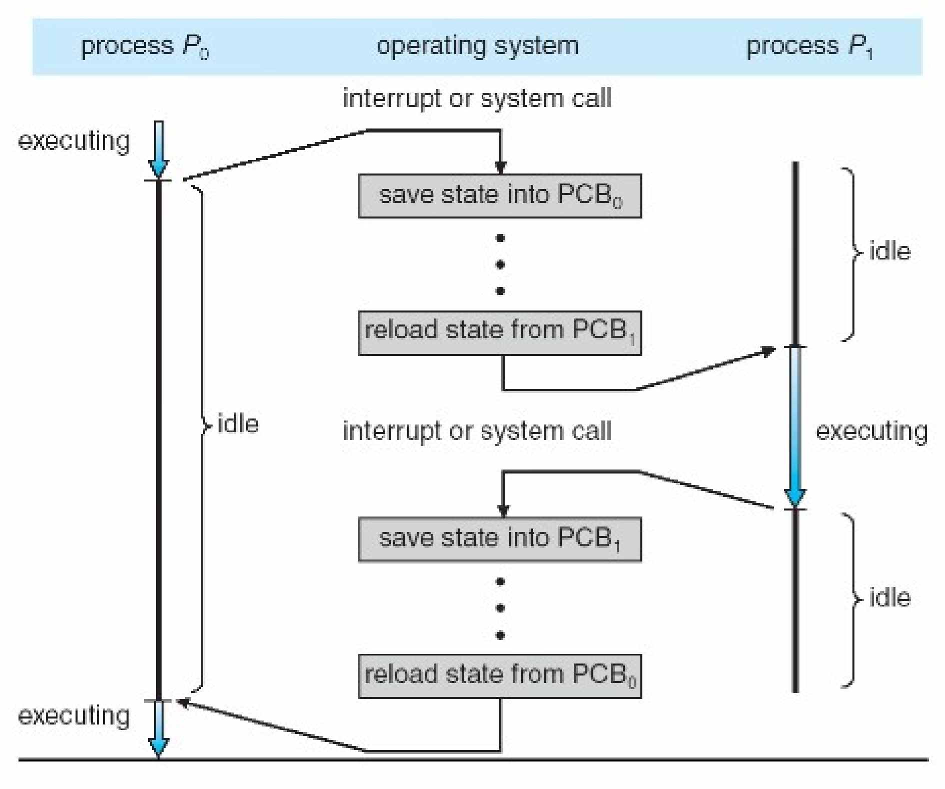


Fig: CPU switch from one process to another

Threads:

A thread is a basic unit of CPU utilization, it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating system resources, such as open files and signals. A traditional (or heavy weight) process has a single thread of control. If a process has multiple thread of control, it can perform more than one task at a time. Fig below illustrate the difference between single threaded process and a multi-threaded process.

Thread simply enable us to split up a program into logically separate pieces and have the pieces run independently of one another until they need to communicate. In a sense threads are a further level of object orientation for multitasking system.

Multithreading:

Many software package that run on modern desktop pcs are multi-threaded. An application is implemented as a separate process with several threads of control. A web browser might have one thread to display images or text while other thread retrieves data from the network. A word-processor may have a thread for displaying graphics, another thread for reading the character entered by user through the keyboard, and a third thread for performing spelling and grammar checking in the background.

Why Multithreading:

In certain situations, a single application may be required to perform several similar task such as a web server accepts client requests for web pages, images, sound, graphics etc. A busy web server may have several clients concurrently accessing it. So if the web server runs on traditional single threaded process, it would be able to service only one client at a time. The amount of time that the client might have to wait for its request to be serviced is enormous.

One solution of this problem can be thought by creation of new process. When the server receives a new request, it creates a separate process to service that request. But this method is heavy weight. In fact this process creation method was common before threads become popular. Process creation is time consuming and resource intensive. If the new process perform the same task as the existing process, why incur all that overhead? It is generally more efficient for one process that contains multiple threads to serve the same purpose. This approach would multithreaded the web server process. The server would cerate a separate thread that would listen for clients requests. When a request is made, rather than creating another process, it will create a separate thread to service the request.

Benefits of Multi-threading:

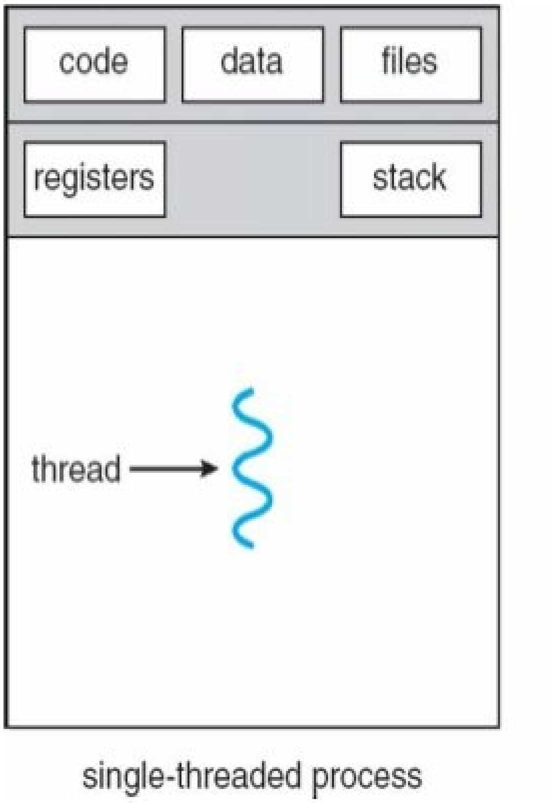
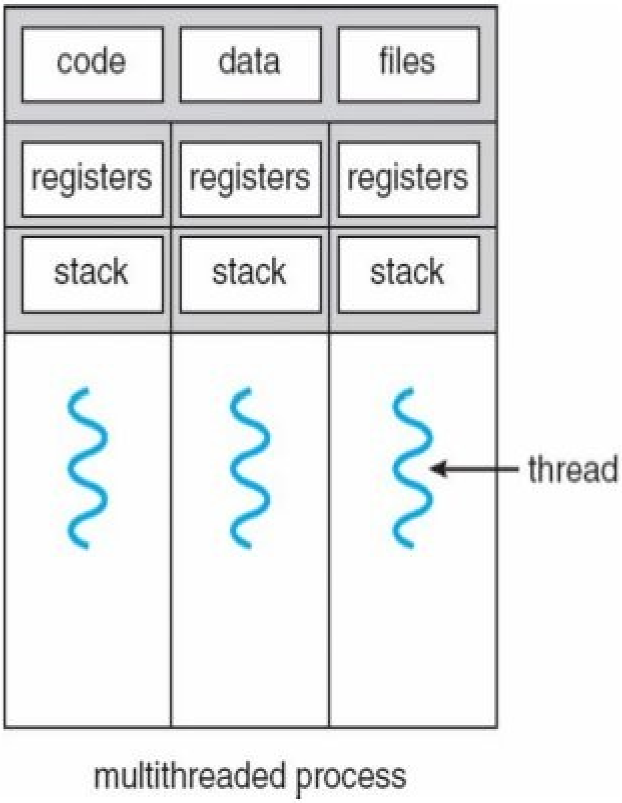
Responsiveness: Mutlithreaded interactive application continues to run even if part of it is blocked or performing a lengthy operation, thereby increasing the responsiveness to the user.

Resource Sharing: By default, threads share the memory and the resources of the process to which they belong. It allows an application to have several different threads of activity withing the same address space.

Economy: Allocating memory and resources for process creation is costly. Since thread shares the resources of the process to which they belong, it is more economical to create and context switch threads. It is more time consuming to create and manage process than threads.

Utilization of multiprocessor architecture: The benefits of multi threading can be greatly increased in multiprocessor architecture, where threads may be running in parallel on different processors. Mutlithreading on a multi-CPU increases concurrency.

Process VS Thread:

Process	***Thread***
 code data files registers stack thread single-threaded process	 code data files registers registers registers stack stack stack thread multithreaded process
Program in execution.	Basic unit of CPU utilization.
Heavy weight	Light weight
Unit of Allocation – Resources, privileges etc	Unit of Execution – PC, SP, registers PC—Program counter, SP—Stack pointer
Inter-process communication is expensive: need to context switch Secure: one process cannot corrupt another process	Inter-thread communication cheap: can use process memory and may not need to context switch Not secure: a thread can write the memory used by another thread
Process are Typically independent	Thread exist as subsets of a process
Process carry considerable state information.	Multiple thread within a process share state as well as memory and other resources.
Processes have separate address space	Thread share their address space

processes interact only through system-provided inter-process communication mechanisms.	Context switching between threads in the same process is typically faster than context switching between processes.

Multi-Threading Model:

User thread are supported above the kernel and are managed without the kernel support whereas kernel threads are supported and are manged directly by the operating system. Virtually all operating-system includes kernel threads. Ultimately there must exists a relationship between user threads and kernel threads. We have three models for it.

1. Many-to-one model

maps many user level threads to one kernel thread. Thread management is done by the thread library in user space so it is efficient; but the entire process will block if a thread makes a blocking system call. Also only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors. Green Threads - a thread library available for Solaris use this model.

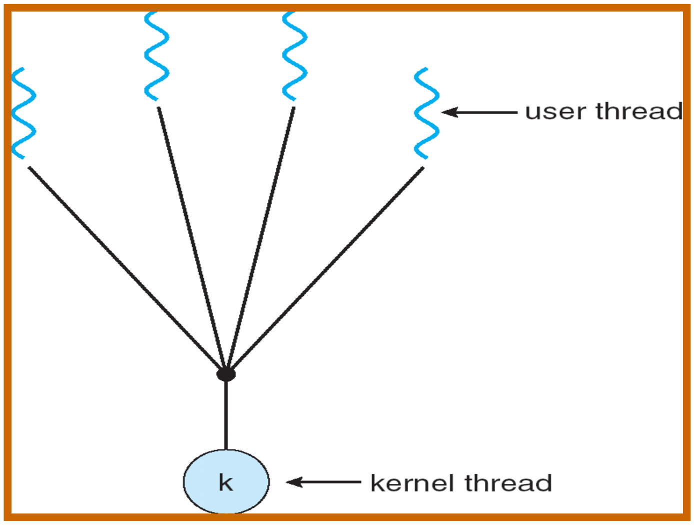


Fig: Many to One threading Model

2. One-to-one Model: maps each user thread to a kernel thread. It provides more concurrency than many to one model by allowing another thread to run when a thread makes a blocking system call. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Linux along with families of Windows operating system use this model.

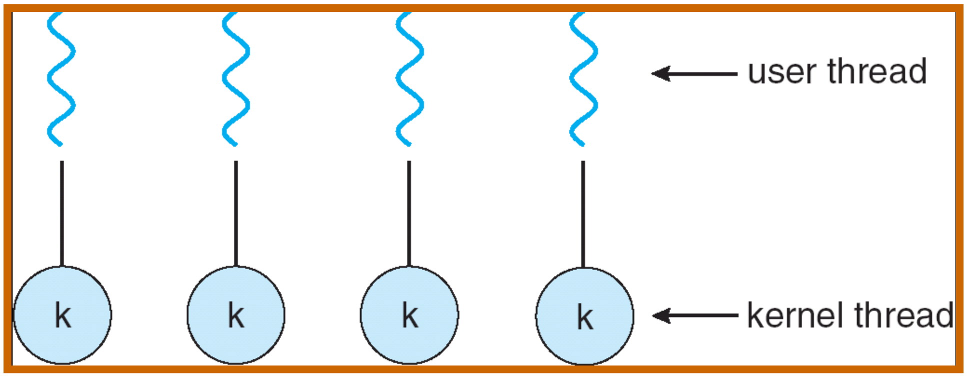


Fig: One to one Threading model

3. Many-to-many Model: multiplexes many user level thread to a smaller or equal number of kernel threads. The number of kernel thread may be specific to either a particular application or a particular machine. Many-to-many model allows the users to create as many threads as he wishes but the true concurrency is not gained because the kernel can schedule only one thread at a time.

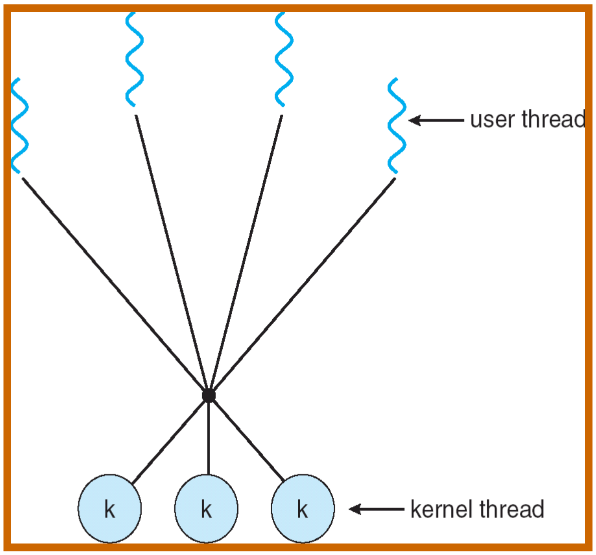


Fig:Many to Many

Interprocess Communication:

Processes frequently needs to communicate with each other. For example in a shell pipeline, the output of the first process must be passed to the second process and so on down the line. Thus there is a need for communication between the process, preferably in a well-structured way not using the interrupts.

IPC enables one application to control another application, and for several applications to share the same data without interfering with one another. Inter-process communication (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for **message passing, synchronization, shared memory, and remote procedure calls (RPC).**

co-operating Process: A process is independent if it can't affect or be affected by another process. A process is co-operating if it can affects other or be affected by the other process. Any process that shares data with other process is called co-operating process. There are many reasons for providing an environment for process co-operation.

1.Information sharing: Several users may be interested to access the same piece of information(for instance a shared file). We must allow concurrent access to such information.

2.Computation Speedup: Breakup tasks into sub-tasks.

3.Modularity: construct a system in a modular fashion.

4.convenience:

co-operating process requires IPC. There are two fundamental ways of IPC.
a. Shared Memory
b. Message Passing

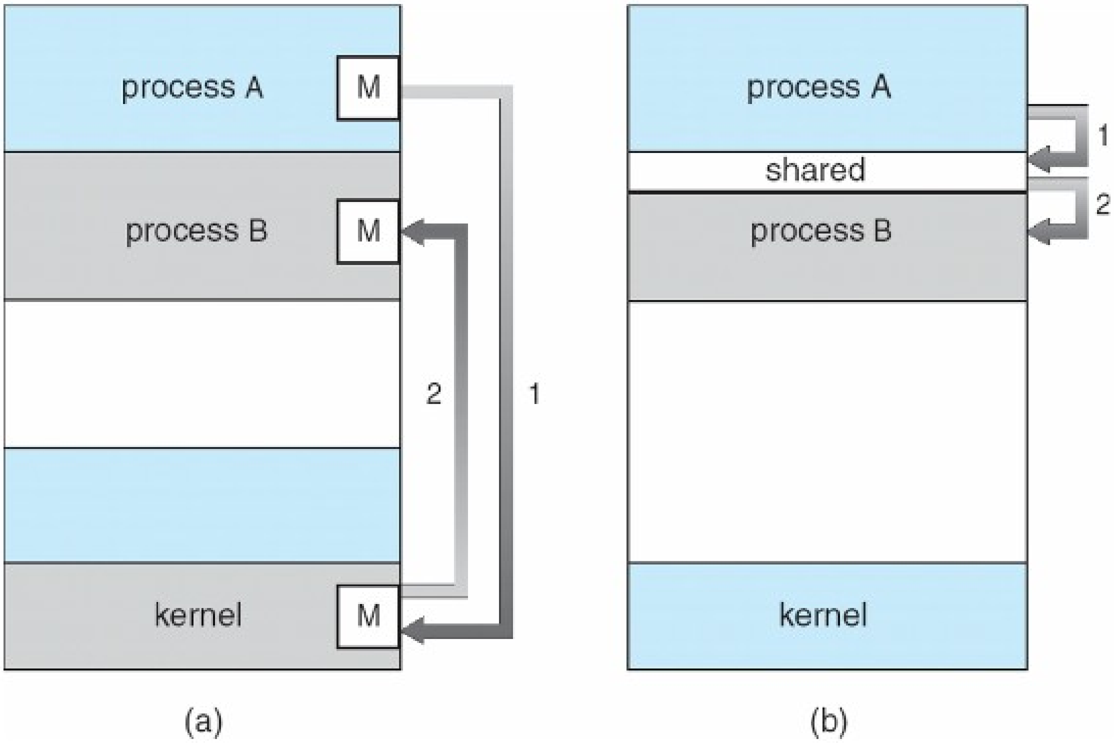


Fig: Communication Model a. Message Passing b. Shared Memory

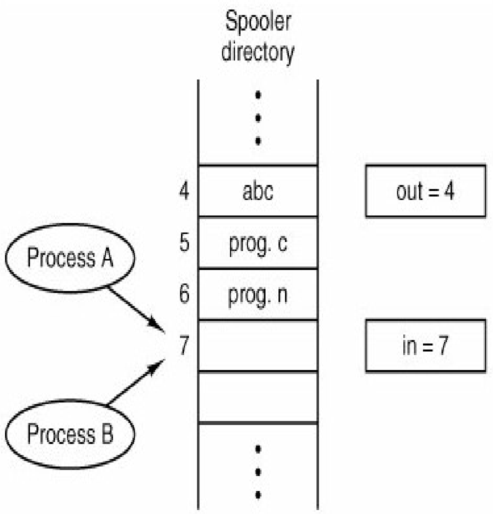
Shared Memory:

- Here a region of memory that is shared by co-operating process is established.
- Process can exchange the information by reading and writing data to the shared region.
- Shared memory allows maximum speed and convenience of communication as it can be done at the speed of memory within the computer.
- System calls are required only to establish shared memory regions. Once shared memory is established no assistance from the kernel is required,all access are treated as routine memory access.

Message Passing:

- communication takes place by means of messages exchanged between the co-operating process
- Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.
- Easier to implement than shared memory.
- Slower than that of Shared memory as message passing system are typically implemented using system call which requires more time consuming task of Kernel intervnetion.

Race Condition:



The situation where two or more processes are reading or writing some shared data & the final results depends on who runs precisely when are called **race conditions.**
To see how interprocess 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 removes their names from the directory.

Imagine that our spooler directory has a 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
in: which points to the next free slot in the directory.
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 to be printed). More or less simultaneously, processes A and B decide they want to queue a file for printing as shown in the fig.

Process A reads in and stores the value, 7, in a local variable called **next_free_slot**. Just then a clock interrupt occurs and the CPU decides that process A has run long enough, so it switches to process B.

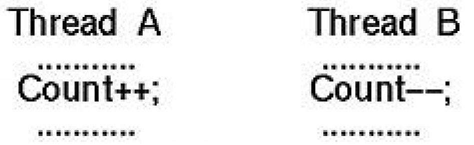
Process B also reads in, and also gets a 7, so it stores the name of its file in slot 7 and updates in to be an 8. Then it goes off and does other things.

Eventually, process A runs again, starting from the place it left off last time. It looks at **next_free_slot**, finds a 7 there, and writes its file name in slot 7, erasing the name that process B just put there. Then it computes **next_free_slot + 1**, which is 8, and sets **in** to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process B will never receive any output.

Spooling: Simultaneous peripheral operations online

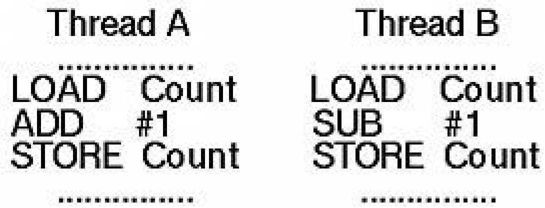
Another Example for Race Condition:

When two or more concurrently running threads/processes access a shared data item or resources and final results depends on the order of execution, we have race condition. Suppose we have two threads A and B as shown below. Thread A increase the share variable count by 1 and Thread B decrease the count by 1.



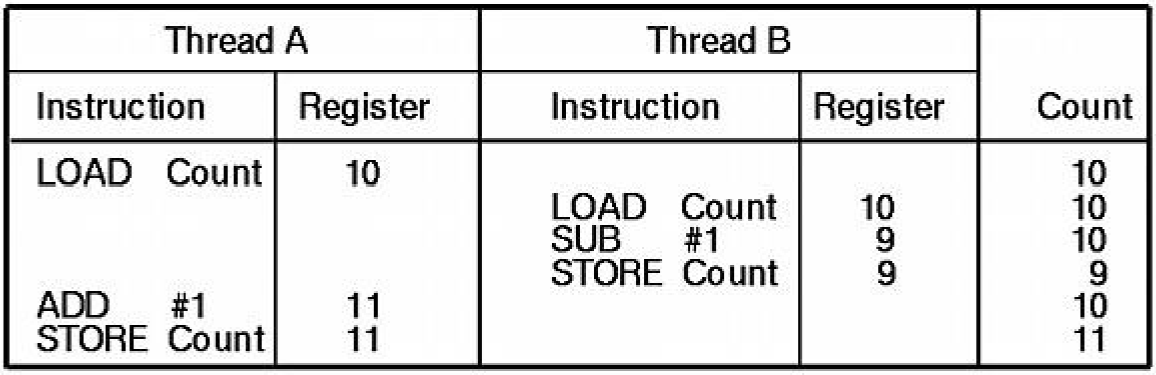
```
Thread  A           Thread  B
 ..........          ..........
 Count++;            Count--;
 ..........          ..........
```

If the current value of Count is 10, both execution orders yield 10 because Count is increased by 1 followed by decreased by 1 for the former case, and Count is decreased by 1 followed by increased by 1 for the latter. However, if Count is not protected by mutual exclusion, we might get difference results. Statements Count++ and Count-- may be translated to machine instructions as shown below:



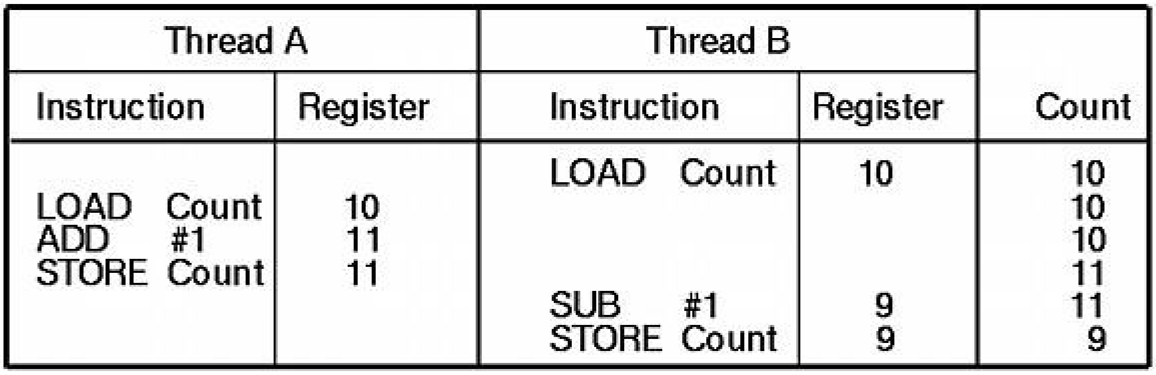
```
   Thread A            Thread B
 ..............      ..............
LOAD   Count        LOAD   Count
ADD     #1          SUB     #1
STORE  Count        STORE  Count
 ..............      ..............
```

If both statements are not protected, the execution of the instructions may be interleaved due to context switching. More precisely, while thread **A** is in the middle of executing Count++, the system may switch **A** out and let thread **B** run. Similarly, while thread **B** is in the middle of executing Count--, the system may switch **B** out and let thread **A** run. Should this happen, we have a problem. The following table shows the execution details. For each thread, this table shows the instruction executed and the content of its register. Note that registers are part of a thread's environment and different threads have different environments. Consequently, the modification of a register by a thread only affects the thread itself and will not affect the registers of other threads. The last column shows the value of Count in memory. Suppose thread **A** is running. After thread **A** executes its LOAD instruction, a context switch switches thread **A** out and switches thread **B** in. Thus, thread **B** executes its three instructions, changing the value of Count to 9. Then, the execution is switched back to thread **A**, which continues with the remaining two instructions. Consequently, the value of Count is changed to 11!



Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
LOAD Count	10			10
		LOAD Count	10	10
		SUB #1	9	10
		STORE Count	9	9
ADD #1	11			10
STORE Count	11			11

The following shows the execution flow of executing thread ***B*** followed by thread ***A***, and the result is 9!



Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
		LOAD Count	10	10
LOAD Count	10			10
ADD #1	11			10
STORE Count	11			11
		SUB #1	9	11
		STORE Count	9	9

This example shows that without mutual exclusion, the access to a shared data item may generate different results if the execution order is altered. This is, of course, a ***race condition***. This race condition is due to no mutual exclusion.

Avoiding Race Conditions:

Critical Section:
To avoid race condition we need **Mutual Exclusion. Mutual Exclusion** is someway of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

The difficulty above in the printer spooler occurs because process B started using one of the shared variables before process A was finished with it.

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 race conditions. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.

(Rules for avoiding Race Condition) Solution to Critical section problem:

1. **No two processes may be simultaneously inside their critical regions. (Mutual Exclusion)**
2. **No assumptions may be made about speeds or the number of CPUs.**
3. **No process running outside its critical region may block other processes.**
4. **No process should have to wait forever to enter its critical region.**

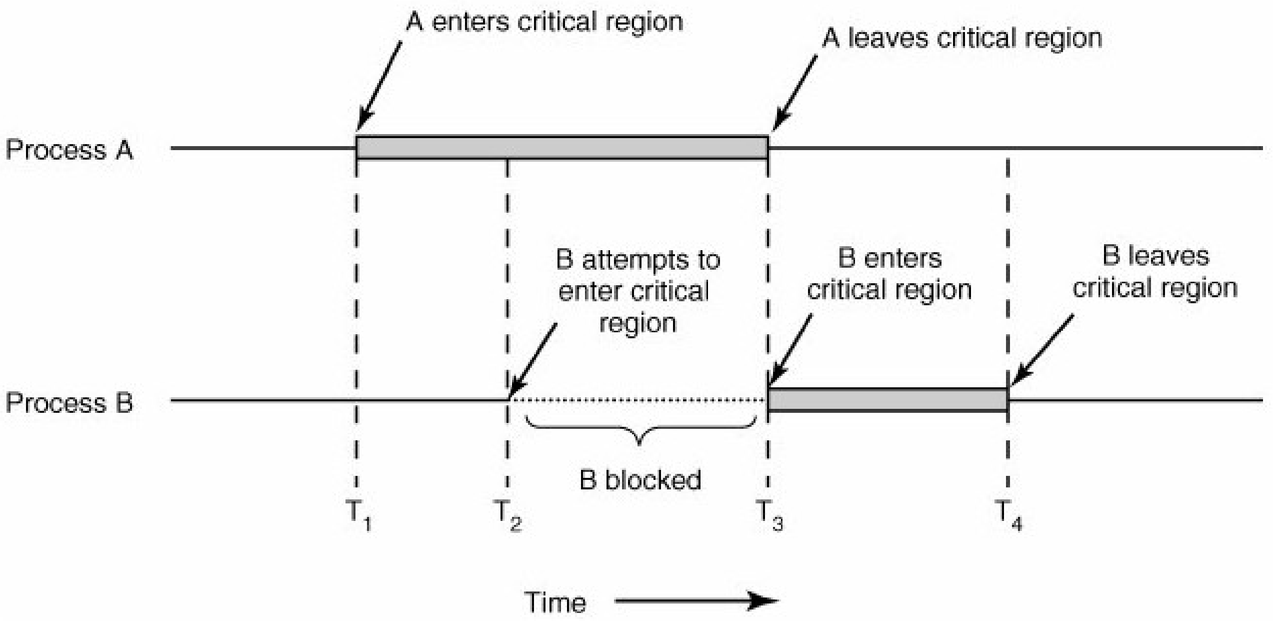


Fig:Mutual Exclusion using Critical Region

Techniques for avoiding Race Condition:

1. Disabling Interrupts
2. Lock Variables
3. Strict Alteration
4. Peterson's Solution
5. TSL instruction
6. Sleep and Wakeup
7. Semaphores
8. Monitors
9. Message Passing

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.

Disadvantages:
1. It is unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did, and then never turned them on again?
2.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.

Advantages:
 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.

2.Lock Variables

- a single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Drawbacks:
Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. 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.

3.Strict Alteration:

```
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)                                                   (b)
```

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Integer variable turn is initially 0.
It 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.
Continuously testing a variable until some value appears is called busy waiting. 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. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region. This way no two process can enters critical region simultaneously.

Drawbacks:

Taking turn is is not a good idea when one of the process is much slower than other. This situation requires that two processes strictly alternate in entering their critical region.

Example:

- Process 0 finishes the critical region it sets turn to 1 to allow process 1 to enter critical region.
- Suppose that process 1 finishes its critical region quickly so both process are in their non critical region with turn sets to 0.
- Process 0 executes its whole loop quickly, exiting its critical region & setting turn to 1. At this point turn is 1 and both processes are executing in their noncritical regions.
- Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now since turn is 1 and process 1 is busy with its noncritical region.

This situation violates the condition 3 set above: No process running outside the critical region may block other process. In fact the solution requires that the two processes strictly alternate in entering their critical region.

4.Peterson's Solution:

```
#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 */
}
```

Peterson's solution for achieving mutual exclusion.

Initially neither process is in 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 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.

5. The TSL Instruction

TSL RX,LOCK
(Test and Set Lock) that works as follows: 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 indivisibleno 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.

```
enter_region:
        TSL REGISTER,LOCK       |copy LOCK to register and set LOCK to 1
        CMP REGISTER,#0         |was LOCK zero?
        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
```

One solution to the critical region problem is now straightforward. 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. 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.

Problems with mutual Exclusion:

The above techniques achieves the mutual exclusion using busy waiting. Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.
Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is

1. This approach waste CPU time
2. There can be an unexpected problem called priority inversion problem.

Priority Inversion Problem:

Consider a computer with two processes, H, with high priority and L, with low priority, which share a critical region. 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. This situation is sometimes referred to as the priority inversion problem.

Let us now look at some IPC primitives that blocks instead of wasting CPU time when they are not allowed to enter their critical regions. Using blocking constructs greatly improves the CPU utilization

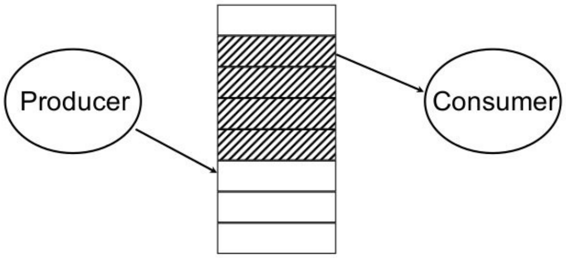
Sleep and Wakeup:

Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. 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.

Examples to use Sleep and Wakeup primitives:

Producer-consumer problem (Bounded Buffer):

Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.



Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
#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 */
    }
}
```

Fig:The producer-consumer problem with a fatal race condition.

N → Size of Buffer

Count--> a variable to keep track of the no. of items in the buffer.

Producers code:

The producers code is 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.

Consumer code:

It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.

Each of the process also tests to see if the other should be awakened and if so wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions as we saw in the spooler directory.

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)

3. The producer creates an item, puts it into the buffer, and increases count.
4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.
5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.
6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

Semaphore:

In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment . Synchronization tool that does not require busy waiting . **A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operations. P and V. If S is the semaphore variable, then,**

P operation: Wait for semaphore to become positive and then decrement
P(S):
while(S<=0)
do no-op;
S=S-1
	V Operation: Increment semaphore by 1
	V(S):
	S=S+1;

Atomic operations: When one process modifies the semaphore value, no other process can simultaneously modify that same semaphores value. In addition, in case of the P(S) operation the testing of the integer value of S (S<=0) and its possible modification (S=S-1), must also be executed without interruption.

Semaphore operations:
P or Down, or Wait: P stands for *proberen* for "to test
V or Up or Signal: Dutch words. **V** stands for *verhogen* ("increase")

wait(sem) -- decrement the semaphore value. if negative, suspend the process and place in queue. (Also referred to as *P(), down* in literature.)

signal(sem) -- increment the semaphore value, allow the first process in the queue to continue. (Also referred to as *V(), up* in literature.)

Counting semaphore – integer value can range over an unrestricted domain
Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement Also known as mutex locks

Provides mutual exclusion

```
Semaphore S;    //  initialized to 1
wait (S);
          Critical Section
signal (S);
```

Semaphore implementation without busy waiting:
Implementation of wait:

```
wait (S){
            value - -;
            if (value < 0) {
                     add this process to waiting queue
                      block();  }
                    }
```

Implementation of signal:

```
Signal (S){
               value++;
                if (value <= 0) {
                         remove a process P from the waiting queue
                          wakeup(P);  }
                     }
```

```
#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 = 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 */
    }
}
```

Fig: The producer-consumer problem using semaphores.

This solution uses three semaphore.

1. Full: For counting the number of slots that are full, initially 0
2. Empty: For counting the number of slots that are empty, initially equal to the no. of slots in the buffer.
3. Mutex: To make sure that the producer and consumer do not access the buffer at the same time, initially 1.

Here in this example seamphores are used in two different ways.

1.**For mutual Exclusion**: The mutex semaphore is for mutual exclusion. It is designed to guarantee that only one process at at time will be reading or writing the buffer and the associated variable.

2.**For synchronization:** The full and empty semaphores are needed to guarantee that certain certain event sequences do or do not occur. In this case, they ensure that producer stops running when the buffer is full and the consumer stops running when it is empty.

The above definition of the semaphore suffer the problem of busy wait. To overcome the need for busy waiting, we can modify the definition of the P and V operation of the semaphore. When a Process executes the P operation and finds that the semaphores value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places into a waiting queue associated with the semaphore, and the state of the process is switched to the

Advantages of semaphores:

- Processes do not busy wait while waiting for resources. While waiting, they are in a "suspended" state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

Disadvantages of semaphores:

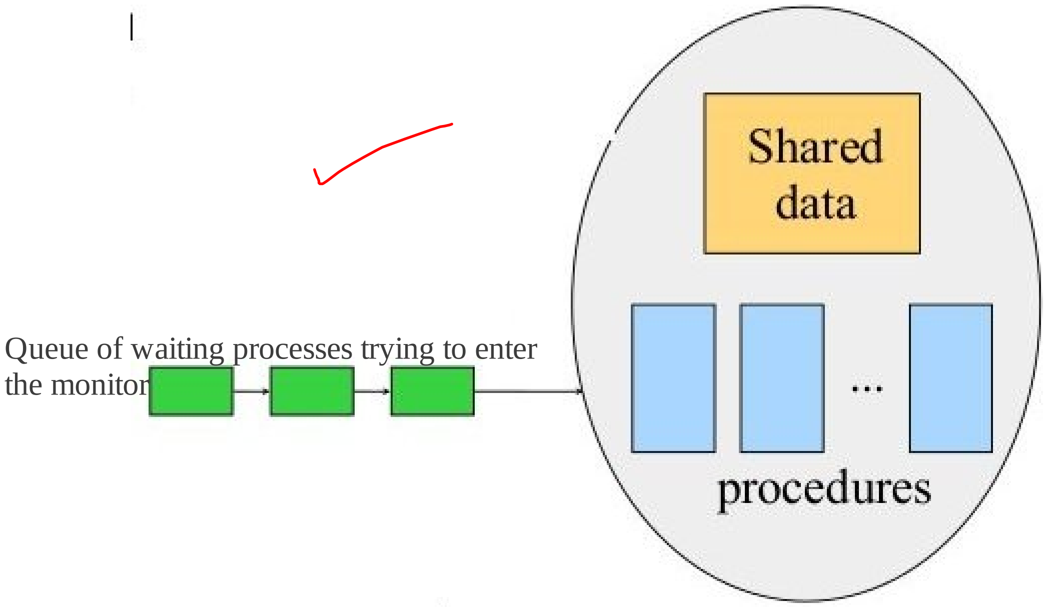
•can only be invoked by processes--not interrupt service routines because interrupt routines cannot block

•user controls synchronization--could mess up.

Monitors:

In concurrent programming, a **monitor** is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.



With semaphores IPC seems easy, but Suppose that the two downs in the producer's code were reversed in order, so mutex was decremented before empty instead of after it. If the buffer were completely full, the producer would block, with mutex set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock.

- A higher level synchronization primitive.
- 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.
- This rule, which is common in modern object-oriented languages such as Java, was relatively unusual for its time,
- Figure below illustrates a monitor written in an imaginary language, Pidgin Pascal.

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

Fig: A monitor

Message Passing:

Message passing in computer science, is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Message passing is a method of communication where messages are sent from a sender to one or more recipients. Forms of messages include **(remote) method invocation, signals, and data packets**. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
   send(destination, &message);
```

and

receive(source, &message);

Synchronous message passing systems requires the sender and receiver to wait for each other to transfer the message

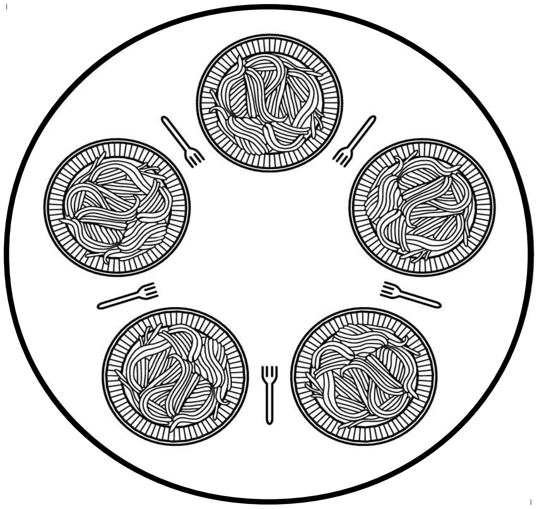
Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready.

Classical IPC Problems

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

1. Dining philosophers problems:
There are N philosphers sitting around a circular table eating spaghetti and discussing philosphy. 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



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

This solution is incorrect: it allows the system to reach deadlock. 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.

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, 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.

Solution:

```
#define N          5           /* 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 THINKING     0           /* philosopher is thinking */
#define HUNGRY       1           /* philosopher is trying to get forks */
#define EATING       2           /* 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 N1 */
{
   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 N1 */
{
   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 N1 */
{
   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 N1* /
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
         state[i] = EATING;
         up(&s[i]);
    }
}
```

Readers Writer problems:

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem which models access to a database (Courtois et al., 1971). Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other process may have access to the database, not even a reader. The question is how do you program the readers and the writers? One solution is shown below.

Solution to Readers Writer problems

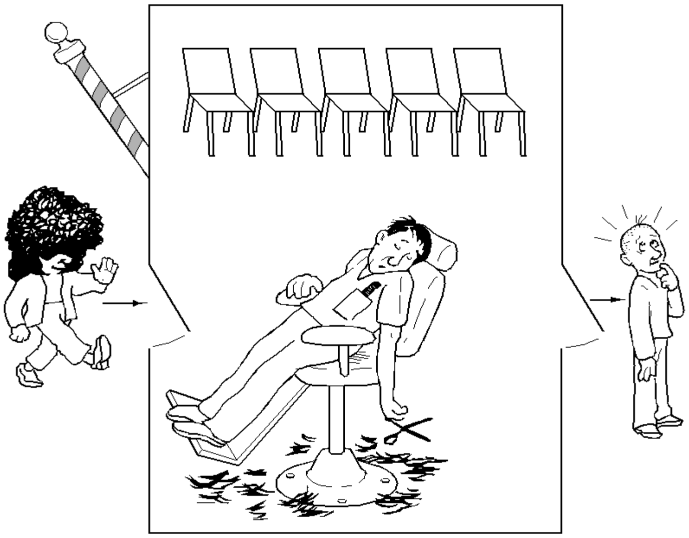
```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */
void reader(void)
{
    while (TRUE){                   /* repeat forever */
         down(&mutex);              /* get exclusive access to 'rc' */
         rc = rc + 1;               /* one reader more now */
         if (rc == 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 */
    }
}
```

In this solution, the first reader to get access to the data base does a down on the semaphore db. Subsequent readers merely have to increment a counter, rc. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

Sleeping Barber Problem

customers arrive to a barber, if there are no customers the barber sleeps in his chair. If the barber is asleep then the customers must wake him up.



The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.

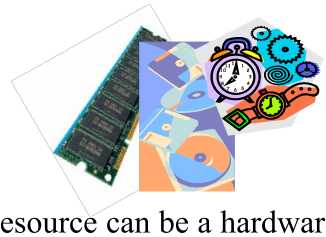
Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, then the customer leaves. Based on a naive analysis, the above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives. In practice, there are a

number of problems that can occur that are illustrative of general scheduling problems.
The problems are all related to the fact that the actions by both the barber and the customer (checking the waiting room, entering the shop, taking a waiting room chair, etc.) all take an unknown amount of time. For example, a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While he is on his way, the barber finishes the haircut he is doing and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to his chair and sleeps. The barber is now waiting for a customer and the customer is waiting for the barber. In another example, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

Solution:

Many possible solutions are available. The key element of each is a mutex, which ensures that only one of the participants can change state at once. The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair. A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair. This eliminates both of the problems mentioned in the previous section. A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room.
A multiple sleeping barbers problem has the additional complexity of coordinating several barbers among the waiting customers

Deadlock:



Resources

Resources are the passive entities needed by processes to do their work. A resource can be a hardware device (eg. a disk space) or a piece of information (a locked record in the database). Example of resources include CPU time, disk space, memory etc. There are two types of resources:
Preemptable – A Preemptable resources is one that can be taken away from the process owing it with no ill effect. Memory is an example of preemptable resources.

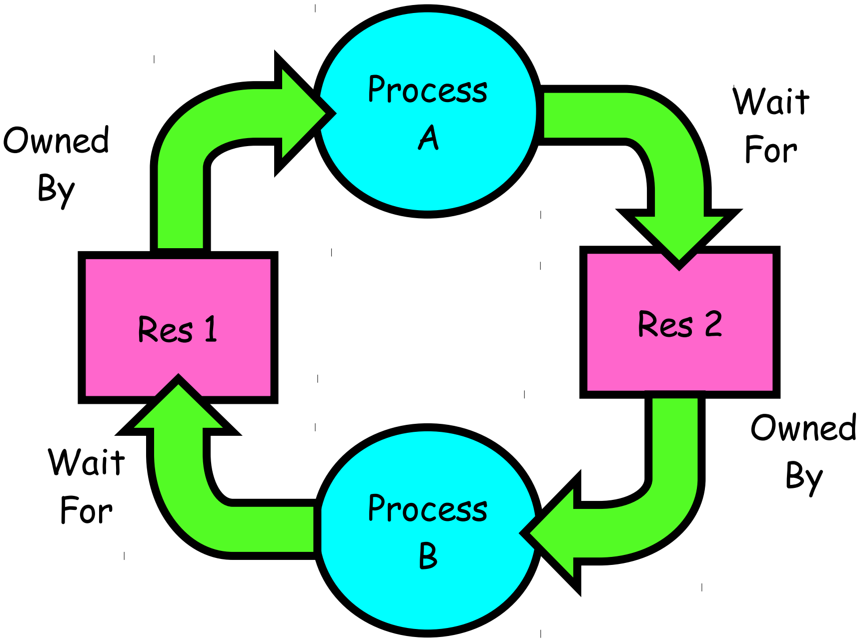
Non-preemptable – A non-preemptable resources in contrast is one that cannot be taken away from its current owner without causing the computation to fail. Examples are CD-recorder and Printers. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD. CD recorders are not preemptable at any arbitrary moment.

Read-only files are typically sharable. Printers are not sharable during time of printing .One of the major tasks of an operating system is to manage resources.
Deadlocks may occur when processes have been granted exclusive access to Resources. A resource may be a hardware device (eg. A tape drive) file or a piece of information (a locked record in a database). In general Deadlocks involves non preemptable resources. The sequence of events required to use a resource is:

1. Request the resource
2. Use the resource
3. Release the resource

What is Deadlock?

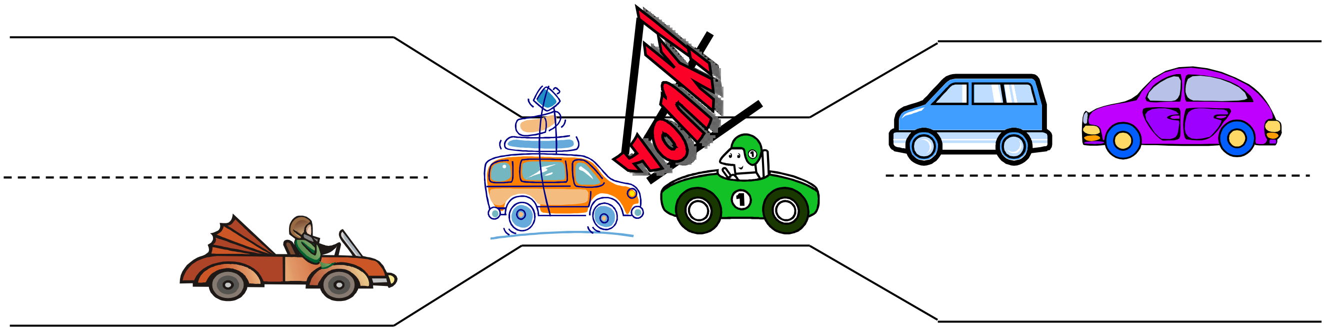
In Computer Science a set of process is said to be in deadlock if each process in the set is waiting for an event that only another process in the set can cause. Since all the processes are waiting, none of them will ever cause any of the events that would wake up any of the other members of the set & all the processes continue to wait forever.



Example 1:

- Two process A and B each want to recored a scanned document on a CD.
- A requests permission to use Scanner and is granted.
- B is programmed differently and requests the CD recorder first and is also granted.
- Now, A ask for the CD recorder, but the request is denied until B releases it. Unfortunately, instead of releasing the CD recorder B asks for Scanner. At this point both processes are blocked and will remain so forever. This situation is called Deadlock.

Example:2
Bridge Crossing Example:



Each segment of road can be viewed as a resource

- Car must own the segment under them
- Must acquire segment that they are moving into

For bridge: must acquire both halves

- Traffic only in one direction at a time
- Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next

If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)

- Several cars may have to be backed up

Starvation is possible

- East-going traffic really fast ==>no one goes west

Starvation vs. Deadlock

Starvation: thread waits indefinitely

Example, low-priority thread waiting for resources constantly in use by high-priority threads

Deadlock: circular waiting for resources

Thread A owns Res 1 and is waiting for Res 2 Thread B owns Res 2 and is waiting for Res 1

Deadlock ==> Starvation but not vice versa

Starvation can end (but doesn't have to)

Deadlock can't end without external intervention

Conditions for Deadlock:

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion
   Only one process at a time can use a resource.
2. Hold and wait
   Process holding at least one resource is waiting to acquire additional resources held by other processes.
3. No preemption
   Resources are released only voluntarily by the process holding the resource, after the process is finished with it
4. Circular wait
   There exists a set {P1 , …, Pn } of waiting processes.
   P1 is waiting for a resource that is held by P2
   P2 is waiting for a resource that is held by P3
   …
   Pn is waiting for a resource that is held by P1

All of these four conditions must be present for a deadlock to occur. If one or more of these conditions is absent, no Deadlock is possible.

Deadlock Modeling:

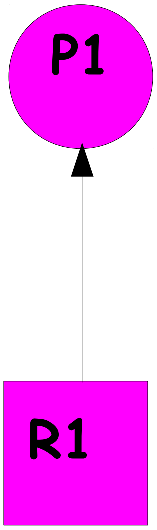
Deadlocks can be described more precisely in terms of
Resource allocation graph. Its a set of vertices V and a set of edges E.
V is partitioned into two types:

P = {P1, P 2, ..., P n}, the set consisting of all the processes in the system.

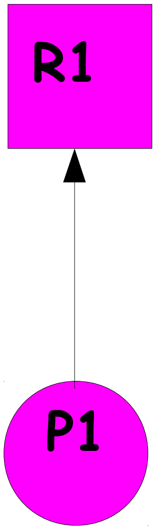
R = {R1, R2 , ..., Rm}, the set consisting of all resource types in the system.

request edge – directed edge Pi → Rj

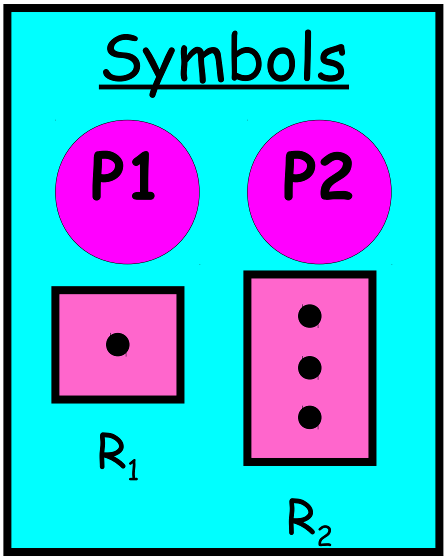
assignment edge – directed edge Rj → Pi

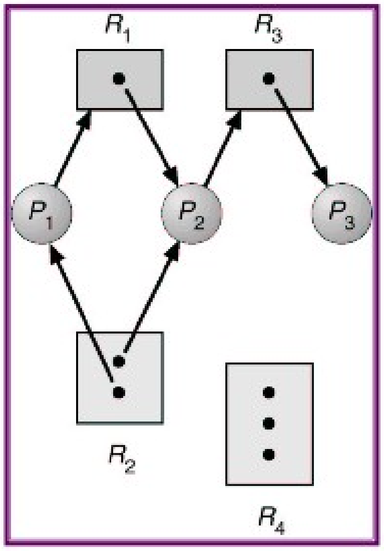


a). P1 is holding R1

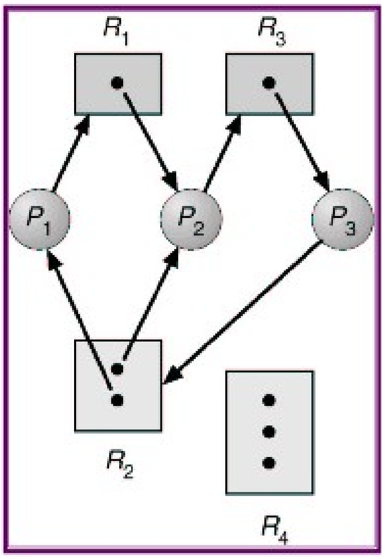


b). P1 requests R1

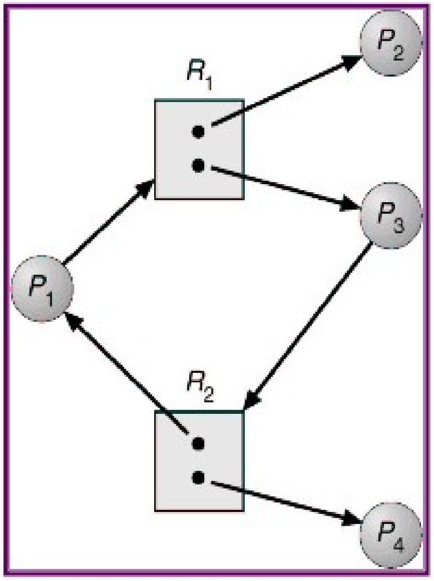




a).eg. of a Resource allocation graph



b).Resource allocation graph with Deadlock



c).Resource Allocation graph with a Cycle but no Deadlock

Basic Facts:
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

Methods for Handling Deadlock:

Allow system to enter deadlock and then recover

- Requires deadlock detection algorithm
- Some technique for forcibly preempting resources and/or terminating tasks

Ensure that system will never enter a deadlock

- Need to monitor all lock acquisitions
- Selectively deny those that might lead to deadlock

Ignore the problem and pretend that deadlocks never occur in the system

- Used by most operating systems, including UNIX

Deadlock Prevention

To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded. The methods for those conditions are as follows:

1. Mutual Exclusion:
In general, we do not have systems with all resources being sharable. Some resources like printers, processing units are non-sharable. So it is not possible to prevent deadlocks by denying mutual exclusion.

2. Hold and Wait:
One protocol to ensure that hold-and-wait condition never occurs says each process must request and get all of its resources before it begins execution.
Another protocol is “Each process can request resources only when it does not occupies any resources.”

The second protocol is better. However, both protocols cause low resource utilization and starvation. Many resources are allocated but most of them are unused for a long period of time. A process that requests several commonly used resources causes many others to wait indefinitely.

3. No Preemption:
One protocol is “If a process that is holding some resources requests another resource and that resource cannot be allocated to it, then it must release all resources that are currently allocated to it.”
Another protocol is “When a process requests some resources, if they are available, allocate them. If a resource it requested is not available, then we check whether it is being used or it is allocated to some other process waiting for other resources. If that resource is not being used, then the OS preempts it from the waiting process and allocate it to the requesting process. If that resource is used, the requesting process must wait.” This protocol can be applied to resources whose states can easily be saved and restored (registers, memory space). It cannot be applied to resources like printers.

4. Circular Wait:
One protocol to ensure that the circular wait condition never holds is “Impose a linear ordering of all resource types.” Then, each process can only request resources in an increasing order of priority.

For example, set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4. With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3.

Another protocol is “Whenever a process requests a resource rj, it must have released all resources rk with priority(rk) ≥ priority (rj).

Deadlock Avoidance:

Given some additional information on how each process will request resources, it is possible to construct an algorithm that will avoid deadlock states. The algorithm will dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources.

Two deadlock avoidance algorithms:

- resource-allocation graph algorithm
- Banker's algorithm

Resource-allocation graph algorithm

- only applicable when we only have 1 instance of each resource type
- claim edge (dotted edge), like a future request edge
- when a process requests a resource, the claim edge is converted to a request edge
- when a process releases a resource, the assignment edge is converted to a claim edge

Bankers Algorithms:

The Banker’s algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra . Resource allocation state is defined by the number of available and allocated resources and the maximum demand of the processes.

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

The system is in a safe state if there exists a safe sequence of all processes: such that they could be run even if all of them request their maximum needs all at once
Sequence < P1, P2, ... Pn > is safe for the current allocation state if, for each Pi, the resources which Pi can still request can be satisfied by

- the currently available resources plus
- the resources held by all of the Pj's, where j < i.

If the system is in a safe state, there can be no deadlock. If the system is in an unsafe state, there is the possibility of deadlock.
A state is safe if the system can allocate resources to each process in some order avoiding a deadlock. A deadlock state is an unsafe state.

Customer =Processes
Units = Resource say tape drive
Bankers=OS

The Banker algorithm does the simple task

– If granting the request leads to an unsafe state the request is denied.

– If granting the request leads to safe state the request is carried out.

Basic Facts:

- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

Bankers Algorithms for a single resource:

Customer	Used	Max
A	0	6
B	0	5
C	0	4
D	0	7

Available units: 10

In the above fig, we see four customers each of whom has been granted a certain no. of credit units (eg. 1 unit=1K dollar). The Banker reserved only 10 units rather than 22 units to service them since not all customer need their maximum credit immediately.

At a certain moment the situation becomes:

Customer	Used	Max
A	1	6
B	1	5
C	2	4
D	4	7

Available units: 2

Safe State:

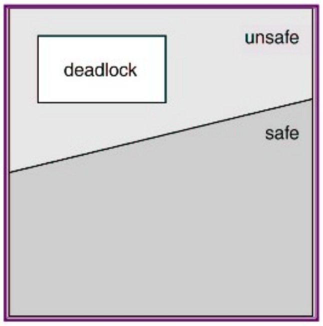
With 2 units left, the banker can delay any requests except C's, thus letting C finish and release all four of his resources. With four in hand, the banker can let either D or B have the necessary units & so on.

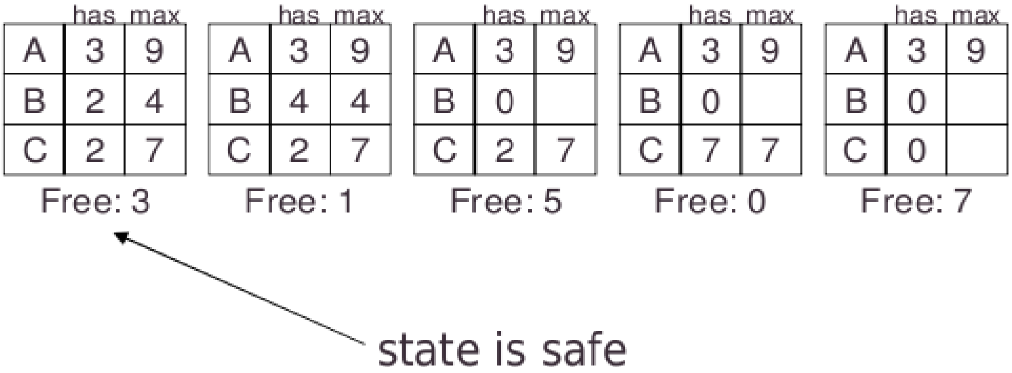
Unsafe State:

B requests one more unit and is granted.

Customer	Used	Max
A	1	6
B	2	5
C	2	4
D	4	7

Available units: 1
this is an unsafe condition. If all of the customer namely A, B,C & D asked for their maximum loans, then Banker couldn't satisfy any of them and we would have deadlock.It is important to note that an unsafe state does not imply the existence or even eventual existence of a deadlock. What an unsafe does imply is that some unfortunate sequence of events might lead a deadlock.





	has	max
A	3	9
B	2	4
C	2	7

Free: 3

	has	max
A	3	9
B	4	4
C	2	7

Free: 1

	has	max
A	3	9
B	0	
C	2	7

Free: 5

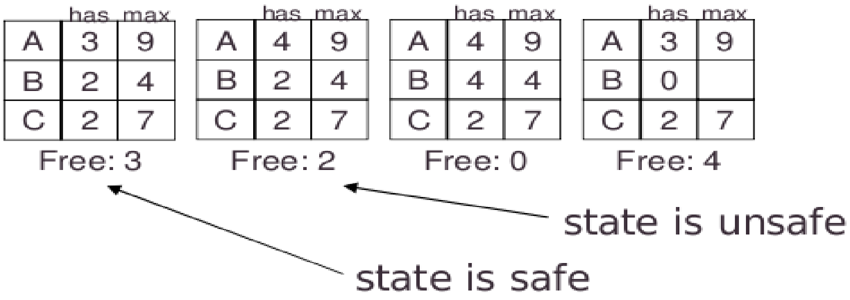
	has	max
A	3	9
B	0	
C	7	7

Free: 0

	has	max
A	3	9
B	0	
C	0	

Free: 7

state is safe



	has	max
A	3	9
B	2	4
C	2	7

Free: 3

	has	max
A	4	9
B	2	4
C	2	7

Free: 2

	has	max
A	4	9
B	4	4
C	2	7

Free: 0

	has	max
A	3	9
B	0	
C	2	7

Free: 4

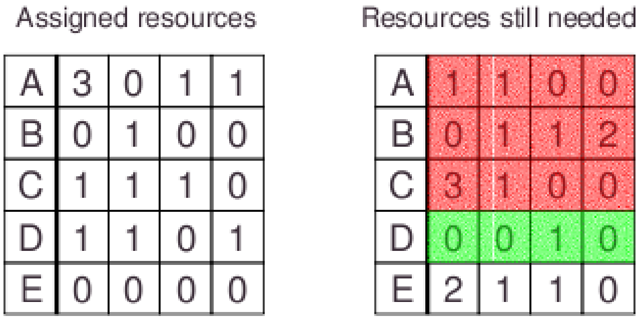
state is unsafe

state is safe

Bankers Algorithms for Multiple Resources:

The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion.
2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.



Assigned resources

A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources still needed

A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

a). Current Allocation Matrix   b). Request Matrix

E (Existing Resources): (6 3 4 2)
P (Processed Resources): (5 3 2 2)
A (Available Resources): (1 0 2 0)

Solution:
Process A, B & C can't run to completion since for Process for each process, Request is greater than Available Resources. Now process D can complete since its requests row is less than that of Available resources.

Step 1:
When D run to completion the total available resources is:
A = (1, 0, 2, 0) + (1, 1, 0 , 1)= (2, 1, 2, 1)

Now Process E can run to completion

Step 2:
Now process E can also run to completion & return back all of its resources.
==> A = (0 , 0, 0, 0) + (2, 1, 2, 1) = (2, 1, 2, 1)

Step 3:
Now process A can also run to completion leading A to
(3, 0, 1, 1) + (2, 1, 2, 1) = (5, 1, 3, 2)

Step 4:
Now process C can also run to completion leading A to
(5, 1, 3, 2) + (1, 1, 1, 0) = (6, 2, 4, 2)

Step 5:
Now process B can run to completion leading A to
(6, 2, 4, 2) + (0, 1, 0 , 0) = (6, 3, 4, 2)

This implies the state is safe and Dead lock free.

Questions:

A system has three processes and four allocable resources. The total four resource types exist in the amount as E= (4, 2, 3, 1). The current allocation matrix and request matrix are as follows: Using Bankers algorithm, explain if this state is deadlock safe or unsafe.

Current Allocation Matrix				
Process	Ro	R1	R2	R3
P0	0	0	1	0
P1	2	0	0	1
P1	0	1	2	0

Allocation Request Matrix				
Process	Ro	R1	R2	R3
P0	2	0	0	1
P1	1	0	1	0
P1	2	1	0	0

Q). Consider a system with five processes P0 through P4 and three resources types A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken

Process	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

1) What will be the content of the need Matrix?
2) Is the system in safe state? If yes, then what is the safe sequence?

1. Need [i,j]= Max [i,j] – Allocation[i,j]

content of Need Matrix is

A		B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

1. applying Safety algorithms

For Pi if $Need_i$ <= Available, then p_i is in Safe sequence,
Available = Available + $Allocation_i$

For P0, need0=7,4,3
Available = 3,3,2
==> Condition is false, So P0 must wait.

For P1 ,need1= 1,2,2
Available=3,3,2
need1< Available
So P1 will be kept in safe sequence. & Available will be updated as:
Available= 3,3,2 + 2,0,0 = 5,3,2

For P2, need2= 6,0,0
Available = 5,3,2
==> condition is again false, so P2 also have to wait.

For P3, need3= 0,1,1
Available= 5,3,2
==> condition is true , P3 will be in safe sequence.
Available = 5,3,2 + 2,1,1 = 7,4,3

For P4, need4= 4,3,1
Available = 7,4,3
==> condition $Need_i$ <= Available is true, so P4 will be in safe sequence
Available = 7,4,3 + 0,0,2 = 7,4,5

Now we have two processes P0 and P2 in waiting state. Either P0 or P1 can be choosen.
Let us take P2 whose need = 6,0,0
Available = 7,4,5

Since condition is true, P2 now comes in safe state leaving the
Available = 7,4,5 + 3,0,2 = 10, 4,7

Next P0 whose need = 7, 4, 3
Available = 10,4,7
since condition is true P0 also can be kept in safe state.
So system is in safe state & the safe sequence is <P1, P3, P4, P2, P0>

Detection and Recovery

A second technique is detection and recovery. When this technique is used, the system does not do anything except monitor the requests and releases of resources. Every time a resource is requested or released, the resource graph is updated, and a check is made to see if any cycles exist. If a cycle exists, one of the processes in the cycle is killed. If this does not break the deadlock, another process is killed, and so on until the cycle is broken.

The Ostrich Algorithm

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all.[] Different people react to this strategy in very different ways. Mathematicians find it completely unacceptable and say that deadlocks must be prevented at all costs.

Most operating systems, including UNIX, MINIX 3, and Windows, just ignore the problem on the assumption that most
users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything. If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom. Under these conditions, general solutions are hard to find.

Chapter 3: Kernel:

Introduction, Context switching (Kernel mode and User mode), First level interrupt handing, Kernel implementation of processes.

Introduction:

The central module of an operating system. It is the part of the operating system that loads first, and it remains in main memory. Because it stays in memory, it is important for the kernel to be as small as possible while still providing all the essential services required by other parts of the operating system and applications. Typically, the kernel is responsible for memory management, process and task management, and disk management.

Context Switch

A **context switch** is the computing process of storing and restoring the state (context) of a CPU so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU. The context switch is an essential feature of a multitasking operating system. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches. A context switch can mean a register context switch, a task context switch, a thread context switch, or a process context switch. What constitutes the context is determined by the processor and the operating system. Switching from one process to another requires a certain amount of time for doing the administration - saving and loading registers and memory maps, updating various tables and list etc.

process can run in two modes:

1.User Mode.
2.Kernel Mode.

1.User Mode:
=>A mode of the CPU when running a program.
=>In this mode ,the user process has no access to the memory locations used by the kernel.When a program is running in User Mode, it cannot directly access the kernel data structures or the kernel programs.

2.Kernel Mode:
=>A mode of the CPU when running a program. =>In this mode, it is the kernel that is running on behalf of the user process and directly access the kernel data structures or the kernel programs.Once the system call returns,the CPU switches back to user mode.

When you execute a C program,the CPU runs in user mode till the system call is invoked. In this mode,the user process has access to a limited section of the computer's memory and can execute a restricted set of machine instructions. however,when the process invokes a system call,the CPU switches from user mode to a more privileged mode the kernel. In this mode ,it is the kernel that runs on behalf of the user process,but it has access to any memory location and can execute any machine instruction. After the system call has returned,the CPU switches back to user mode.

Types Of Kernels

1 Monolithic Kernels

Earlier in this type of kernel architecture, all the basic system services like process and memory management, interrupt handling etc were packaged into a single module in kernel space. This type of architecture led to some serious drawbacks like 1) Size of kernel, which was huge. 2)Poor maintainability, which means bug fixing or addition of new features resulted in recompilation of the whole kernel which could consume hours .

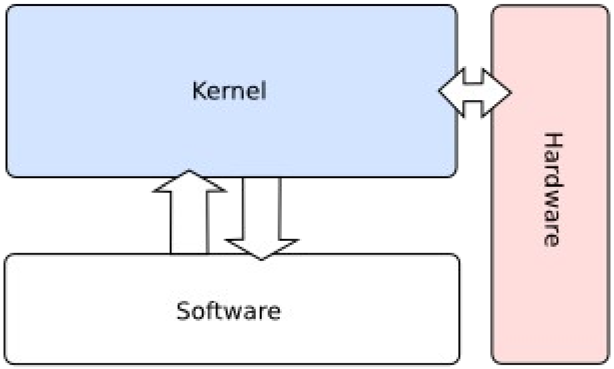


Fig: Monolithic Kernel

In a modern day approach to monolithic architecture, the kernel consists of different modules which can be dynamically loaded and un-loaded. This modular approach allows easy extension of OS's capabilities. With this approach, maintainability of kernel became very easy as only the concerned module needs to be loaded and unloaded every time there is a change or bug fix in a particular module. So, there is no need to bring down and recompile the whole kernel for a smallest bit of change. Also, stripping of kernel for various platforms (say for embedded devices etc) became very easy as we can easily unload the module that we do not want.

Examples:

- Linux
- Windows 9x (95, 98, Me)
- Mac OS <= 8.6
- BSD

2 Microkernels

A Microkernel tries to run most services - like networking, filesystem, etc. - as daemons / servers in user space. All that's left to do for the kernel are basic services, like memory allocation (however, the actual memory manager is implemented in userspace), scheduling, and messaging (Inter Process Communication).

This architecture majorly caters to the problem of ever growing size of kernel code which we could not control in the monolithic approach. This architecture allows some basic services like device driver management, protocol stack, file system etc to run in user space. This reduces the kernel code size and also increases the security and stability of OS as we have the bare minimum code running in kernel. So, if suppose a basic service like network service crashes due to buffer overflow, then only the networking service's memory would be corrupted, leaving the rest of the system still functional.

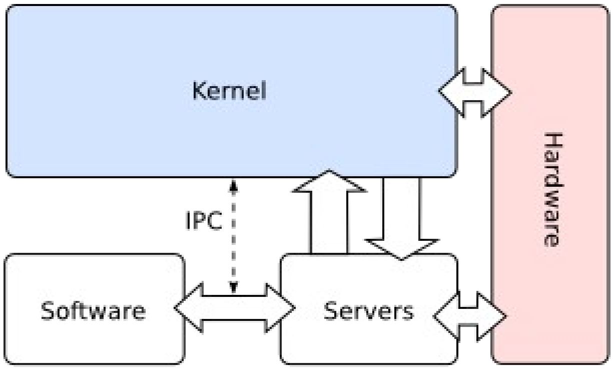


Fig: Microkernel

In this architecture, all the basic OS services which are made part of user space are made to run as servers which are used by other programs in the system through inter process communication (IPC). eg: we have servers for device drivers, network protocol stacks, file systems, graphics, etc. Microkernel servers are essentially daemon programs like any others, except that the kernel grants some of them privileges to interact with parts of physical memory that are otherwise off limits to most programs. This allows some servers, particularly device drivers, to interact directly with hardware. These servers are started at the system start-up.

So, what the bare minimum that microKernel architecture recommends in kernel space?

* Managing memory protection
* Process scheduling
* Inter Process communication (IPC)

Apart from the above, all other basic services can be made part of user space and can be run in the form of servers.

Examples:

- QNX
- L4
- AmigaOS
- Minix

QNX follows the Microkernel approach

3. Exo-kernel:

Exokernels are an attempt to separate security from abstraction, making non-overrideable parts of the operating system do next to nothing but securely multiplex the hardware. The goal is to avoid forcing any particular abstraction upon applications, instead allowing them to use or implement whatever abstractions are best suited to their task without having to layer them on top of other abstractions which may impose limits or unnecessary overhead. This is done by moving abstractions into untrusted user-space libraries called "library operating systems" (libOSes), which are linked to applications and call the operating system on their behalf.

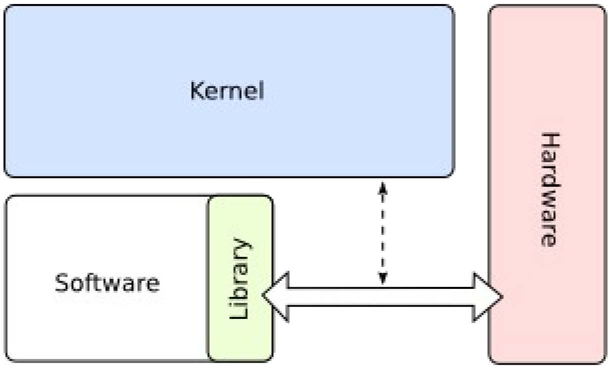


Fig: Exokernel

Interrupt Handler:

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in operating system or device driver whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated and the speed at which the interrupt handler completes its task.

First Level Interrupt Handler (FLIH)

- save registers of current process in PCB
- Determine the source of interrupt
- Initiate service of interrupt - calls interrupt handler
  Hardware dependent - implemented in assembler

In several operating systems - Linux, Unix, Mac OS X, Microsoft Windows, and some other operating systems in the past, interrupt handlers are divided into two parts: the First-Level Interrupt Handler (FLIH) and the Second-Level Interrupt Handlers (SLIH). FLIHs are also known as hard interrupt handlers or fast interrupt handlers, and SLIHs are also known as slow/soft interrupt handlers, Deferred Procedure Call.

A FLIH implements at minimum platform-specific interrupt handling similarly to interrupt routines. In response to an interrupt, there is a context switch, and the code for the interrupt is loaded and executed. The job of a FLIH is to quickly service the interrupt, or to record platform-specific critical information which is only available at the time of the interrupt, and schedule the execution of a SLIH for further long-lived interrupt handling.

Introduction: Scheduling levels, Scheduling objectives and criteria, Quantum size, Policy versus Mechanism in Scheduling, Preemptive versus No preemptive Scheduling, Scheduling techniques: Priority scheduling, deadline scheduling, First-In – First – Out scheduling, Round Robin Scheduling, Shortest – Job – First (SJF) Scheduling, Shortest – Remaining – Time (SRT) scheduling, Highest – Response – Ration – Next (HRN) scheduling Multilevel Feedback Queues.

Chapter 4: Scheduling:

Introduction:

In a multiprogramming system, frequently multiple process competes for the CPU at the same time. When two or more process are simultaneously in the ready state a choice has to be made which process is to run next. This part of the OS is called Scheduler and the algorithm is called scheduling algorithm. Process execution consists of cycles of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst that is followed by I/O burst, which is followed by another CPU burst then another i/o burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution.

Long term and short term scheduling:

If we consider batch systems, there will often be more processes submitted than the number of processes that can be executed immediately. So incoming processes are spooled (to a disk).

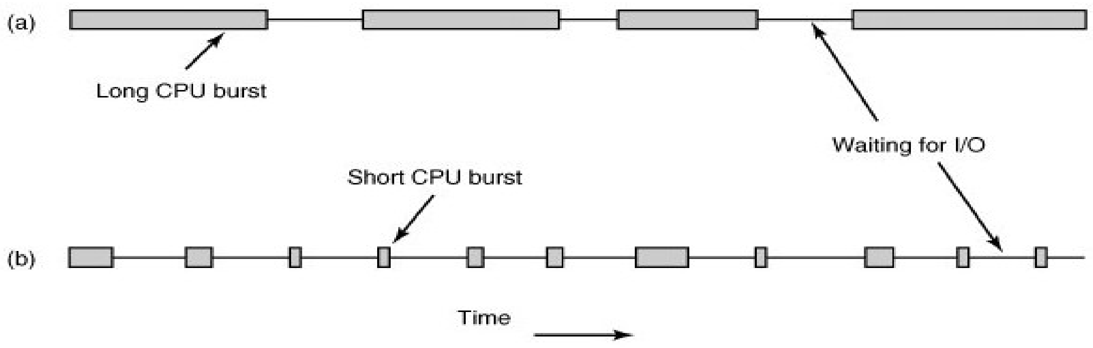
The long-term scheduler selects processes from this process pool and loads selected processes into memory for execution.

The short-term scheduler selects the process to get the processor from among the processes which are already in memory.

The short-time scheduler will be executing frequently (mostly at least once every 10 milliseconds). So it has to be very fast in order to achieve a better processor utilization. Time-sharing systems (mostly) have no long-term scheduler. The stability of these systems either depends upon a physical limitation (number of available terminals) or the self-adjusting nature of users (if you can't get response, you quit). It can sometimes be good to reduce the degree of multiprogramming by removing processes from memory and storing them on disk. These processes can then be reintroduced into memory by the medium-term scheduler. This operation is also known as swapping. Swapping may be necessary to improve the process mix or to free memory.

Process Behavior:

Fig: . Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O bound process



CPU Bound(or compute bound): Some process such as the one shown fig a. above spend most of their time computing. These processes tend to have long CPU burst and thus infrequent I/O waits. example: Matrix multiplication
I/O Bound: Some process such as the one shown in fig. b above spend most of their time waiting for I/O. They have short CPU bursts and thus frequent I/O waits. example: Firefox

Scheduling Criteria:

Many criteria have been suggested for comparison of CPU scheduling algorithms.

CPU utilization: we have to keep the CPU as busy as possible. It may range from 0 to 100%. In a real system it should range from 40 – 90 % for lightly and heavily loaded system.

Throughput: It is the measure of work in terms of number of process completed per unit time. For long process this rate may be 1 process per hour, for short transaction, throughput may be 10 process per second.

Turnaround Time: It is the sum of time periods spent in waiting to get into memory, waiting in ready queue, execution on the CPU and doing I/O. The interval form the time of submission of a process to the time of completion is the turnaround time. Waiting time plus the service time.
Turnaround time= Time of completion of job - Time of submission of job. (waiting time + service time or burst time)

Waiting time: its the sum of periods waiting in the ready queue.

Response time: in interactive system the turnaround time is not the best criteria. Response time is the amount of time it takes to start responding, not the time taken to output that response.

Types of Scheduling:

Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.
1.Preemptive Scheduling
2.Non preemptive Scheduling
Nonpreemptive scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even it runs for hours, it will not be forceably suspended. In effect no scheduling decisions are made during clock interrupts.
In contrasts, a **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 peemptive scheduling requires having a clock interrupt occur at the end of time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is only the option.

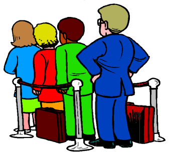
CPU scheduling decisions may take place under the following four circumstances:

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

Scheduling Algorithms:

1. First come First Serve:

FCFS is the simplest **non-preemptive** algorithm. Processes are assigned the CPU in the order they request it. That is the process that requests the CPU first is allocated the CPU first. The implementation of FCFS is policy is managed with a FIFO(First in first out) queue. When the first job enters the system from the outside in the morning, it is started immediately and allowed to run as long as it wants to. As other jobs come in, they are put onto the end of the queue. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.



Advantages:

1.Easy to understand and program. With this algorithm a single linked list keeps track of all ready processes.
2.Equally fair.,
3.Suitable specially for Batch Operating system.

Disadvantages:

1.FCFS is not suitable for time-sharing systems where it is important that each user should get the CPU for an equal amount of arrival time.

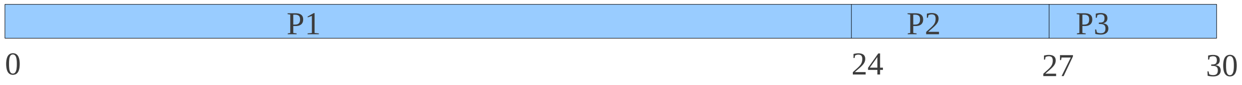
Consider the following set of processes having their burst time mentioned in milliseconds. CPU burst time indicates that for how much time, the process needs the cpu.

Process	Burst Time
P1	24
P2	3
P3	3

Calculate the average waiting time if the processes arrive in the order of:
a). P1, P2, P3

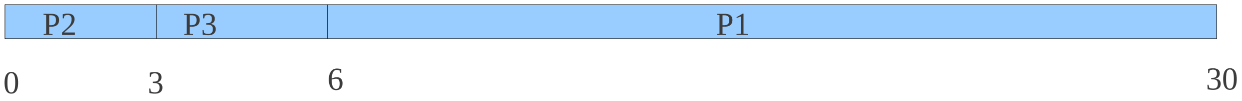
b). P2, P3, P1

a. The processes arrive the order P1, P2, P3. Let us assume they arrive in the same time at 0 ms in the system. We get the following gantt chart.



Waiting time for P1= 0ms , for P2 = 24 ms for P3 = 27ms
Avg waiting time: (0+24+27)/3= 17

b.) If the process arrive in the order P2,P3, P1



Average waiting time: (0+3+6)/3=3. Average waiting time vary substantially if the process CPU burst time vary greatly.

2. Shortest Job First:

When several equally important jobs are sitting in the i/p queue waiting to be started, the scheduler picks the shortest jobs first.

8	4	4	4
A	B	C	D

Original order: (Turn Around time)

Here we have four jobs A,B,C ,D with run times of 8 , 4, 4 and 4 minutes respectively. By running them in that order the turnaround time for A is 8 minutes, for B 12 minutes, for C 16 minutes and for D 20 minutes for an average of 14 minutes.
Now let us consider running these jobs in the shortest Job First.

B C D and then A.

the turnaround times are now , 4, 8, 12 and 20 minutes giving the average of 11. Shortest job first is probably optimal.

Consider the four jobs with run times of a, b,c,d. The first job finished at a, the second at a+b and so on. So the mean turnaround time is (4a+3b+2c+d)/4. It is clear that the a contributes more to the average than any other. So it should be the shortest one.

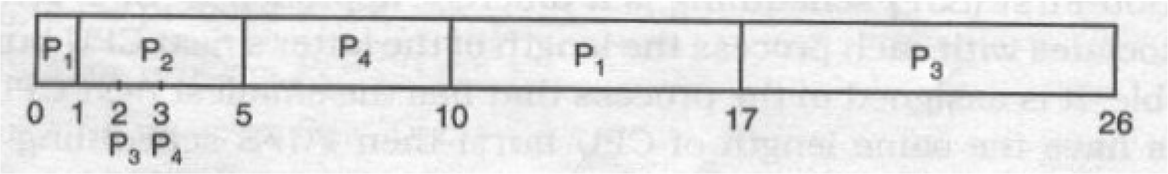
The disadvantages of this algorithm is the problem to know the length of time for which CPU is needed by a process. The SJF is optimal when all the jobs are available simultaneously.

The SJF is either preemptive or non preemptive. Preemptive SJF scheduling is sometimes called **Shortest Remaining Time First** scheduling. With this scheduling algorithms the scheduler always chooses the process whose remaining run time is shortest.
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 is started. This scheme allows new short jobs to get good service.

Q). Calculate the average waiting time in 1). Preemptive SJF and 2). Non Preemptive SJF
Note: SJF Default: (Non Preemptive)

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

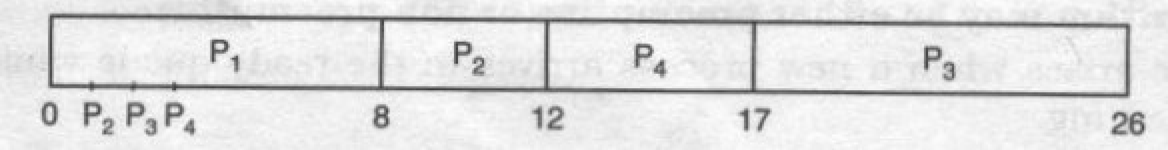
a. Preemptive SJF (Shortest Remaining Time First):



At t=0ms only one process P1 is in the system, whose burst time is 8ms; starts its execution. After 1ms i.e., at t=1, new process P2 (Burst time= 4ms) arrives in the ready queue. Since its burst time is less than the remaining burst time of P1 (7ms) P1 is preempted and execution of P2 is started.
Again at t=2, a new process P3 arrive in the ready queue but its burst time (9ms) is larger than remaining burst time of currently running process (P2 3ms). So P2 is not preempted and continues its execution. Again at t=3 , new process P4 (burst time 5ms) arrives . Again for same reason P2 is not preempted until its execution is completed.

Waiting time of P1: 0ms + (10 – 1)ms = 9ms
Waiting time of P2: 1ms – 1ms = 0ms
Waiting time of P3: 17ms – 2ms = 15ms
Waiting time of P4: 5ms – 3ms = 2ms
Avg waiting time: (9+0+15+2)/4 = 6.5ms

Non-preemptive SJF:



Since its non-preemptive process is not preempted until it finish its execution.
Waiting time for P1: 0ms
Waiting time for P2: (8-1)ms = 7ms
Waiting time for P3: (17 – 2) ms = 15ms
Waiting time for P4: (12 – 3)ms = 9ms

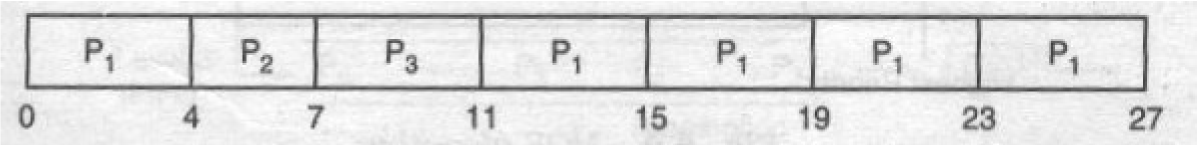
Average waiting time: (0+7+15+9)/4 = 7.75ms

3. Round-Robin Scheduling Algorithms:

•One of the oldest, simplest, fairest and most widely used algorithm 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.
•If the the process has blocked or finished before the quantum has elapsed the CPU switching is done.
•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 on quite long.
Consider the following set of processes that arrives at time 0 ms.

Process	Burst Time
P1	20
P2	3
P3	4

If we use time quantum of 4ms then calculate the average waiting time using R-R scheduling.



According to R-R scheduling processes are executed in FCFS order. So, firstly P1(burst time=20ms) is executed but after 4ms it is preempted and new process P2 (Burst time = 3ms) starts its execution whose execution is completed before the time quantum. Then next process P3 (Burst time=4ms) starts its execution and finally remaining part of P1 gets executed with time quantum of 4ms.

Waiting time of Process P1: 0ms + (11 – 4)ms = 7ms
Waiting time of Process P2: 4ms
Waiting time of Process P3: 7ms

Average Waiting time: (7+4+7)/3=6ms

4. Priority Scheduling:

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in the FCFS order.

Assigning priority:
1.To prevent high priority process from running indefinitely the scheduler may decrease the priority of the currently running process at each clock interrupt. If this causes its priority to drop below that of the next highest process, a process switch occurs.
2.Each process may be assigned a maximum time quantum that is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

Priorities can be assigned statically or dynamically. For UNIX system there is a command nice for assigning static priority.
It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class.

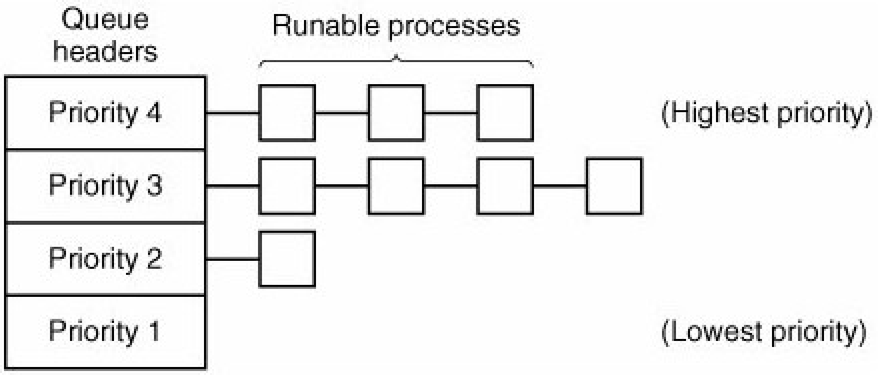
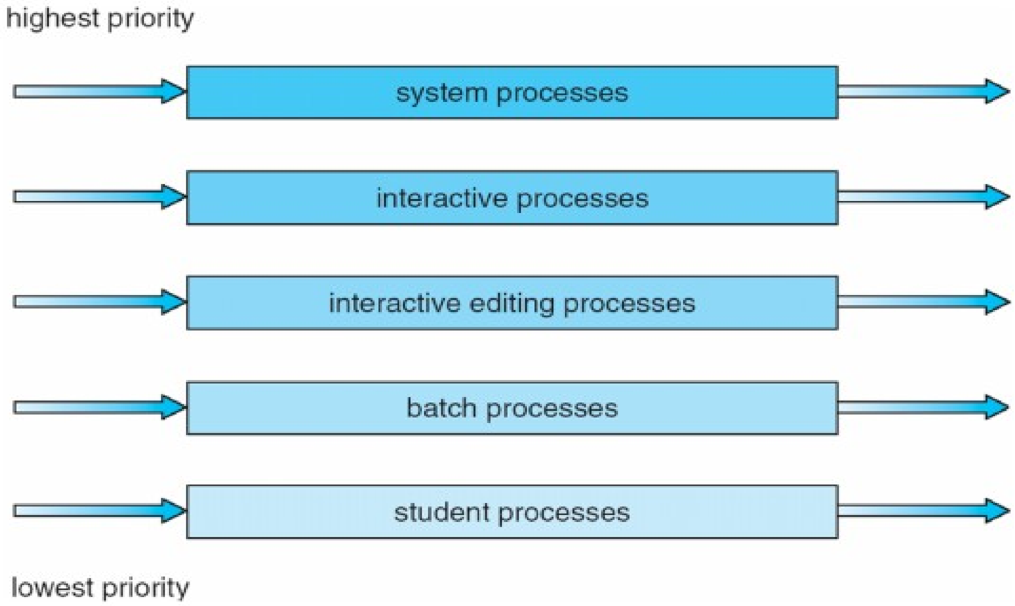


Fig: A scheduling algo. with four Priority classes

Problems in Priority Scheduling:
Starvation:
Low priority process may never execute.
Solution: **Aging**: As time progress increase the priority of Process.

Multilevel Queue Scheduling:

In this scheduling processes are classified into different groups. A common example may be foreground(or Interactive processes) or background (or batch processes).



Ready queue is partitioned into separate queues:
foreground (interactive)
background (batch)

Let us look at an example of a multilevel queue scheduling algorithm with five queues, listed below in the order of priority.
1.System processes
2.Interactive processes
3.Interactive editing processes
4.Batch processes
5.Student processes
Each queue has absolute priority over lower priority queues. No processes in the batch queue, for example could run unless the queue for System processes , interactive processes and interactive editing processes were all empty. If an interactive editing process enters the ready queue while a batch process was running the batch process would be preempted.
Another possibility is to time slice between the queues. For instance foreground queue can be given 80% of the CPU time for RR scheduling among its processes, whereas the background receives 20% of the CPU time.

Guaranteed Scheduling:

•Make real promises to the users about performance.
•If there are n users logged in while you are working, you will receive about 1 /n of the CPU power.
•Similarly, on a single-user system with n processes running, all things being equal, each one should get 1 /n of the CPU cycles.
•To make good on this promise, the system must keep track of how much CPU each process has had since its creation.
•CPU Time entitled= (Time Since Creation)/n
•Then compute the ratio of Actual CPU time consumed to the CPU time entitled.
•A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to.
•The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

Lottery Scheduling:

Lottery Scheduling is a probabilistic scheduling algorithm for processes in an operating system. Processes are each assigned some number of lottery tickets for various system resources such as CPU time.;and the scheduler draws a random ticket to select the next process. The distribution of tickets need not be uniform; granting a process more tickets provides it a relative higher chance of selection. This technique can be used to approximate other scheduling algorithms, such as Shortest job next and Fair-share scheduling.

Lottery scheduling solves the problem of starvation. Giving each process at least one lottery ticket guarantees that it has non-zero probability of being selected at each scheduling operation.

More important process can be given extra tickets to increase their odd of winning. If there are 100 tickets outstanding, & one process holds 20 of them it will have 20% chance of winning each lottery. In the long run it will get 20% of the CPU. A process holding a fraction f of the tickets will get about a fraction of the resource in questions.

Two-Level Scheduling:

Performs process scheduling that involves swapped out processes. Two-level scheduling is needed when memory is too small to hold all the ready processes .
Consider this problem: A system contains 50 running processes all with equal priority. However, the system's memory can only hold 10 processes in memory simultaneously. Therefore, there will always be 40 processes swapped out written on virtual memory on the hard disk
It uses two different schedulers, one lower-level scheduler which can only select among those processes in memory to run. That scheduler could be a Round-robin scheduler. The other scheduler is the higher-level scheduler whose only concern is to swap in and swap out processes from memory. It does its scheduling much less often than the lower-level scheduler since swapping takes so much time. the higher-level scheduler selects among those processes in memory that have run for a long time and swaps them out

Scheduling in Real Time System:

Time is crucial and plays an essential role.
eg. the computer in a compact disc player gets the bits as they come off the drive and must convert them into music with a very tight time interval. If the calculation takes too long the music sounds peculiar. Other example includes

- Auto pilot in Aircraft
- Robot control in automated factory.
- Patient monitoringin Factory. (ICU)

Two types:

1. **Hard Real Time system:** There are absolute deadline that must be met.
2. **Soft Real Time system:** Missing an occasional deadline is undesirable but nevertheless tolerable.

In both cases real time behavior is achieved by dividing the program into a no. of processes, each of whose behavior is predictable & known in advance. These processes are short lived and can run to completion. Its the job of schedulers to schedule the process in such a way that all deadlines are met.

If there are m periodic events and event i occurs with period Pi and requires Ci seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criteria is said to be schedulable.

Policy VS Mechanism:

The separation of mechanism and policy is a design principle in computer science. It states that mechanisms (those parts of a system implementation that control the authorization of operations and the allocation of resources) should not dictate (or overly restrict) the policies according to which decisions are made about which operations to authorize, and

- All the processes in the system belong to different users and are thus competing for the CPU.
- sometimes it happens that one process has many children running under its control. For example, a database management system process may have many children. Each child might be working on a different request, or each one might have some specific function to perform (query parsing, disk access, etc.)
- Unfortunately, none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.
- The solution to this problem is to separate the scheduling mechanism from the scheduling policy.
- What this means is that the scheduling algorithm is parameterized in some way, but the

parameters can be filled in by user processes.

- Let us consider the database example once again. Suppose that the kernel uses a priority scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its children. In this way the parent can control in detail how its children are scheduled, even though it does not do the scheduling itself. Here the mechanism is in the kernel but policy is set by a user process.

Chapter 5: Memory Management

Introduction, storage organization and hierarchy, contiguous versus noncontiguous storage allocation, Logical and physical memory, fragmentation, fixed partition multiprogramming, variable partition multiprogramming, relocation and protection, Coalescing and Compaction, Virtual Memory: Introduction, Paging, , Page tables, Block mapping, Direct mapping, TLB (Translation Look aside Buffers), Page Fault, Page Replacement algorithms, Optimal Page Replacement algorithm, Not Recently Used Page Replacement algorithm, First- In- First Out algorithm, Second Chance Page Replacement algorithm, Working Set Page Replacement algorithm, WS Clock Page Replacement algorithm, Segmentation, implementation of pure segmentation, Segmentation with Paging.

Types of Memory:

Primary Memory (eg. RAM)
Holds data and programs used by a process that is executing
Only type of memory that a CPU deals with
Secondary Memory (eg. hard disk)
Non-volatile memory used to store data when a process is not executing.

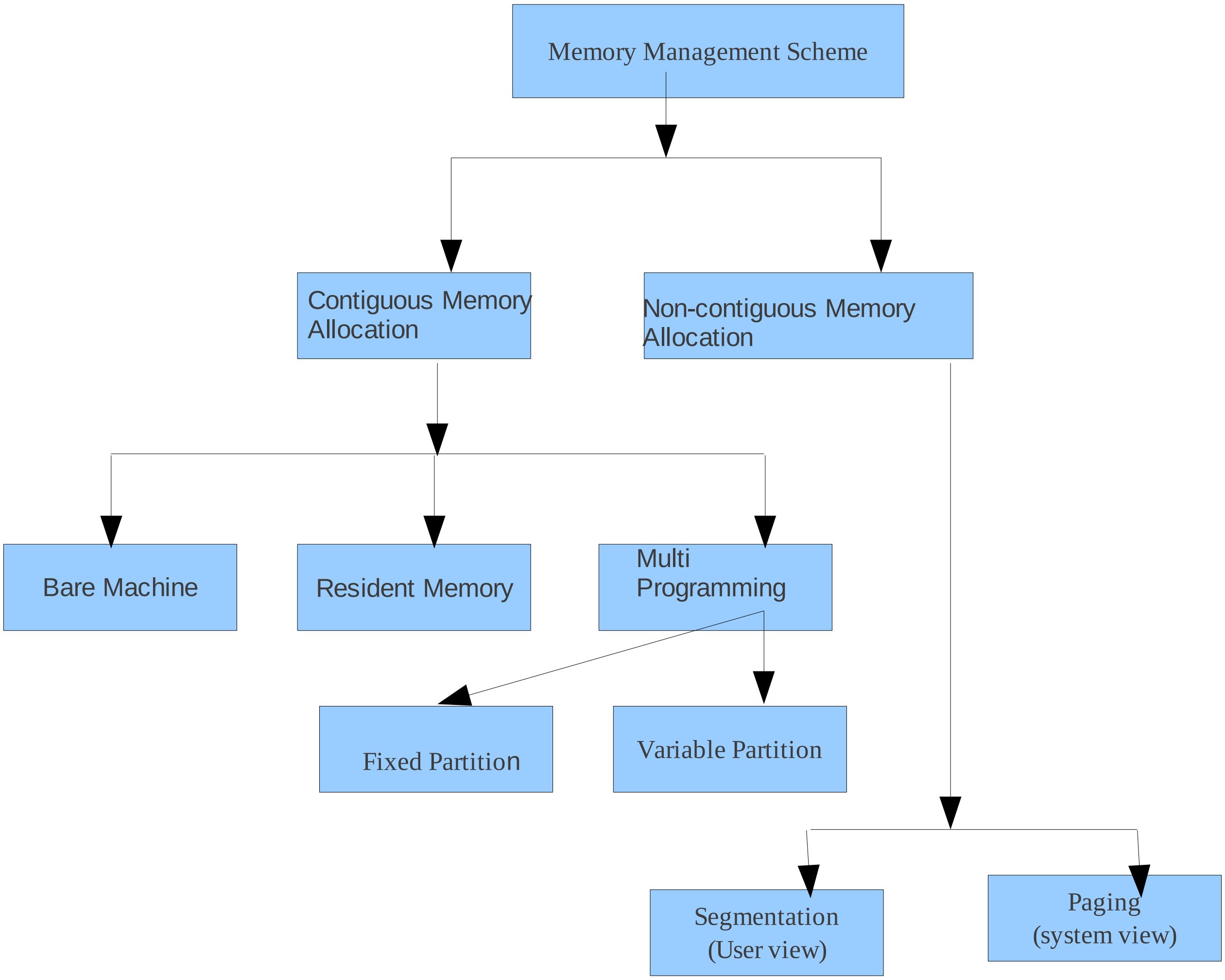


Fig:Types of Memory management

Memory Management:

Memory management is the act of managing computer memory. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed. The management of main memory is critical to the computer system.

In a uniprogramming system, Main memory is divided into two parts:
- one part for the OS
- one part for the program currently being executed.

In a multiprogramming system, the user part of the memory must be further subdivided to accommodate multiple processes. The task of subdivision is carried out dynamically by the Operating System and is known as Memory Management.

Two major schemes for memory management.

1.Contiguous Memory Allocation
2.Non-contiguous memory Allocation

Contiguous allocation

It means that each logical object is placed in a set of memory locations with strictly consecutive addresses.

Non-contiguous allocation

It implies that a single logical object may be placed in non-consecutive sets of memory locations. Paging (System view) and Segmentation (User view) are the two mechanisms that are used to mange non-contiguous memory allocation.

Memory Partitioning:

1. Fixed Partitioning:

Main memory is divided into a no. of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.
Memory Manager will allocate a region to a process that best fits it

Unused memory within an allocated partition called internal fragmentation

Advantages:
Simple to implement
Little OS overhead

Disadvantages:
Inefficient use of Memory due to internal fragmentation. Main memory utilization is extremely

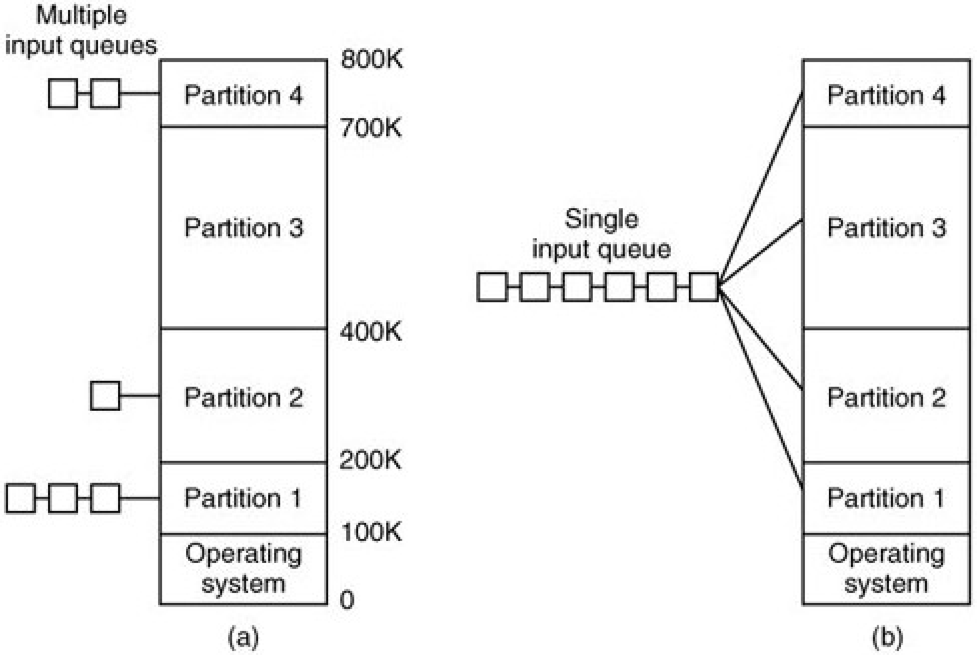
inefficient. Any program, no matter how small, occupies an entire partition. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as **internal fragmentation.**

Two possibilities:

a). Equal size partitioning

b). Unequal size Partition

Not suitable for systems in which process memory requirements not known ahead of time; i.e. timesharing systems.



(a) Fixed memory partitions with separate input queues for each partition.
(b) Fixed memory partitions with a single input queue.

When the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3. Here small jobs have to wait to get into memory, even though plenty of memory is free

An alternative organization is to maintain a single queue as in Fig. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

2.Dynamic/Variable Partitioning:

To overcome some of the difficulties with fixed partitioning, an approach known as dynamic partitioning was developed . The partitions are of variable length and number. When a process is brought into main memory, it is allocated exactly as much memory as it requires and no more. An example, using 64 Mbytes of main memory, is shown in Figure

Eventually it leads to a situation in which there are a lot of small holes in memory. As time goes on,

memory
becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as **external fragmentation**, indicating that the memory that is external to all partitions becomes increasingly fragmented.
One technique for overcoming external fragmentation is compaction: From time to time, the operating system shifts the processes so that they are contiguous and so that all of the free memory is together in one block. For example, in Figure h, compaction will result in a block of free memory of length 16M. This may well be sufficient to load in an additional process. The difficulty with compaction is that it is a time consuming procedure and wasteful of processor time.

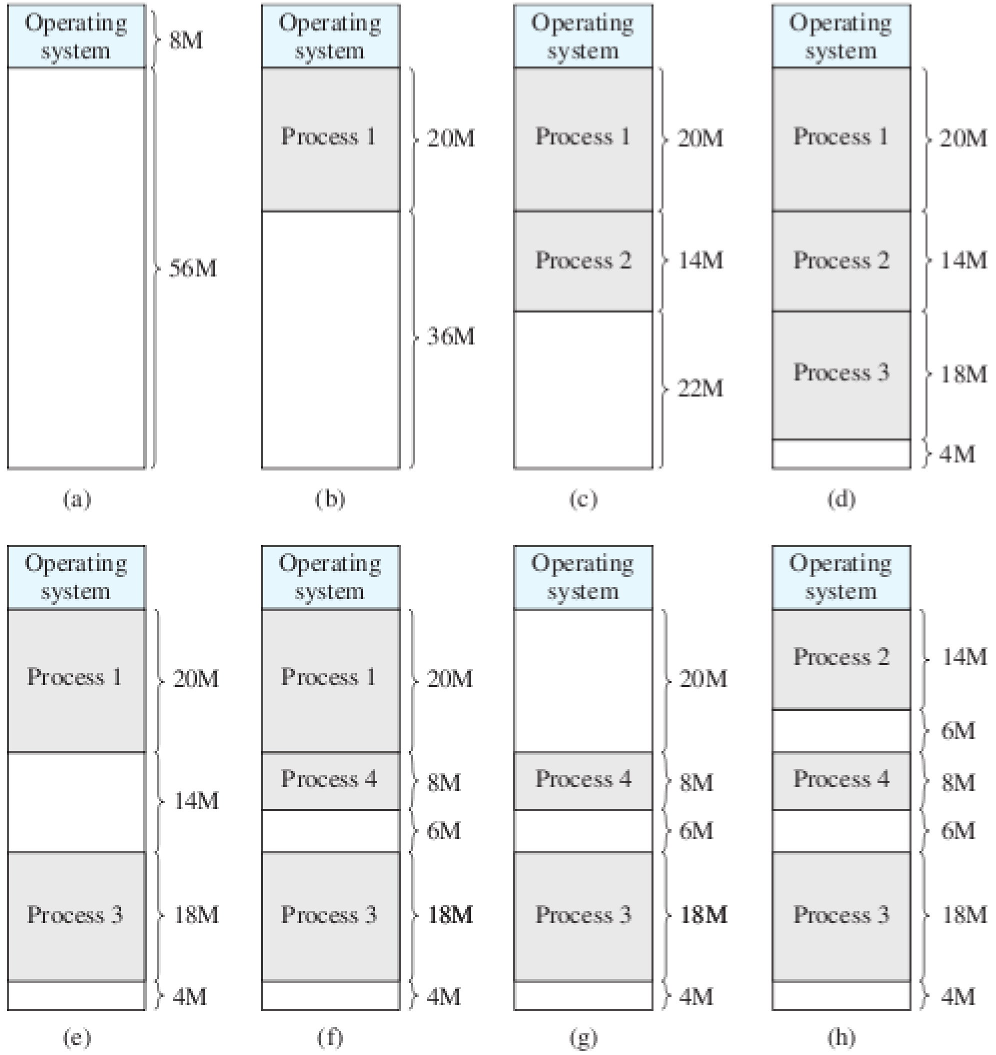


Fig.The Effect of dynamic partitioning

Memory Management with Bitmaps:

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 below shows part of memory and the corresponding bitmap.

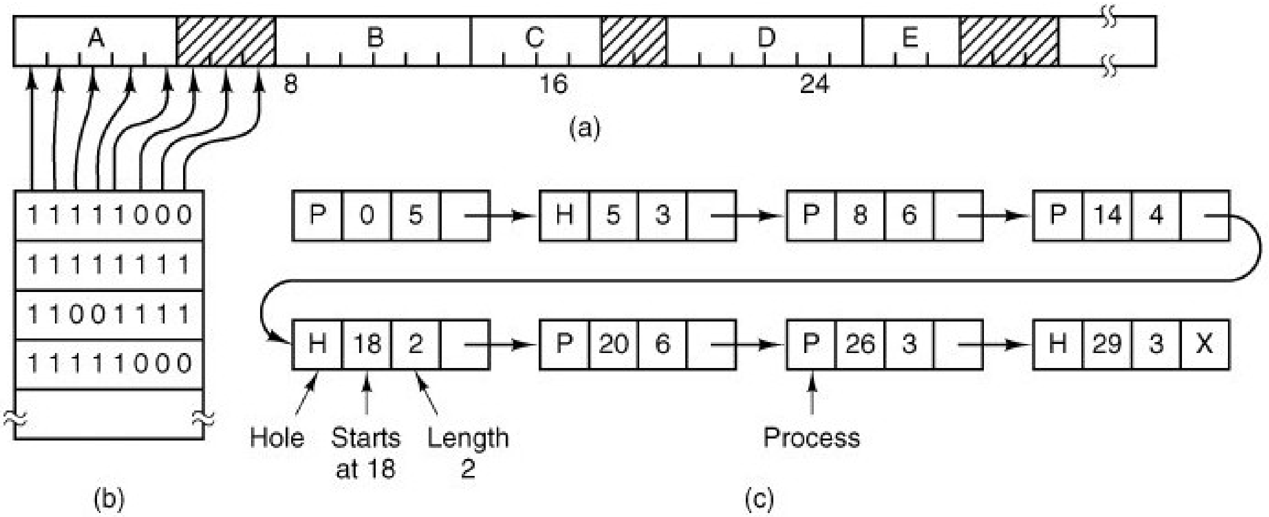


Fig:(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the 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.

Memory Management with Linked Lists

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.

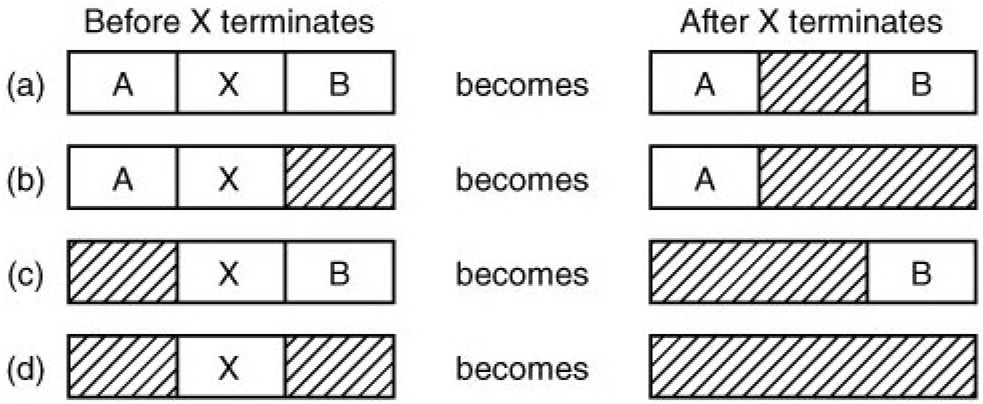


Fig:Four neighbor combinations for the terminating process, X.

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 very bottom of memory). These may be either processes or holes, leading to the four combinations shown in fig above.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate.

First Fit: The simplest algorithm is first fit. The process 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.

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.

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.

Worst Fit: Always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

Quick Fit: maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of say, 21 KB, could either be put on the 20-KB list or on a special list of odd-sized holes. 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 neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

Buddy-system:

Both fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system .

In a buddy system, memory blocks are available of size 2^K words, $L \leq K \leq U$, where
2^L = smallest size block that is allocated
2^U=largest size block that is allocated; generally 2U is the size of the entire memory available for allocation

In a buddy system, the entire memory space available for allocation is initially treated as a single block whose size is a power of 2. When the first request is made, if its size is greater than half of the initial block then the entire block is allocated. Otherwise, the block is split in two equal companion buddies. If the size of the request is greater than half of one of the buddies, then allocate one to it. Otherwise, one of the buddies is split in half again. This method continues until the smallest block greater than or equal to the size of the request is found and allocated to it. In this method, when a process terminates the buddy block that was allocated to it is freed. Whenever possible, an unallocated buddy is merged with a companion buddy in order to form a larger free block. Two blocks are said to be companion buddies if they resulted from the split of the same direct parent block.

The following fig. illustrates the buddy system at work, considering a 1024k (1-megabyte) initial block and the process requests as shown at the left of the table.

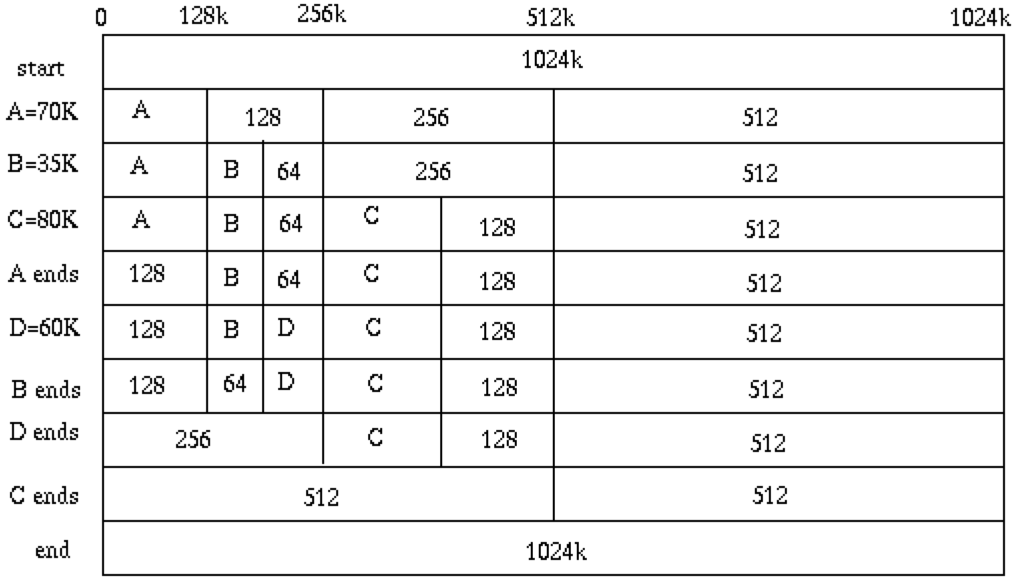


Fig: Example of buddy system

Swapping:

A process must be in memory to be executed. A process, however can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. For example assume a multiprogramming environment with a Round Robin CPU scheduling algorithms. When a quantum expires, the memory manger will start to swap out the process that just finished and to swap another process into the memory space that has been freed.

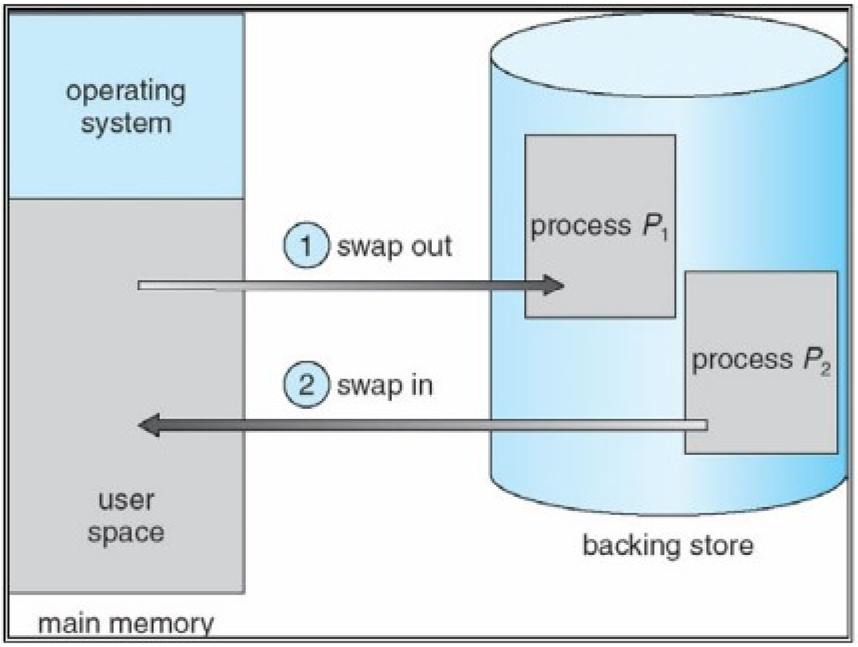


Fig: Swapping of two processes using a disk as a backing store

Logical Address VS Physical Address:

An address generated by the CPU is commonly referred to as a l**ogical address**, whereas address seen by the memory unit- that is the one loaded into the memory-address register of the memory- is commonly referred to as a **physical address**.

The compile time and load time address binding methods generate identical logical and physical addresses. However the execution time address binding scheme results in differing logical and physical address. In that case we usually refer to the logical address as Virtual address.

The run time mapping from virtual address to physical address is done by a hardware called **Memory management unit (MMU).**

Set of all logical address space generated by a program is known as l**ogical address space** and the set of all physical addresses which corresponds to these logical addresses is called **physical address space**.

Non-contiguous Memory allocation:

Fragmentation is a main problem is contiguous memory allocation. We have seen a method called compaction to resolve this problem. Since its an I/O operation system efficiency gets reduced. So, a better method to overcome the fragmentation problem is to make our logical address space non-contiguous.

Consider a system in which before applying compaction, there are holes of size 1K and 2K. If a new process of size 3K wants to be executed then its execution is not possible without compaction. An alternative approach is divide the size of new process P into two chunks of 1K and 2K to be able to load them into two holes at different places.

1. If the chunks have to be of same size for all processes ready for the execution then the memory management scheme is
called **PAGING.**
2. If the chunks have to be of different size in which process image is divided into logical segments of different sizes then this method is called **SEGMENTATION.**
3. If the method can work with only some chunks in the main memory and the remaining on the disk which can be brought into main memory only when its required, then the system is called **VIRTUAL MEORY MANAGEMENT SYSTEM.**

Virtual Memory:

The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 512-MB program can run on a 256-MB machine by carefully choosing which 256 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.
Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process, the same way as in any other multiprogramming system.

Virtual memory systems separate the memory addresses used by a process from actual physical addresses, allowing separation of processes and increasing the effectively available amount of RAM using disk swapping.

Paging:

Most virtual system uses a techniques called paging that permits the physical address space of a process to be non-contiguous. 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 Fig

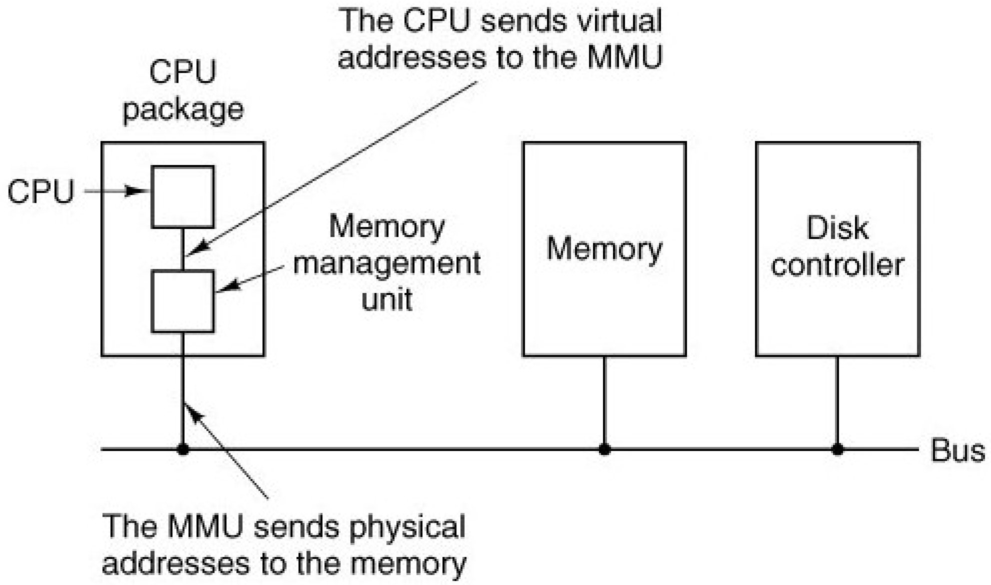

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Fig:The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays

The basic method for implementing paging involves breaking physical memory into fixed size block called **frames** and breaking logical memory into blocks of the same size called **pages.** size is power of 2, between 512 bytes and 8,192 bytes
When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-size block that are of the same size as memory frames.
Backing store(2) is typically part of a hard disk that is used by a paging or swapping system to store information not currently in main memory. Backing store is slower and cheaper than main memory.

A very simple example of how this mapping works is shown in Fig. below. In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses. This computer, however, has only 32 KB of physical memory, so although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run.

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.
When the program tries to access address 0, for example, using the instruction

MOV REG,0

virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, an instruction **MOV REG,8192** is effectively transformed into **MOV REG,24576.** because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

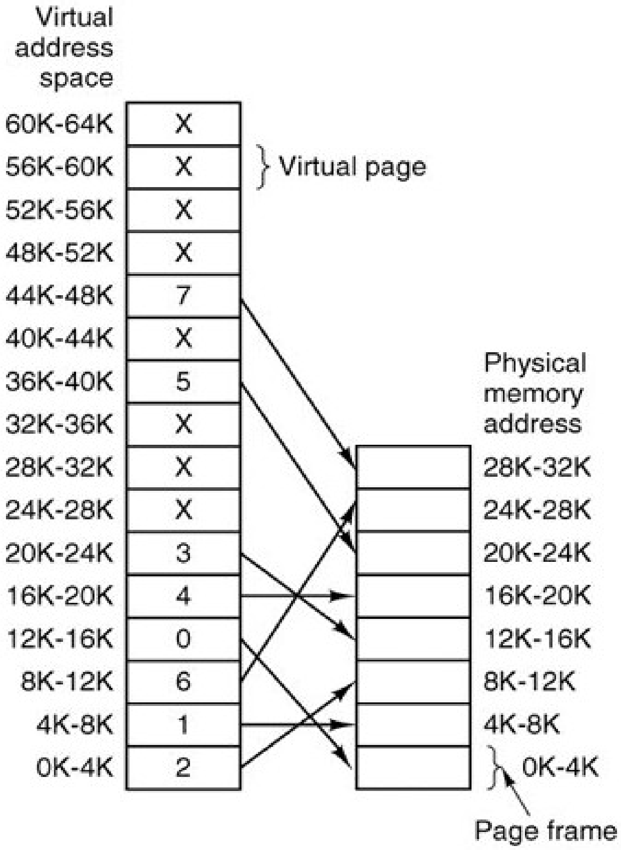


Fig:The relation between virtual addresses and physical memory addresses is given by the page table.

PAGE Fault:

A **page fault** is a trap to the software raised by the hardware when a program accesses a page that is mapped in the virtual address space, but not loaded in physical memory. In the typical case the operating system tries to handle the page fault by making the required page accessible at a location in physical memory or kills the program in the case of an illegal access. The hardware that detects a page fault is the memory management unit in a processor. The exception handling software that handles the page fault is generally part of the operating system.
What happens if the program tries to use an unmapped page, for example, by using the instruction

MOV REG,32780

which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

In computer storage technology, a page is a fixed-length block of memory that is used as a unit of transfer between physical memory and external storage like a disk, and a page fault is an interrupt (or exception) to the software raised by the hardware, when a program accesses a page that is mapped in address space, but not loaded in physical memory.
An interrupt that occurs when a program requests data that is not currently in real memory. The interrupt triggers the operating system to fetch the data from a virtual memory and load it into RAM.
An invalid page fault or page fault error occurs when the operating system cannot find the data in virtual memory. This usually happens when the virtual memory area, or the table that maps virtual addresses to real addresses, becomes corrupt.

Paging Hardware:

The hardware support for the paging is as shown in fig. Every address generated by the CPU is divided into two parts:
a page number(p) and a page offset (d)

Page number (p) – used as an index into a page table which contains base address of each page in physical memory.

Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit.

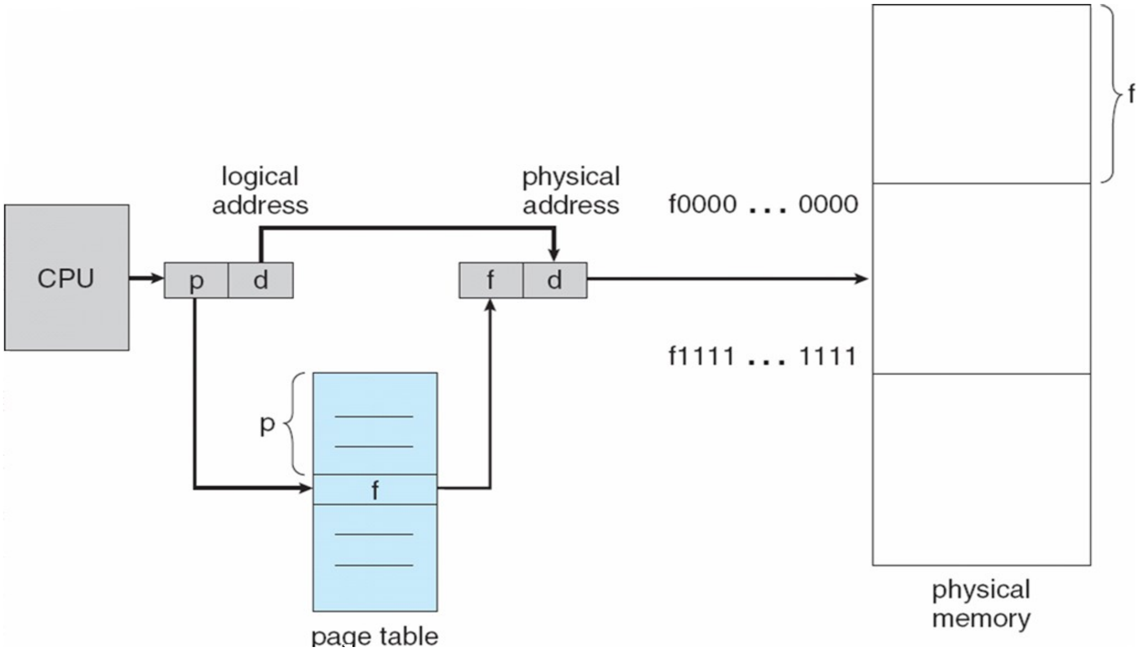


Fig:Paging Hardware

page number	page offset
p	d
$m - n$	n

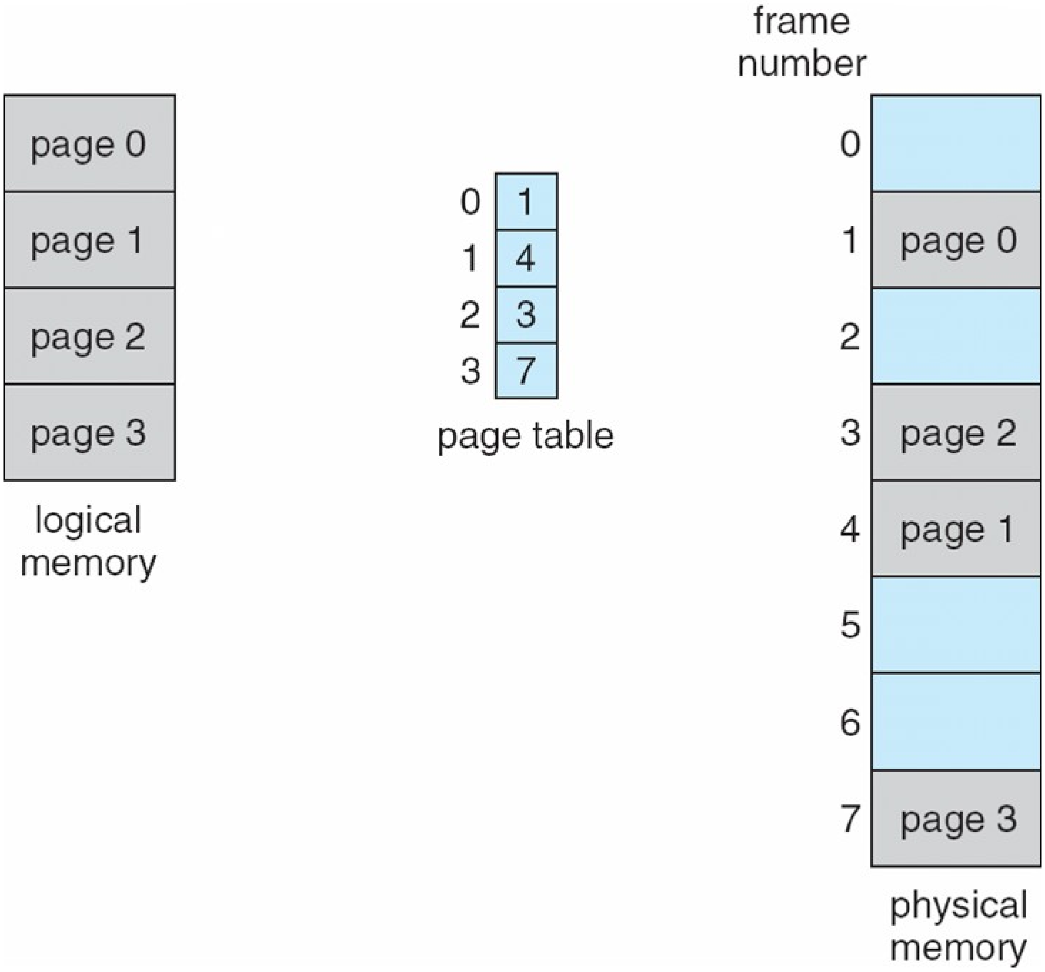


Fig:Paging model of Logical and Physical Memory

When we use a paging scheme, we have no external fragmentation. Any free frame can be allocated to a process that needs it. However we may have some internal fragmentation.

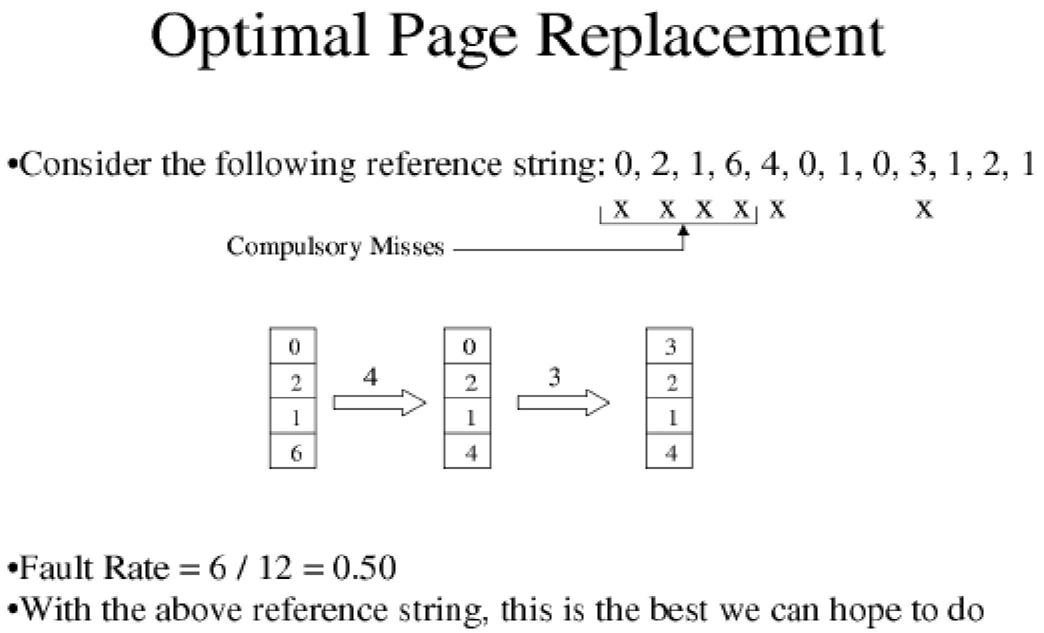
Page Replacement Algorithms:

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. The page replacement is done by swapping the required pages from backup storage to main memory and vice-versa. If the page 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. *A page replacement algorithm is evaluated by running the particular algorithm on a string of memory references and compute the page faults.Referenced string is a sequence of pages being referenced. Page fault is not an error.* Contrary to what their name might suggest, page faults are not errors and are common and necessary to increase the amount of memory available to programs in any operating system that utilizes virtual memory, including Microsoft Windows, Mac OS X, Linux and Unix.

Each operating system uses different page replacement algorithms. To select the particular algorithm, the algorithm with lowest page fault rate is considered.
1.Optimal page replacement algorithm
2.Not recently used page replacement
3.First-In, First-Out page replacement
4.Second chance page replacement
5.Clock page replacement
6.Least recently used page replacement

The Optimal Page Replacement Algorithm:

The algorithm has lowest page fault rate of all algorithm. This algorithm state that: Replace the page which will not be used for longest period of time i.e future knowledge of reference string is required.
•Often called Balady's Min Basic idea: Replace the page that will not be referenced for the longest time.
•Impossible to implement



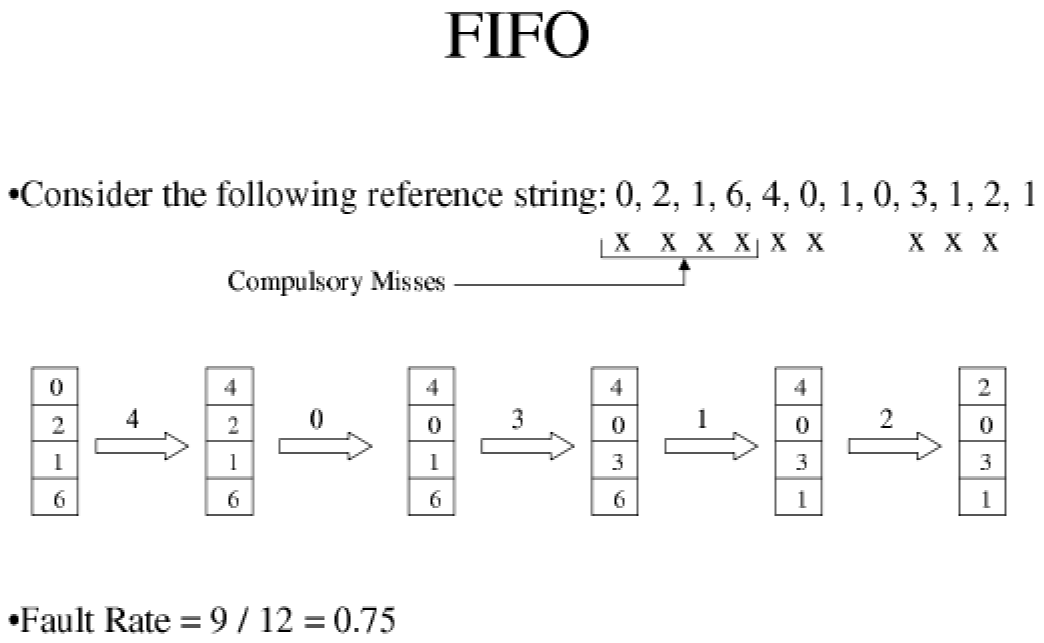
Optimal Page Replacement

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

•Fault Rate = 6 / 12 = 0.50
•With the above reference string, this is the best we can hope to do

FIFO: (First In First Out)

•The oldest page in the physical memory is the one selected for replacement.
•Very simple to implement.
- Keep a list
On a page fault, the page at the head is removed and the new page added to the tail of the list



FIFO

•Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

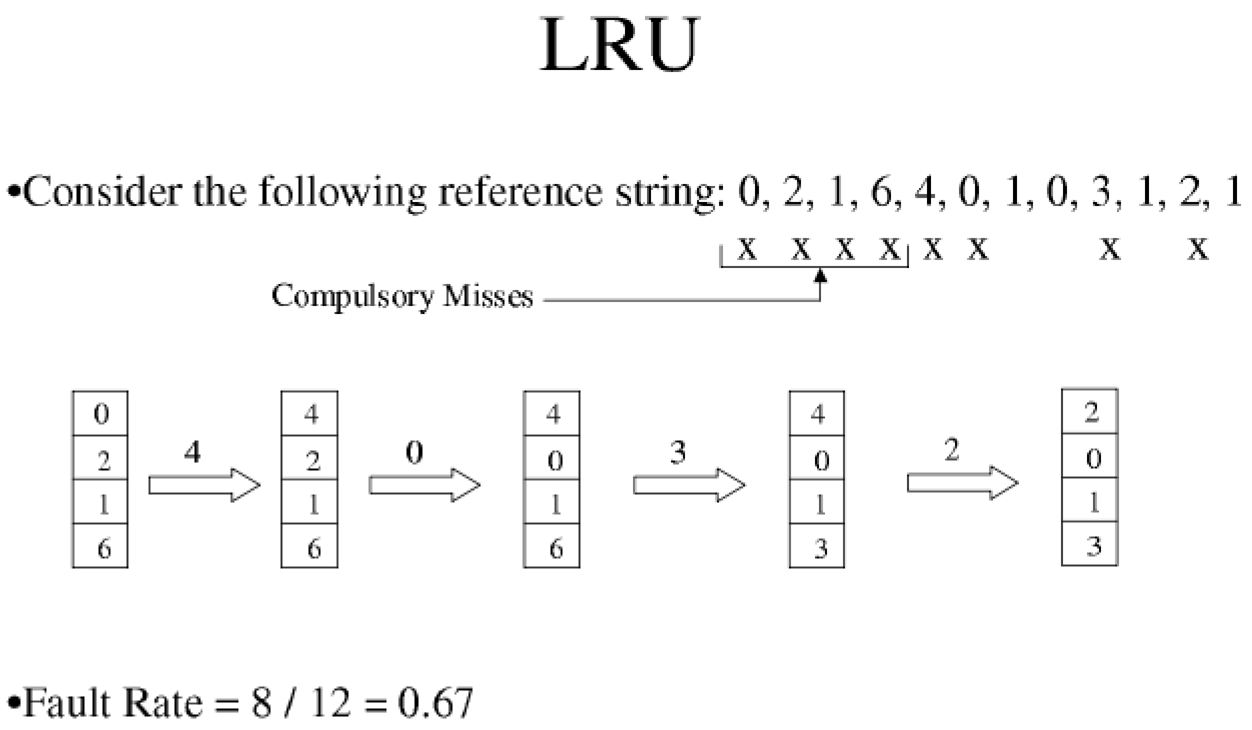
•Fault Rate = 9 / 12 = 0.75

Issues:
•poor replacement policy
•FIFO doesn't consider the page usage.

LRU(Least Recently Used):

In this algorithm, the page that has not been used for longest period of time is selected for replacement.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time-consuming operation, even in hardware (assuming that such hardware could be built).



LRU

- Consider the following reference string: 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

0	4	4	4	2
2	2	0	0	0
1	1	1	1	1
6	6	6	3	3

- Fault Rate = 8 / 12 = 0.67

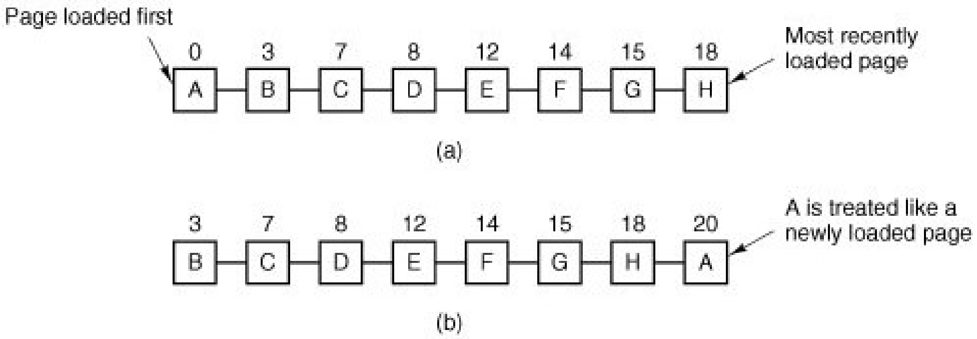
The Not Recently Used Page Replacement Algorithm

Two status bit associated with each page. R is set whenever the page is referenced (read or written). M is set when the page is written to (i.e., modified).

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their R and M bits:
Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.
The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class.

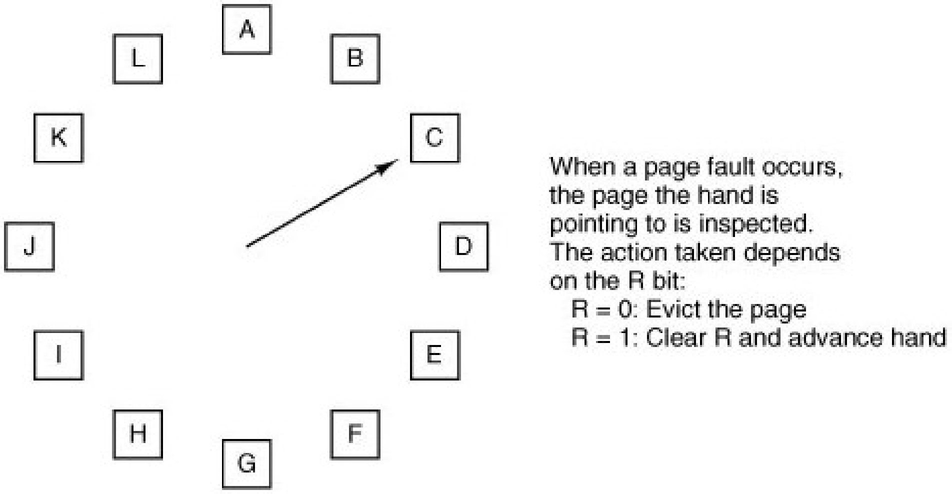
The Second Chance Page Replacement Algorithm:

A simple modification to FIFO that avoids the problem of heavily used page. It inspects the R bit 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.



The Clock Page Replacement Algorithm

keep all the page frames on a circular list in the form of a clock, as shown in Fig. A hand points to the oldest page.


A
B
C
D
E
F
G
H
I
J
K
L
When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

When a page fault occurs, the page being pointed to by the hand is inspected. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0

question:

Calculate the no. of page fault for FIFO, LRU and Optimal for the reference string 7,0,1,2,0,3,0,4,2,3,0,3. Assume there are three Frames available in the memory.

External fragmentation:

- free space divided into many small pieces
- result of allocating and deallocating pieces of the storage space of many different sizes
- one may have plenty of free space, it may not be able to all used, or at least used as efficiently as one would like to
- Unused portion of main memory

Internal fragmentation:

- result of reserving a piece of space without ever intending to use it
- Unused portion of page

Segmentation:

segmentation is another techniques of non-contiguous memory allocation method. Its different from paging as pages are physical in nature and hence are fixed in size, whereas the segments are logical in nature and hence are variable size.

It support the user view of the memory rather than system view as supported by paging. In segmentation we divide the the logical address space into different segments. The general division can be: main program, set of subroutines, procedures, functions and set of data structures(stack, array etc). Each segment has a name and length which is loaded into physical memory as it is. For simplicity the segments are referred by a segment number, rather than a segment name. Virtual address space is divided into two parts in which high order units refer to 's' i.e., segment number and lower order units refer to 'd' i.e.,displacement (limit value).

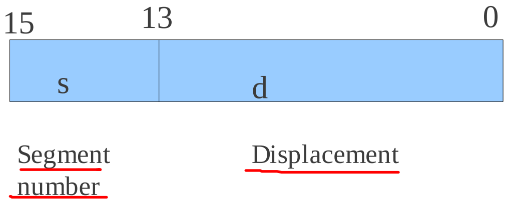
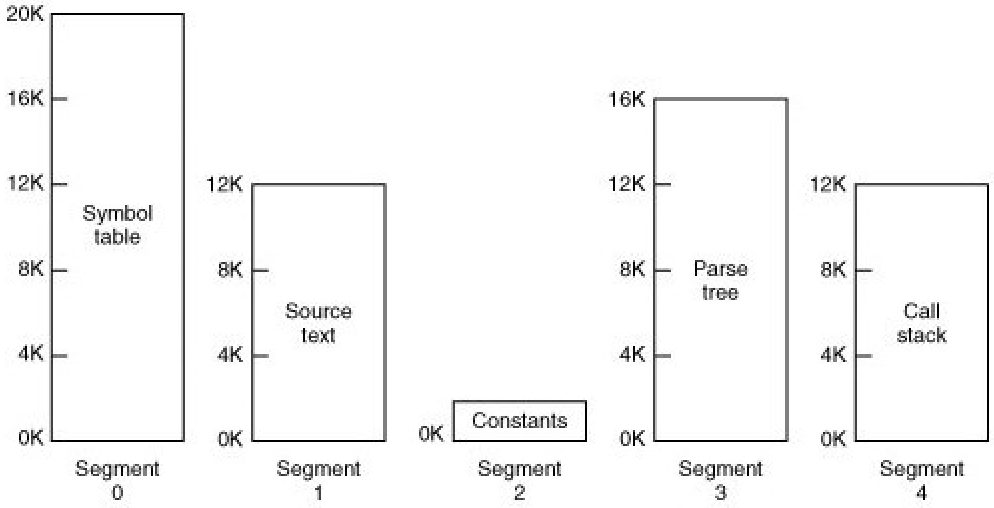
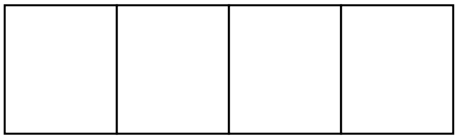
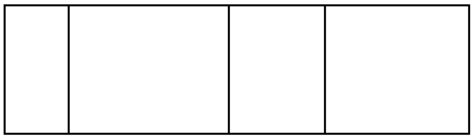


Fig:Virtual address space or logical address space (16 bit)



Segmentation maintains multiple separate virtual address spaces per process. Allows each table to grow or shrink, independently.

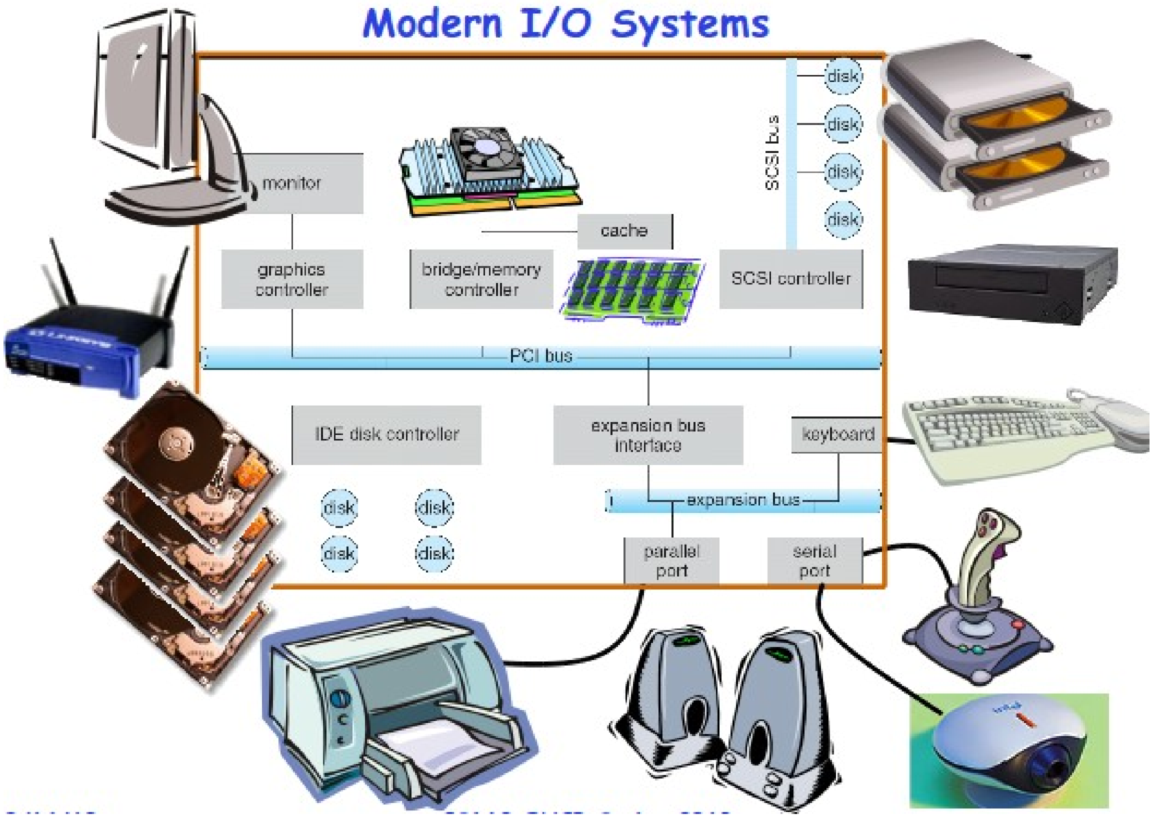
Paging Vs Segmentation:

Sno.	Paging	Segmentation
1	Block replacement easy Fixed-length blocks 	Block replacement hard Variable-length blocks Need to find contiguous, variable-sized, unused part of main memory 
8	No Protection Management	Strong Protection Management
9	Page Table	Segment Table
2	Invisible to application programmer	Visible to application programmer.
3	No external fragmentation, But there is Internal Fragmentation unused portion of page.	No Internal Fragmentation, But there is external Fragmentation unused portion of main memory.
4	Units of code and date are broken into separate pages.	Keeps blocks of code or data as a single units.
5	segmentation is a logical unit visible to the user's program and id of arbitrary size	paging is a physical unit invisible to the user's view and is of fixed size
6	~~Segmentation maintains multiple address spaces per process~~..	~~Paging maintains one address space~~.
7	No sharing of procedures between users is facilitated.	sharing of procedures between users is facilitated.

Introduction, Principals of I/O hardware: I/O devices, device controllers, memory – mapped I/O, DMA (Direct Memory Access), Principles of I/O software: Polled I/O versus Interrupt driven I/O, Character User Interface and Graphical User
Interface, Goals of I/O software, device drivers, device independent I/O software, Disk, disk hardware arm scheduling algorithms, RAID (Redundant Array of Inexpensive Disks)

Chapter:6 Input/Output:

Introduction, Principals of I/O hardware: I/O devices, device controllers, memory – mapped I/O, DMA (Direct Memory Access), Principles of I/O software: Polled I/O versus Interrupt driven I/O, Character User Interface and Graphical User Interface, Goals of I/O software, device drivers, device independent I/O software, Disk, disk hardware arm scheduling algorithms, RAID (Redundant Array of Inexpensive Disks)



What about I/O?

- Without I/O, computers are useless (disembodied brains?)
- But… thousands of devices, each slightly different
- How can we standardize the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
- How can we make them reliable???
- Devices unpredictable and/or slow
- How can we manage them if we don’t know what they will do or how they will perform?

Some operational parameters:

Byte/Block
Some devices provide single byte at a time (*e.g.* keyboard)
Others provide whole blocks (*e.g.* disks, networks, etc)
Sequential/Random
Some devices must be accessed sequentially (*e.g.* tape)
Others can be accessed randomly (*e.g.* disk, cd, etc.)

Polling/Interrupts
Some devices require continual monitoring
Others generate interrupts when they need service

I/O devices can be roughly divided into two categories: **block devices and character devices**. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 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. Disks are the most common block devices.

The other type of I/O device is the **character device.** A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

Block Devices: e.g. disk drives, tape drives, DVD-ROM
Access blocks of data
Commands include open() , read() , write() , seek()
Raw I/O or file-system access
Memory-mapped file access possible

Character Devices: e.g. keyboards, mice, serial ports, some USB devices
Single characters at a time
Commands include get() , put()
Libraries layered on top allow line editing

Network Devices: e.g. Ethernet, Wireless, Bluetooth
Different enough from block/character to have own interface
Unix and Windows include socket interface
Separates network protocol from network operation
Includes select() functionality
Usage: pipes, FIFOs, streams, queues, mailboxes

Device Controllers:

A device controller is a hardware unit which is attached with the input/output bus of the computer and provides a hardware interface between the computer and the input/output devices. On one side it knows how to communicate with input/output devices and on the other side it knows how to communicate with the computer system though input/output bus. A device controller usually can control several input/output devices.

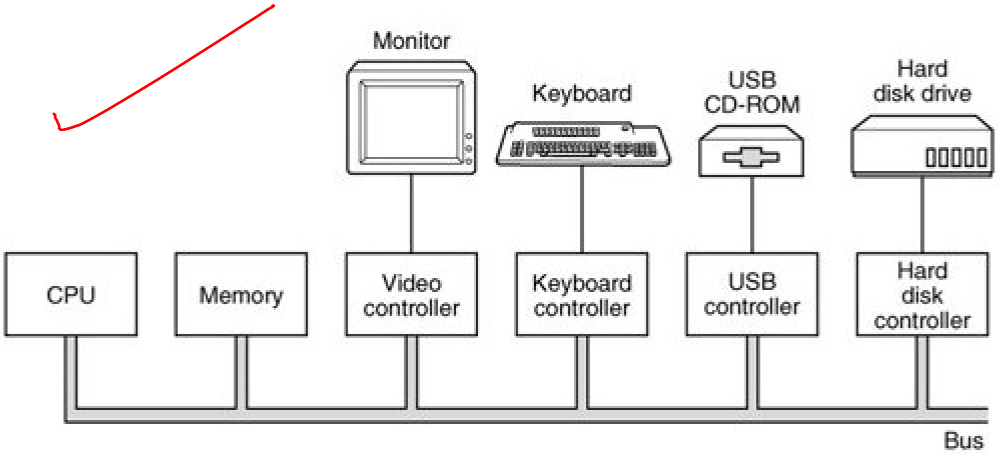


Fig: A model for connecting the CPU, memory, controllers, and I/O devices

Typically the controller is on a card (eg. LAN card, USB card etc). Device Controller play an important role in order to operate that device. It's just like a bridge between device and operating system.

Most controllers have DMA(Direct Memory Access) capability, that means they can directly read/write memory in the system. A controller without DMA capability provide or accept the data, one byte or word at a time; and the processor takes care of storing it, in memory or reading it from the memory. DMA controllers can transfer data much faster than non-DMA controllers. Presently all controllers have DMA capability.

DMA is a memory-to-device communication method that by passes the CPU.

Memory-mapped Input/Output:

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.
In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

There are two alternatives that the CPU communicates with the control registers and the device data buffers.

Port-mapped I/O :

each control register is assigned an I/O port number, an 8- or 16-bit integer. Using a special I/O instruction such as

IN REG,PORT

the CPU can read in control register PORT and store the result in CPU register REG. Similarly, using

OUT PORT,REG

the CPU can write the contents of REG to a control register. Most early computers, including nearly all mainframes, such as the IBM 360 and all of its successors, worked this way.

In this scheme, the address spaces for memory and I/O are different, as shown in Fig. (a).Port-mapped I/O uses a special class of CPU instructions specifically for performing I/O.

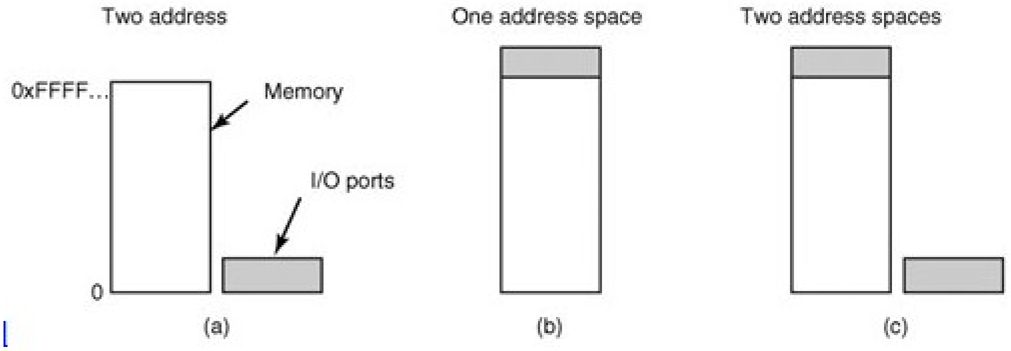


Fig:(a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

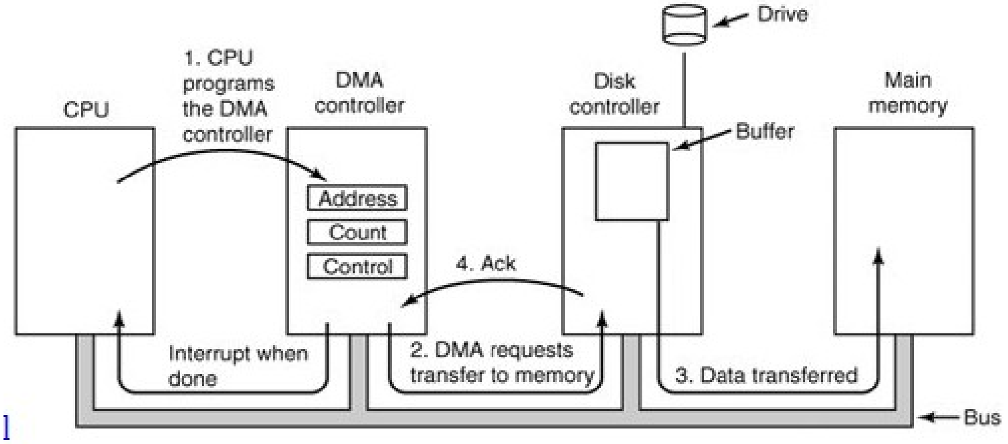
On other computers, I/O registers are part of the regular memory address space, as shown in Fig.(b). This scheme is called memory-mapped I/O, and was introduced with the PDP-11 minicomputer.Memory-mapped I/O (not to be confused with memory-mapped file I/O) uses the same address bus to address both memory and I/O devices, and the CPU instructions used to access the memory are also used for accessing devices. In order to accommodate the I/O devices, areas of the CPU's addressable space must be reserved for I/O.

DMA: (Direct Memory Access)

Short for direct memory access, a technique for transferring data from main memory to a device without passing it through the CPU. Computers that have DMA channels can transfer data to and from devices much more quickly than computers without a DMA channel can. This is useful for making quick backups and for real-time applications.

Direct Memory Access (DMA) is a method of allowing data to be moved from one location to another in a computer without intervention from the central processor (CPU).

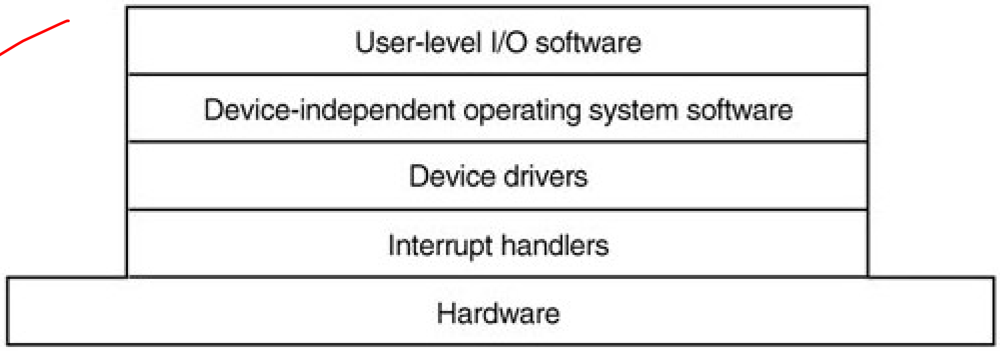
Make FlowChart also



First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Fig.). 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 address lines of the bus 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 this point the controller causes an interrupt. When the operating system starts up, it does not have to copy the block to memory; it is already there.

Layers of the I/O software system:



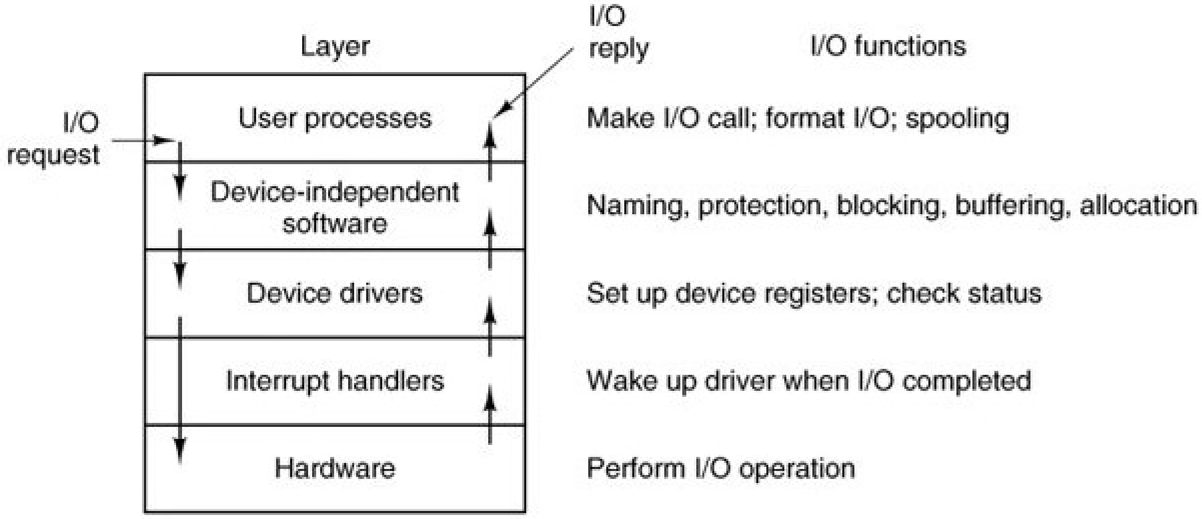

Layer
I/O reply
I/O functions
I/O request
User processes
Device-independent software
Device drivers
Interrupt handlers
Hardware
Make I/O call; format I/O; spooling
Naming, protection, blocking, buffering, allocation
Set up device registers; check status
Wake up driver when I/O completed
Perform I/O operation

Fig. Layers of the I/O system and the main functions of each layer.

The arrows in fig above show the flow of control. When a user program tries to read a block from a file, for example, the operating system is invoked to carry out the call. The device-independent software looks for it in the buffer cache, for example. If the needed block is not there, it calls the device driver to issue the request to the hardware to go get it from the disk. The process is then blocked until the disk operation has been completed.
When the disk is finished, the hardware generates an interrupt. The interrupt handler is run to discover what has happened, that is, which device wants attention right now. It then extracts the status from the device and wakes up the sleeping process to finish off the I/O request and let the user process continue.

Device Driver:

In computing, a device driver or software driver is a computer program allowing higher-level computer programs to interact with a hardware device.
A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

Each device controller has registers used to give it commands or to read out its status or both. The number of registers and the nature of the commands vary radically from device to device. For example, a mouse driver has to accept information from the mouse telling how far it has moved and which buttons are currently depressed. In contrast, a disk driver has to know about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, and all the other mechanics of making the disk work properly. Obviously, these drivers will be very different.

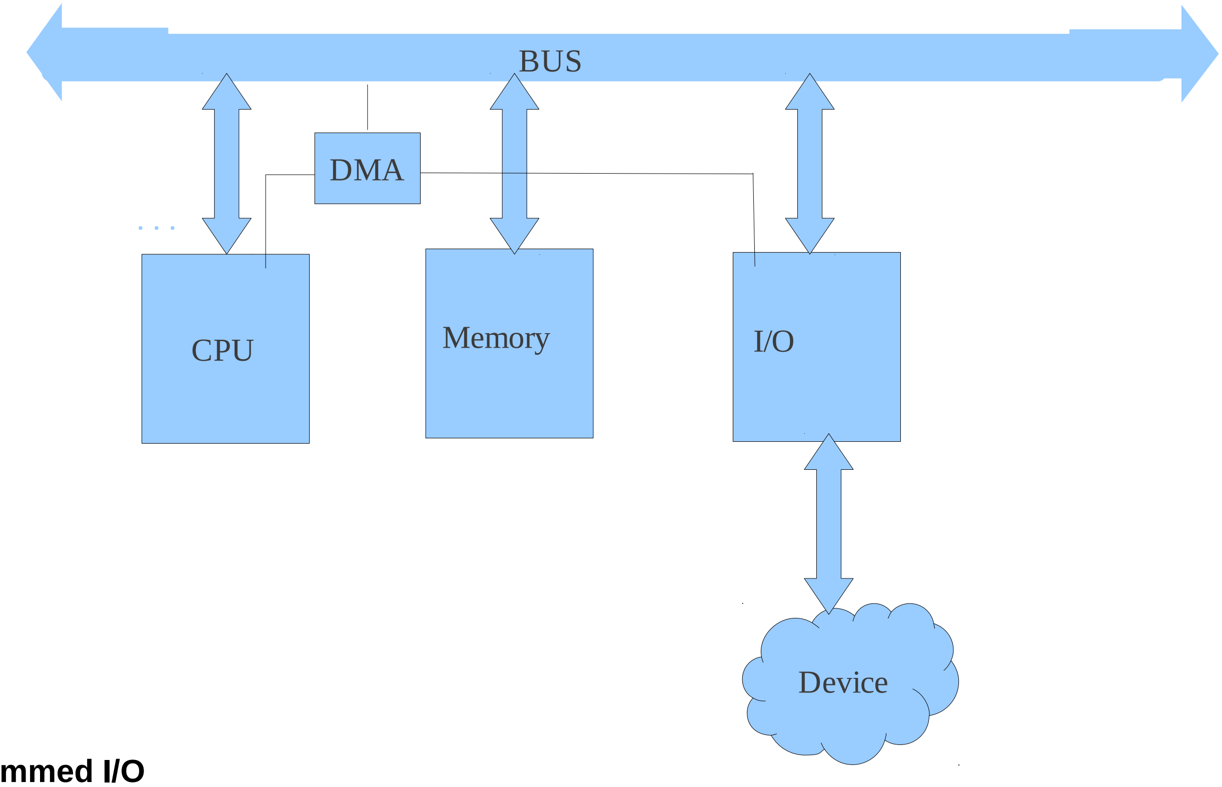
Thus, each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the device driver, is generally written by the device's manufacturer and delivered along with the device on a CD-ROM. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.

Each device driver normally handles one device type, or one class of closely related devices. For example, it would probably be a good idea to have a single mouse driver, even if the system supports several different brands of mice. As another example, a disk driver can usually handle multiple disks of different sizes and different speeds, and perhaps a CD-ROM as well. On the other hand, a mouse and a disk are so different that different drivers are necessary.

Ways to do INPUT/OUTPUT:

There are three fundamentally different ways to do I/O.

1. Programmed I/O
2. Interrupt-driven
3. Direct Memory access



Programmed I/O

The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy waits for the operation to be completed before proceeding.

When the processor is executing a program and encounters an instruction relating to input/output, it executes that instruction by issuing a command to the appropriate input/output module. With the programmed input/output, the input/output module will perform the required action and then set the appropriate bits in the input/output status register. The input/output module takes no further action to alert the processor. In particular it doesn't interrupt the processor. Thus, it is the responsibility of the processor to check the status of the input/output module periodically, until it finds that the operation is complete.

It is simplest to illustrate programmed I/O by means of an example . Consider a process that wants to print the Eight character string ABCDEFGH.

1. It first assemble the string in a buffer in user space as shown in fig.
2. The user process then acquires the printer for writing by making system call to open it.

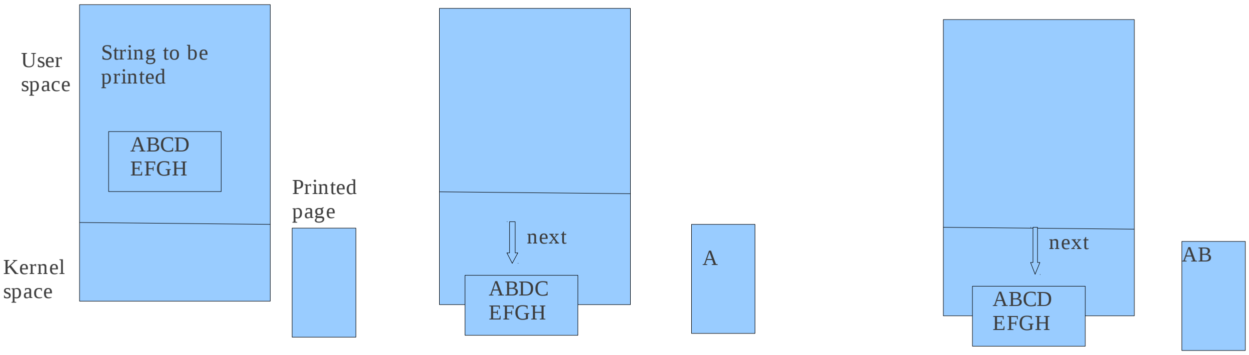


Fig. Steps in Printing a string

3. If printer is in use by other the call will fail and enter an error code or will block until printer is available, depending on OS and the parameters of the call.
4. Once it has printer the user process makes a system call to print it.
5. OS then usually copies the buffer with the string to an array, say P in the kernel space where it is more easily accessed since the kernel may have to change the memory map to get to user space.
6. As the printer is available the OS copies the first character to the printer data register, in this example using memory mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing.
7. As soon as it has copied the first character to the printer the OS checks to see if the printer is ready to accept another one.
8. Generally printer has a second register which gives its status

The action followed by the OS are summarized in fig below. First data are copied to the kernel, then the OS enters a tight loop outputting the characters one at a time. The essentials aspects of programmed I/O is after outputting a character, the CPU continuously polls the device to see if it is ready to accept one. This behavior is often called polling or Busy waiting.

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

Programmed I/O is simple but has disadvantages of tying up the CPU full time until all the I/O is done. In an embedded system where the CPU has nothing else to do, busy waiting is reasonable. However in more complex system where the cpu has to do other things, busy waiting is inefficient. A better I/O method is needed.

Interrupt-driven I/O:

The problem with the programmed I/O is that the processor has to wait a long time for the input/output module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the Input/ Output module. As a result the level of performance of entire system is degraded.

An alternative approach for this is interrupt driven Input / Output. The processor issue an Input/Output command to a module and then go on to do some other useful work. The input/ Output module will then interrupt the processor to request service, when it is ready to exchange data with the processor. The processor then executes the data transfer as before and then resumes its former processing. Interrupt-driven input/output still consumes a lot of time because every data has to pass with processor.

DMA:

The previous ways of I/O suffer from two inherent drawbacks.

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer;a number of instructions must be executed for each I/O transfer.

When large volumes of data are to be moved, a more efficient technique is required:Direct memory access. The DMA function can be performed by a separate module on the system bus, or it can be incorporated into an I/O module. In either case , the technique works as follow.

When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending the following information.

- Whether a read or write is requested.
- The address of the I/O devices.
- Starting location in memory to read from or write to.
- The number of words to be read or written.

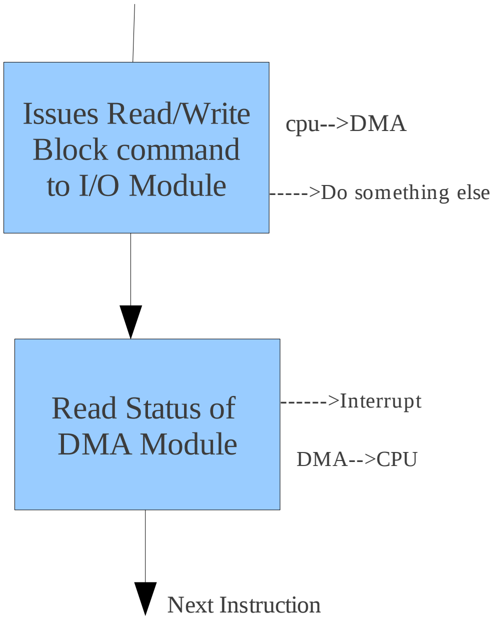


Fig:DMA

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at at time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus the processor is involved only at the beginning and at the end of the transfer.

In programmed I/O cpu takes care of whether the device is ready or not. Data may be lost. Whereas in Interrupt-driven I/O, device itself inform the cpu by generating an interrupt signal. if the data rate of the i/o is too fast. Data may be lost. In this case cpu most be cut off, since cpu is too slow for the particular device. the initial state is too fast.

it is meaningful to allow the device to put the data directly to the memory. This is called DMA.

dma controller will take over the task of cpu. Cpu is general purpose but the dma controller is specific purpose.

A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for

the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

Disks:

All real disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on some hard disks. The simplest designs have the same number of sectors on each track.

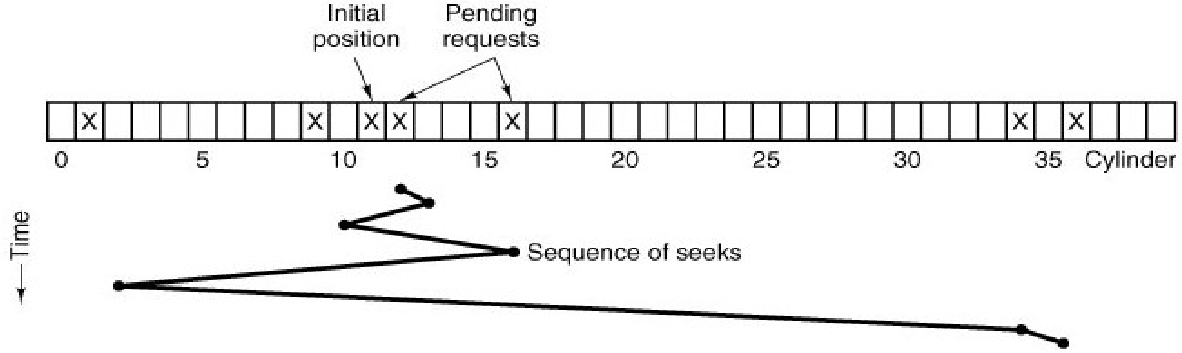
Disk Arm Scheduling Algorithms:

1. First come First server FCFS
2. Shortest Seek First Shortest Seek First (SSF) disk scheduling algorithm.
3. The elevator algorithm for scheduling disk requests.

Consider a disk with 40 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order.

Shortest Seek First (SSF) disk scheduling algorithm.

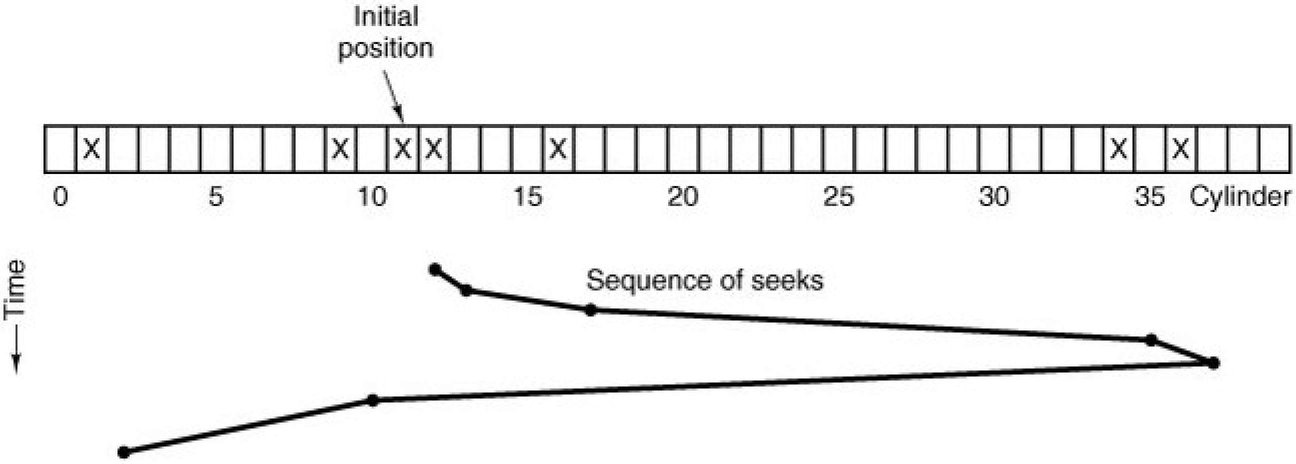
Pick the request closet to the head.



Total no of Cylinder movement: 61

The elevator algorithm for scheduling disk requests:
Keep moving in same direction until there are no more outstanding requests in that direction, then they switch direction. The elevator algorithm requires software to maintain 1 bit, the current direction bit. UP or DOWN. If it is UP the arm is moved to the next highest pending request and if it is DOWN if it moved to the next lowest pending request if any.

the elevator algorithm using the same seven requests as shown above, assuming the direction bit was initially UP. The order in which the cylinders are serviced is 12, 16, 34, 36, 9, and 1, which yields arm motions of 1, 4, 18, 2, 27, and 8, for a total of 60 cylinders.



Questions:
The disk requests come in to the disk driver for cylinders 10, 20, 22,2,40,6 and 30, in the order. A seek takes 6 msec/cylinder moved. How much seek time is needed for:

i) FCFS

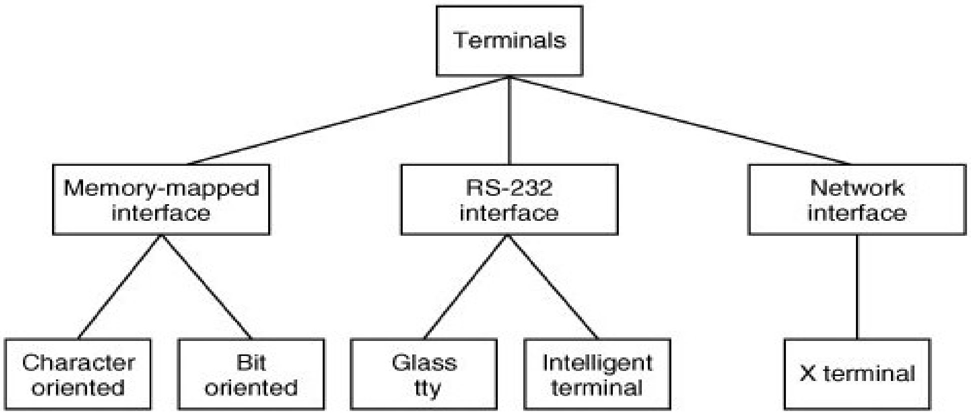
ii) Shortest Seek First

iii) Elevator algorithm

Terminals:

For decades, users have communicated with computers using devices consisting of a keyboard for user input and a display for computer output. For many years, these were combined into free-standing devices called terminals, which were connected to the computer by a wire. Large mainframes used in the financial and travel industries sometimes still use these terminals, typically connected to the mainframe via a modem, especially when they are far from the mainframe.

Terminal types.



Clock:

clock also called timers are essential to the operation of any multiprogrammed system for variety of reasons.

- maintain time of day
- prevent one process from monopolizing the CPU among other things.
- clock software can take the form of device driver, but it is neither a block device like disk neither a character like mouse.

Two clock:

the simpler clock is tied to 110 or 220 volt power line and causes and interrupt on every voltage cycle at 50 or 60hz.

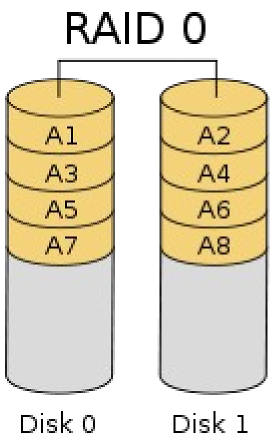
the other kind of clock built of 3 components a crystal oscillator, a counter and a holding register.

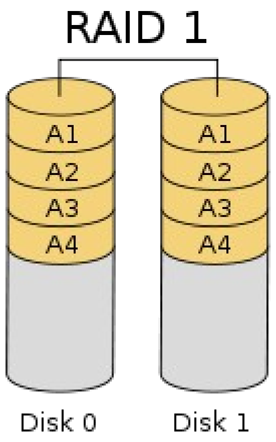
RAID

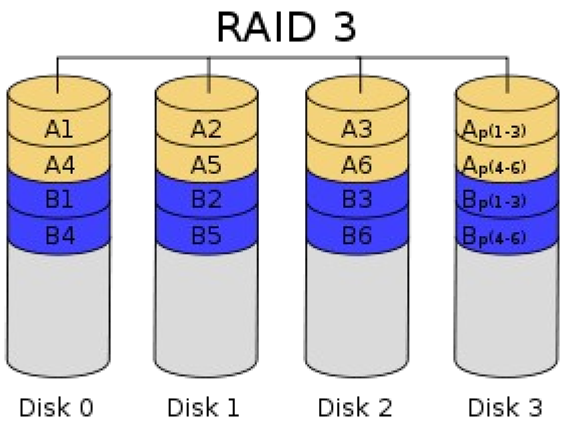
RAID (**redundant array of independent disks**, originally **redundant array of inexpensive disks** is a storage technology that combines multiple disk drive components into a logical unit. Data is distributed across the drives in one of several ways called "RAID levels", depending on what level of redundancy and performance (via parallel communication) is required.

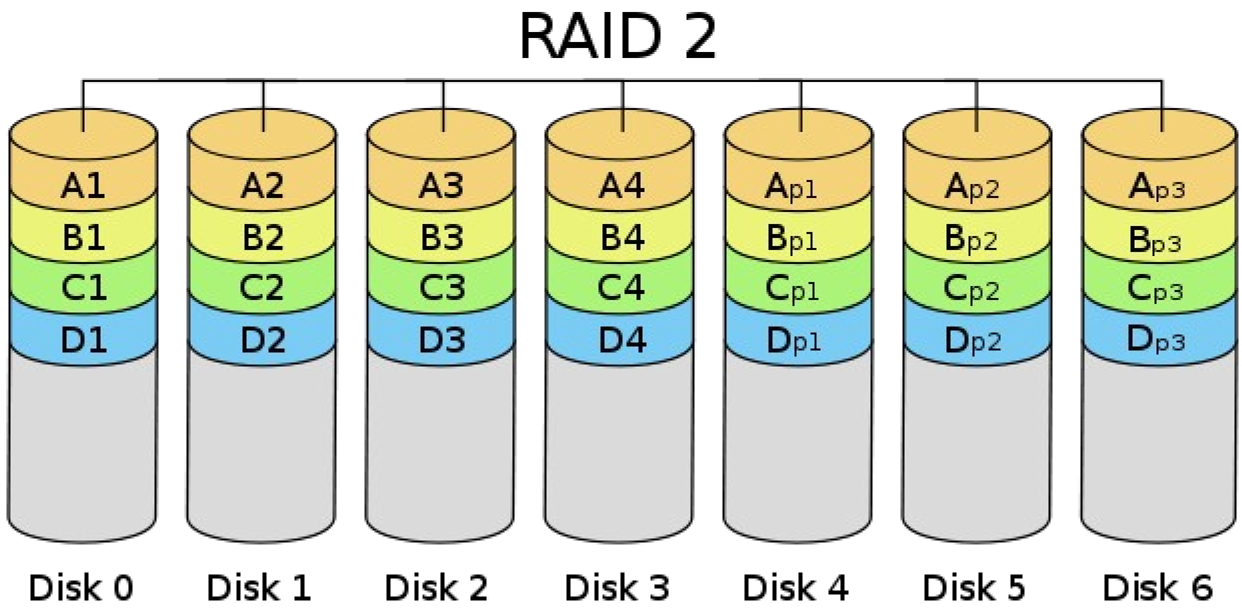
RAID Levels:

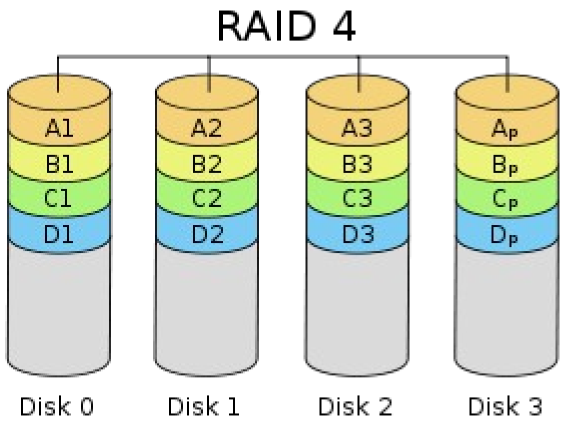
RAID 0 (block-level striping without parity or mirroring) has no (or zero) redundancy. It provides improved performance and additional storage but no fault tolerance. Hence simple stripe sets are normally referred to as RAID 0. Any drive failure destroys the array, and the likelihood of failure increases with more drives in the array (at a minimum, catastrophic data loss is almost twice as likely compared to single drives without RAID). A single drive failure destroys the entire array because when data is written to a RAID 0 volume, the data is broken into fragments called blocks. The number of blocks is dictated by the stripe size, which is a configuration parameter of the array. The blocks are written to their respective drives simultaneously on the same sector. This allows smaller sections of the entire chunk of data to be read off each drive in parallel, increasing bandwidth. RAID 0 does not implement error checking, so any error is uncorrectable. More drives in the array means higher bandwidth, but greater risk of data loss.

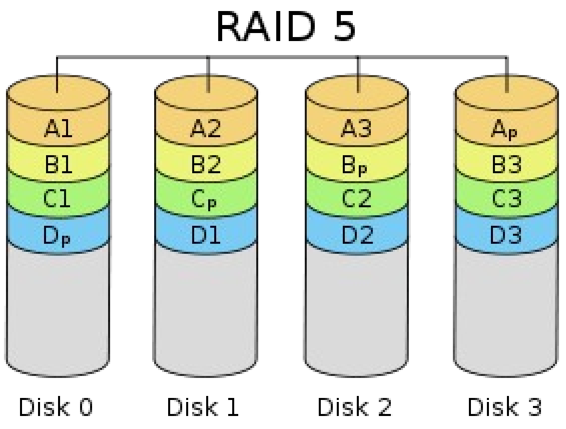

RAID 0
A1
A3
A5
A7
A2
A4
A6
A8
Disk 0
Disk 1


RAID 1
A1
A2
A3
A4
A1
A2
A3
A4
Disk 0
Disk 1


RAID 3
A1
A4
B1
B4
A2
A5
B2
B5
A3
A6
B3
B6
Ap(1-3)
Ap(4-6)
Bp(1-3)
Bp(4-6)
Disk 0
Disk 1
Disk 2
Disk 3


RAID 2
A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3
A4
B4
C4
D4
Ap1
Bp1
Cp1
Dp1
Ap2
Bp2
Cp2
Dp2
Ap3
Bp3
Cp3
Dp3
Disk 0
Disk 1
Disk 2
Disk 3
Disk 4
Disk 5
Disk 6


RAID 4
A1
B1
C1
D1
A2
B2
C2
D2
A3
B3
C3
D3
Ap
Bp
Cp
Dp
Disk 0
Disk 1
Disk 2
Disk 3


RAID 5
A1
B1
C1
Dp
A2
B2
Cp
D1
A3
Bp
C2
D2
Ap
B3
C3
D3
Disk 0
Disk 1
Disk 2
Disk 3

In ***RAID 1*** (mirroring without parity or striping), data is written identically to multiple drives, thereby producing a "mirrored set"; at least 2 drives are required to constitute such an array. While more constituent drives may be employed, many implementations deal with a maximum of only 2; of course, it might be possible to use such a limited level 1 RAID itself as a constituent of a level 1 RAID, effectively masking the limitation.The array continues to operate as long as at least one drive is functioning. With appropriate operating system support, there can be increased read performance, and only a minimal write performance reduction; implementing RAID 1 with a separate controller for each drive in order to perform simultaneous reads (and writes) is sometimes called *multiplexing* (or *duplexing* when there are only 2 drives).

In ***RAID 2*** (bit-level striping with dedicated Hamming-code parity), all disk spindle rotation is synchronized, and data is striped such that each sequential bit is on a different drive. Hamming-code parity is calculated across corresponding bits and stored on at least one parity drive.

In ***RAID 3*** (byte-level striping with dedicated parity), all disk spindle rotation is synchronized, and data is striped so each sequential byte is on a different drive. Parity is calculated across corresponding bytes and stored on a dedicated parity drive.

RAID 4 (block-level striping with dedicated parity) is identical to RAID 5 (see below), but confines all parity data to a single drive. In this setup, files may be distributed between multiple drives. Each drive operates independently, allowing I/O requests to be performed in parallel. However, the use of a dedicated parity drive could create a performance bottleneck; because the parity data must be written to a single, dedicated parity drive for each block of non-parity data, the overall write performance may depend a great deal on the performance of this parity drive.

RAID 5 (block-level striping with distributed parity) distributes parity along with the data and requires all drives but one to be present to operate; the array is not destroyed by a single drive failure. Upon drive failure, any subsequent reads can be calculated from the distributed parity such that the drive failure is masked from the end user. However, a single drive failure results in reduced performance of the entire array until the failed drive has been replaced and the associated data rebuilt. Additionally, there is the potentially disastrous RAID 5 write hole. RAID 5 requires at least 3 disks.

RAID 6 (block-level striping with double distributed parity) provides fault tolerance of two drive failures; the array continues to operate with up to two failed drives. This makes larger RAID groups more practical, especially for high-availability systems. This becomes increasingly important as large-capacity drives lengthen the time needed to recover from the failure of a single drive. Single-parity RAID levels are as vulnerable to data loss as a RAID 0 array until the failed drive is replaced and its data rebuilt; the larger the drive, the longer the rebuild takes. Double parity gives additional time to rebuild the array without the data being at risk if a single additional drive fails before the rebuild is complete.

Chapter:7 File-systems

File naming , file structure, file types, file access, file attributes, file operations, File descriptor, Access Control Matrix, sharing, ACL (Access Control List), Directories and directory hierarchy, File system implementation, contiguous allocation, linked list allocation, I-nodes, security and Multi – media files.

What is File-System?

From the user point of view one of the most important part of the operating system is file-system. The file-system provides the resource abstraction typically associated with secondary storage. The file system permits users to create data collections, called files, with desirable properties, such as

- **Long-term existence:** Files are stored on disk or other secondary storage and do not disappear when a user logs off.
- **Sharable between processes:** Files have names and can have associated access permissions that permit controlled sharing.
- **Structure:** Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of 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.

Operation Performed on Files:

1. Creating a File
2. Writing a file
3. Reading a file
4. Repositioning a file
5. Deleting a file
6. Truncating a file

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. We will call these extra items the file's **attributes** although some people called them **metadata**. The list of attributes varies considerably from system to system.

Attribute	**Meaning**
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

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. A system call for this purpose is always provided.
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 some 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.

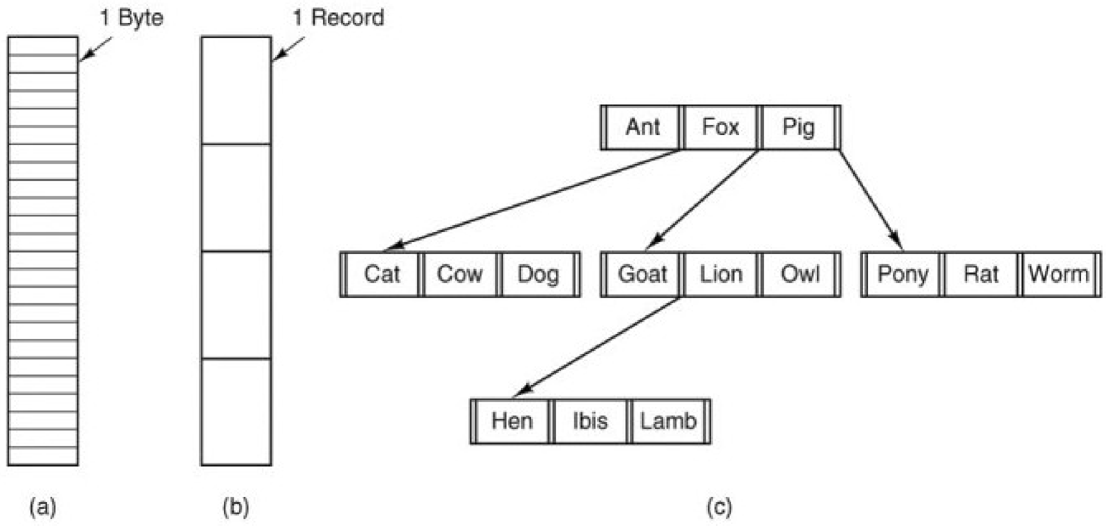
9. **Get attributes**. Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

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.

12. **Lock.** Locking a file or a part of a file prevents multiple simultaneous access by different process. For an airline reservation system, for instance, locking the database while making a reservation prevents reservation of a seat for two different travelers.

File Structure:



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

a. Byte Sequence:

The file in Fig. (a) is just 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. Both UNIX and Windows 98 use this approach.

b. Record Sequence:

In this model, 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. As a historical note, when the 80-column punched card was king many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images

c. Record Sequence:

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.

Files Organization and Access Mechanism:

When a file is used then the stored information in the file must be accessed and read into the memory of a computer system. Various mechanism are provided to access a file from the operating system.

i. Sequential access
ii. Direct Access
iii. Index Access

Sequential Access:
It is the simplest access mechanism, in which informations stored in a file are accessed in an order such that one record is processed after the other. For example editors and compilers usually access files in this manner.

Direct Access:
It is an alternative method for accessing a file, which is based on the disk model of a file, since disk allows random access to any block or record of a file. For this method, a file is viewed as a numbered sequence of blocks or records which are read/written in an arbitrary manner, i.e. there is no restriction on the order of reading or writing. It is well suited for Database management System.

Index Access:
In this method an index is created which contains a key field and pointers to the various block. To find an entry in the file for a key value , we first search the index and then use the pointer to directly access a file and find the desired entry.

With large files , the index file itself may become too large to be keep in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items.

File Allocation Method:

1. **Contiguous Allocation**
2. **Linked List Allocation**
3. **Linked List Allocation Using a Table in Memory**
4. **I-Nodes**

Contiguous allocation:

It requires each file to occupy a set of contiguous addresses on a disk. It sore each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. Both sequential and direct access is supported by the contiguous allocation method.
Contiguous disk space allocation has two significant advantages.

1. First, i**t 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. Second, **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.
Unfortunately, contiguous allocation also has a major drawback: in time, the disk becomes fragmented, consisting of files and holes. It needs compaction to avoid this.

Example of contiguous allocation: CD and DVD ROMs

Linked List Allocation:

keep each file as a linked list of disk blocks as shown in the fig. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

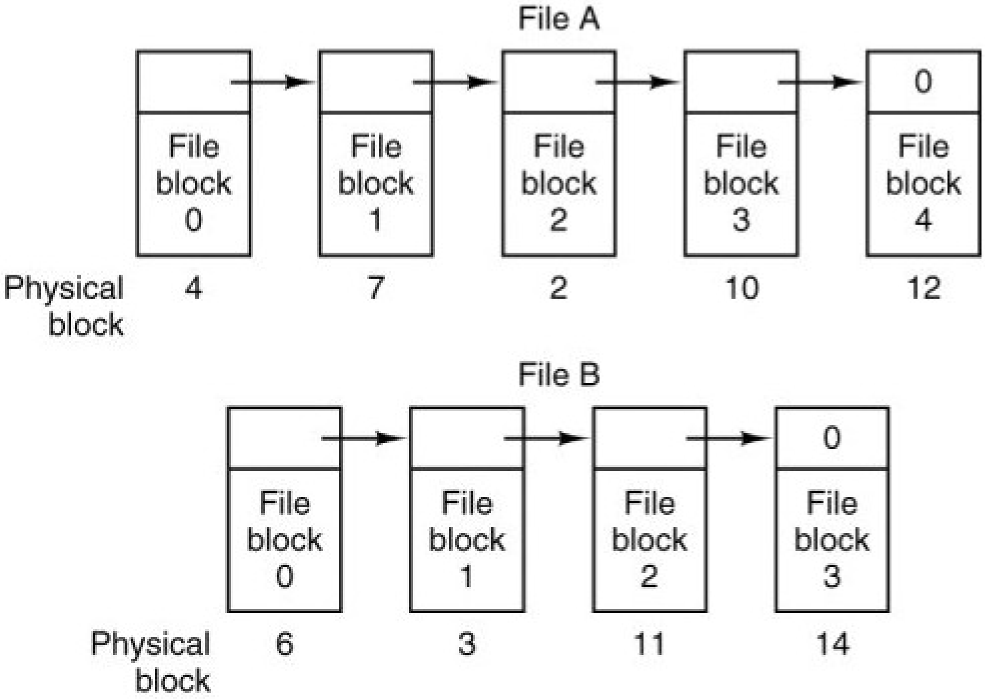


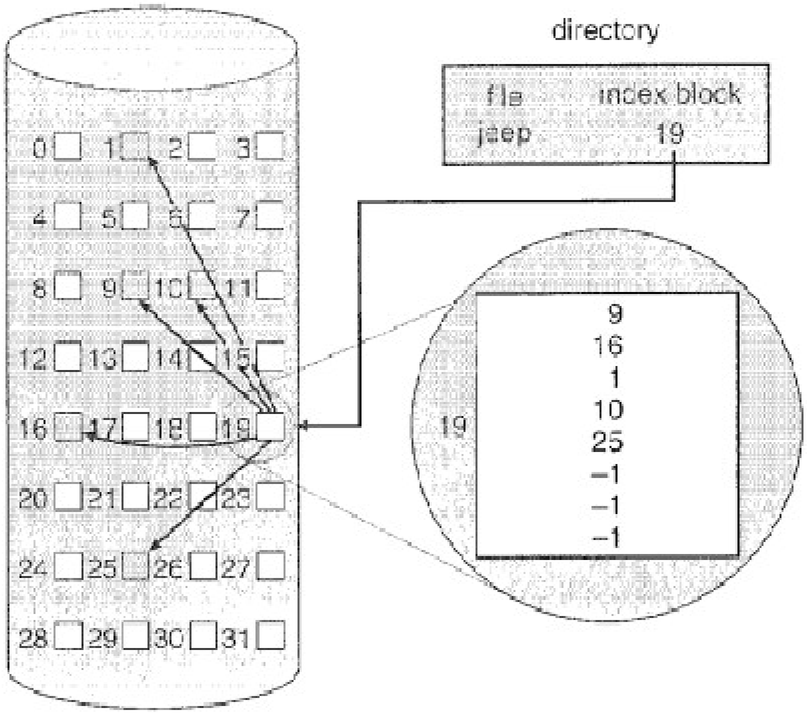
Fig:Storing a file as a linked list of disk blocks.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation. The major problem with linked allocation is that it can be used only for sequential access files. To find the i^{th} block of a file, we must start at the beginning of that file, and follow the pointers until we get the i^{th} block. It is inefficient to support direct access capability for linked allocation of files.

Another problem of linked list allocation is reliability. Since the files are linked together with the pointer scattered all over the disk. Consider what will happen if a pointer is lost or damaged.

Indexed allocation (I-Nodes):

It solves the external fragmentation and size declaration problems of contiguous allocation. In this allocation all pointers are brought together into one location called **Index block.**
Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block.



Inex alloction of Disk sapce

To read the ith block, we use the pointer in the ith index block entry to find and read the desired block. This scheme is similar to the paging scheme.

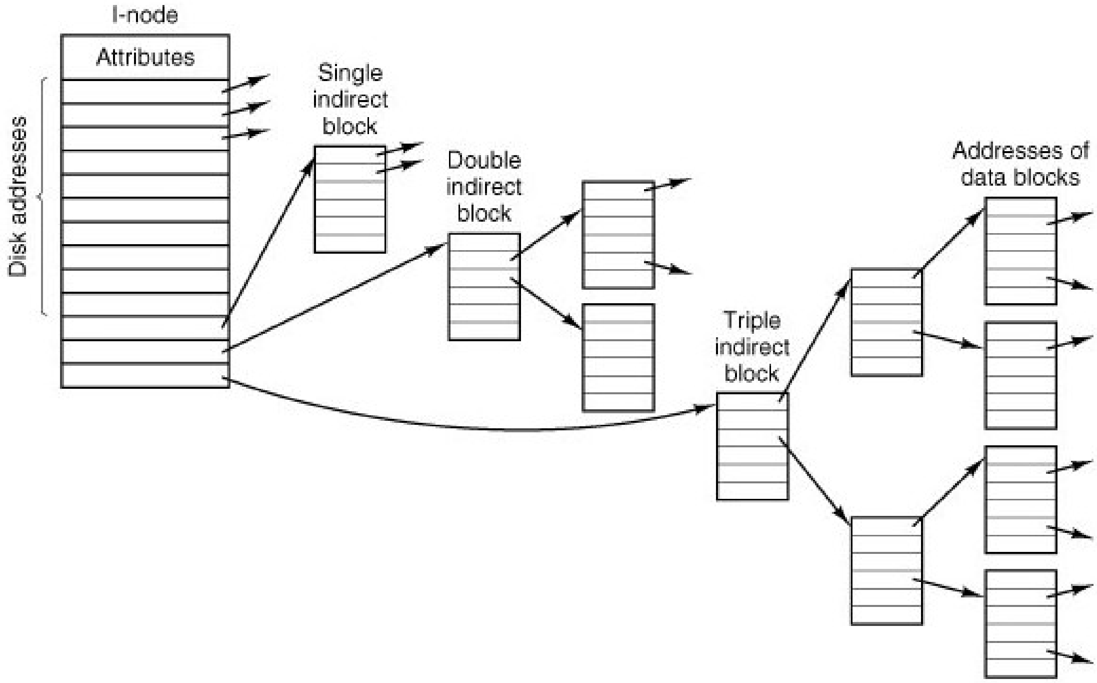


Fig: An i-node with three levels of indirect blocks.

File System Layout:

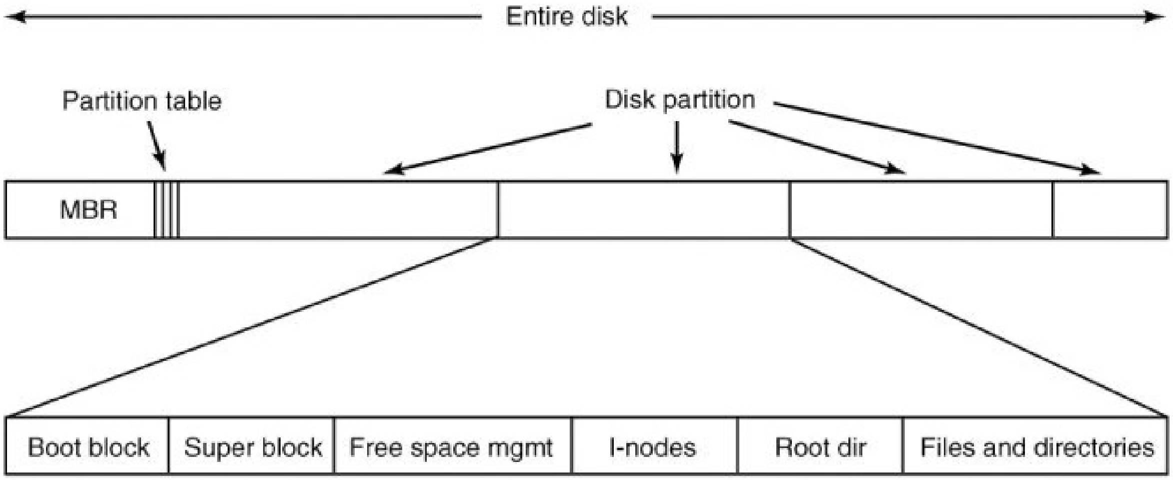


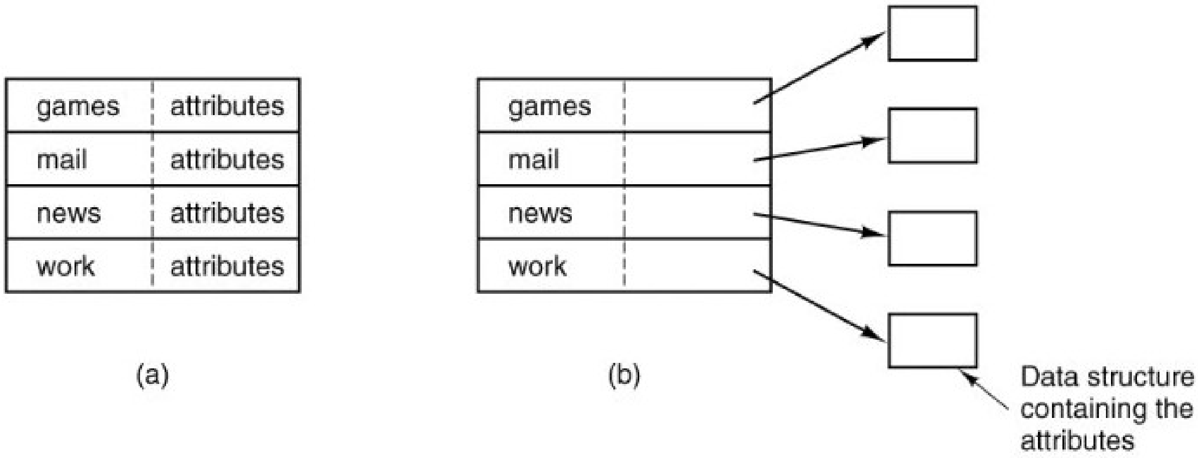
Fig:A possible file system layout.

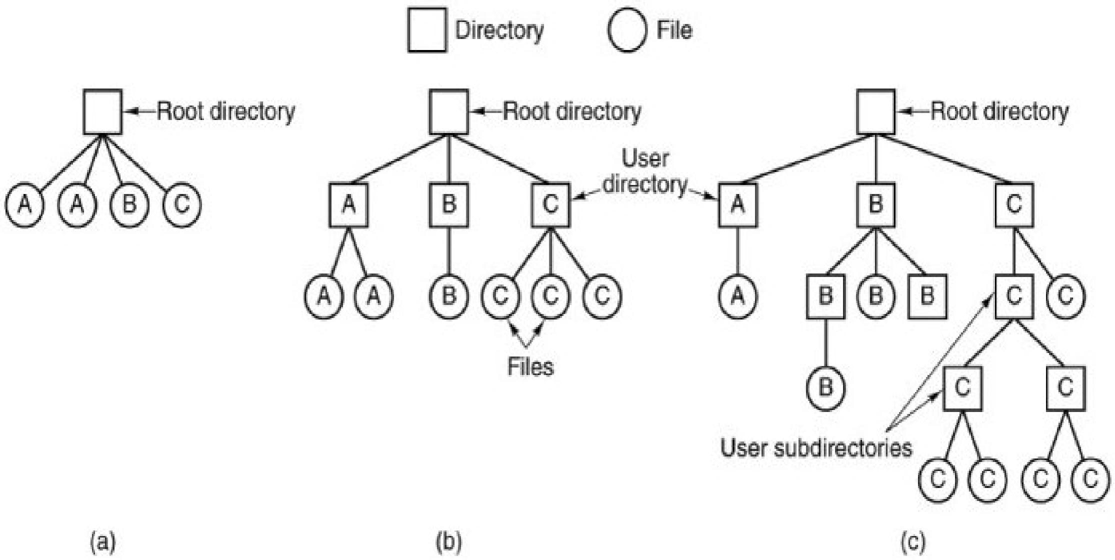
Directories:

To keep track of files, file systems normally have directories or folders, which, in many systems, are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

Simple Directories

A directory typically contains a number of entries, one per file. One possibility is shown in Fig. (a), in which each entry contains the file name, the file attributes, and the disk addresses where the data are stored.

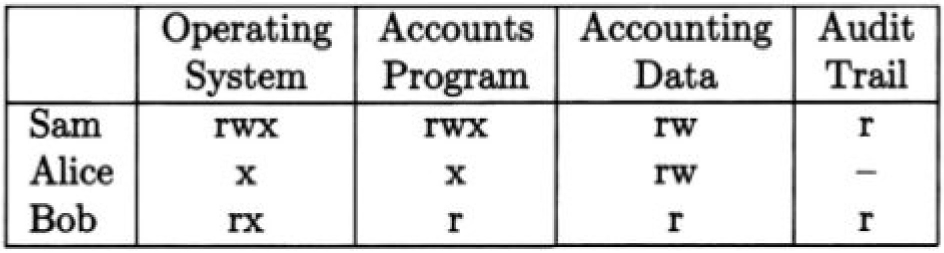




Three file system designs. (a) Single directory shared by all users. (b) One directory per user. (c) Arbitrary tree per user. The letters indicate the directory or file's owner.

Access Control Matrix

The access controls provided with an operating system typically authenticate principals using some mechanism such as passwords or Kerberos, then mediate their access to files, communications ports, and other system resources. Their effect can often be modelled by a matrix of access permissions, with columns for files and rows for users. We'll write r for permission to read, w for permission to write, x for permission to execute a program, and (–) for no access at all, as shown in Figure below.



	Operating System	Accounts Program	Accounting Data	Audit Trail
Sam	rwx	rwx	rw	r
Alice	x	x	rw	–
Bob	rx	r	r	r

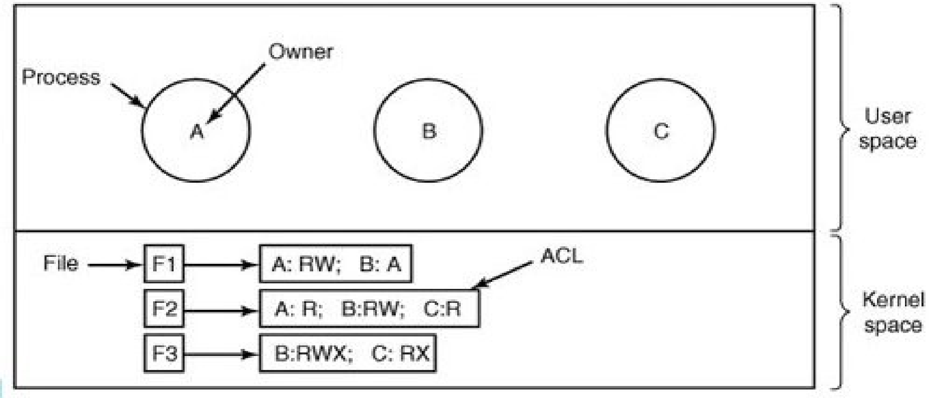
Fig: Access Control Matrix

In this simplified example, Sam is the system administrator, and has universal access (except to the audit trail, which even he should only be able to read). Alice, the manager, needs to execute the operating system and application, but only through the approved interfaces—she mustn't have the ability to tamper with them. She also needs to read and write the data. Bob, the auditor, can read everything.

Access control matrices (whether in two or three dimensions) can be used to implement protection mechanisms, as well as just model them. But they do not scale well. For instance, a bank with 50,000 staff and 300 applications would have an access control matrix of 15 million entries. This is inconveniently large. It might not only impose a performance problem but also be vulnerable to administrators' mistakes. We will usually need a more compact way of storing and managing this information. The two main ways of doing this are to use groups or roles to manage the privileges of large sets of users simultaneously, or to store the access control matrix either by columns (access control lists) or rows (capabilities, sometimes known as "tickets") or certificates .

Access Control List:

An ACL consists of a set of ACL entries. An ACL entry specifies the access permissions on the associated object for an individual user or a group of users as a combination of read, write and execute permissions.



Each file has an ACL associated with it. File F1 has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user A may read and write the file. The second entry says that any process owned by user B may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user A can read and write file F1. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID, that matters. File F2 has three entries in its ACL: A, B, and C can all read the file, and in addition B can also write it. No other accesses are allowed. File F3 is apparently an executable program, since B and C can both read and execute it. B can also write it.

Unix Operating System Security

In Unix (and its popular variant Linux), files are not allowed to have arbitrary access control lists, but simply rwx attributes for the resource owner, the group, and the world. These attributes allow the file to be read, written, and executed. The access control list as normally displayed has a flag to show whether the file is a directory; then flags r, w, and x for owner, group, and world respectively; it then has the owner's name and the group name.

A directory with all flags set would have the ACL:

drwxrwxrwx Alice Accounts

-rw-r——---Alice Accounts

This records that the file is not a directory; the file owner can read and write it; group members can read it but not write it; nongroup members have no access at all; the file owner is Alice; and the group is Accounts.

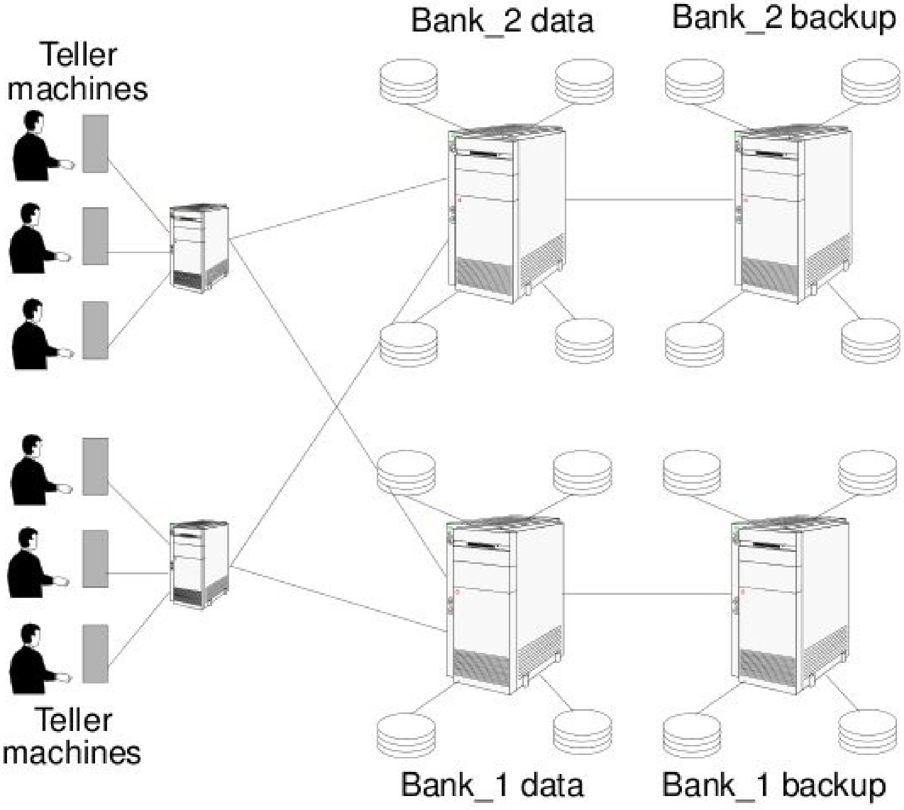
Chapter 8: Distributed Operating-system

Introduction, Goals Network architecture hardware and software concepts, Communication in distributed systems, ATM (Asynchronous Transfer Mode, Client- Server Model, RPC (Remote Procedure Call), Group communication, Processes and Processors in distributed System, taxonomy of MIMD computer system, Clock synchronization, scheduling in distributed system.

A distributed system:
– Multiple connected CPUs working together
– A collection of independent computers that appears to its users as a single coherent system
• Examples: parallel machines, networked machines

A distributed system is the collection of independent computers that appears to the user of the system as a single computer. This definition has two aspects. The first one deals with the hardware: the machine are autonomous. The second one deals with the software, the users think of the system as a single computer

Example of distributed system: Automatic banking (teller machine) system



Advantages:

- **Performance:** very often a collection of processors can provide higher performance (and better price/performance ratio) than a centralized computer.

- **Distribution:** many applications involve, by their nature, spatially separated machines (banking, commercial, automotive system).
- **Reliability** (fault tolerance): if some of the machines crash, the system can survive.
- **Incremental growth:** as requirements on processing power grow, new machines can be added incrementally.
- **Sharing of data/resources**: shared data is essential to many applications (banking, computer-supported cooperative work, reservation systems); other resources can be also shared (e.g. expensive printers).
- **Communication:** facilitates human-to-human communication.

Disadvantages:

Difficulties of developing distributed software: how should operating systems, programming languages and applications look like?

Networking problems: several problems are created by the network infrastructure, which have to be dealt with: loss of messages, overloading, ...

Security problems: sharing generates the problem of data security.

Design issues that arise specifically from the distributed nature of the application

Transparency

Communication

Performance & scalability

Heterogeneity

Openness

Reliability & fault tolerance

Security

Transparency:

How to achieve the single system image?

How to "fool" everyone into thinking that the collection of machines is a "simple" computer?

Access transparency

- local and remote resources are accessed using identical operations.

Location transparency

- users cannot tell where hardware and software resources (CPUs, files, data bases) are located; the name of the resource shouldn't encode the location of the resource.

Migration (mobility) transparency

- resources should be free to move from one location to another without having their names changed.

Replication transparency

- the system is free to make additional copies of files and other resources (for purpose of performance and/or reliability), without the users noticing.

Example: several copies of a file; at a certain request that copy is accessed which is the closest to the client.

Concurrency transparency

- the users will not notice the existence of other users in the system (even if they access the same resources).

• Failure transparency

- applications should be able to complete their task despite failures occurring in certain components of the system.

• Performance transparency

- load variation should not lead to performance degradation.

This could be achieved by automatic reconfiguration as response to changes of the load; it is difficult to achieve.

Communication:

Components of a distributed system have to communicate in order to interact. This implies support at two levels:

1. Networking infrastructure (interconnections & network software).
2. Appropriate communication primitives and models and their implementation:

• communication primitives:
- send
- receive

(Message Passing)
- remote procedure call (RPC)

• communication models
- **client-server communication**: implies a message exchange between two processes: the process which requests a service and the one which provides it;
- group muticast: the target of a message is a set of processes, which are members of a given group.

Performance and Scalability

Several factors are influencing the performance of a distributed system:
The performance of individual workstations.
The speed of the communication infrastructure
Extent to which reliability (fault tolerance) is provided (replication and preservation of coherence imply large overheads).
Flexibility in workload allocation: for example, idle processors (workstations) could be allocated automatically to a user's task.

Scalability

The system should remain efficient even with a significant increase in the number of users and resources connected:
- cost of adding resources should be reasonable;
- performance loss with increased number of users and resources should be controlled;
- software resources should not run out (number of bits allocated to addresses, number of entries in tables, etc.)

Architecture

Architectures for Parallel Programming Parallel computers can be divided into two categories: those that contain physical shared memory, called multiprocessors and those that do not, called multicomputers .A simple taxonomy is given in Fig.

Hardware and Software Concept:
Even though all distributed system consists of multiple cpus, there are several different ways the hardware can be organized in terms of how they are interconnected and how they communicate. Various classification schemes for multiple CPU computer system have been proposed. Most frequently cited taxonomy is Flynn's although it is fairly rudimentary.

Flynn proposed the following categories of computer systems:
Single instruction single data (SISD) stream: A single processor executes a single instruction stream to operate on data stored in a single memory. All traditional uniprocessor computers (i.e those having only one CPU) fall in this category, from personal computers to mainframes.
Single instruction multiple data (SIMD) stream: This type refers to array processors with one instruction unit that fetches an instruction, and then commands many data units to carry it out in parallel, each with its own data. These machines are useful for computation that repeat the same calculation on many sets of data.Vector and array processors fall into this category.
Multiple instruction single data (MISD) stream: A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure has never been implemented. No known computers fit in this model.
Multiple instruction multiple data (MIMD) stream: A set of processors simultaneously execute different instruction sequences on different data sets. Which essentially means a group of independent computers, each with its own program counter, program and data. All distributed system are MIMD.

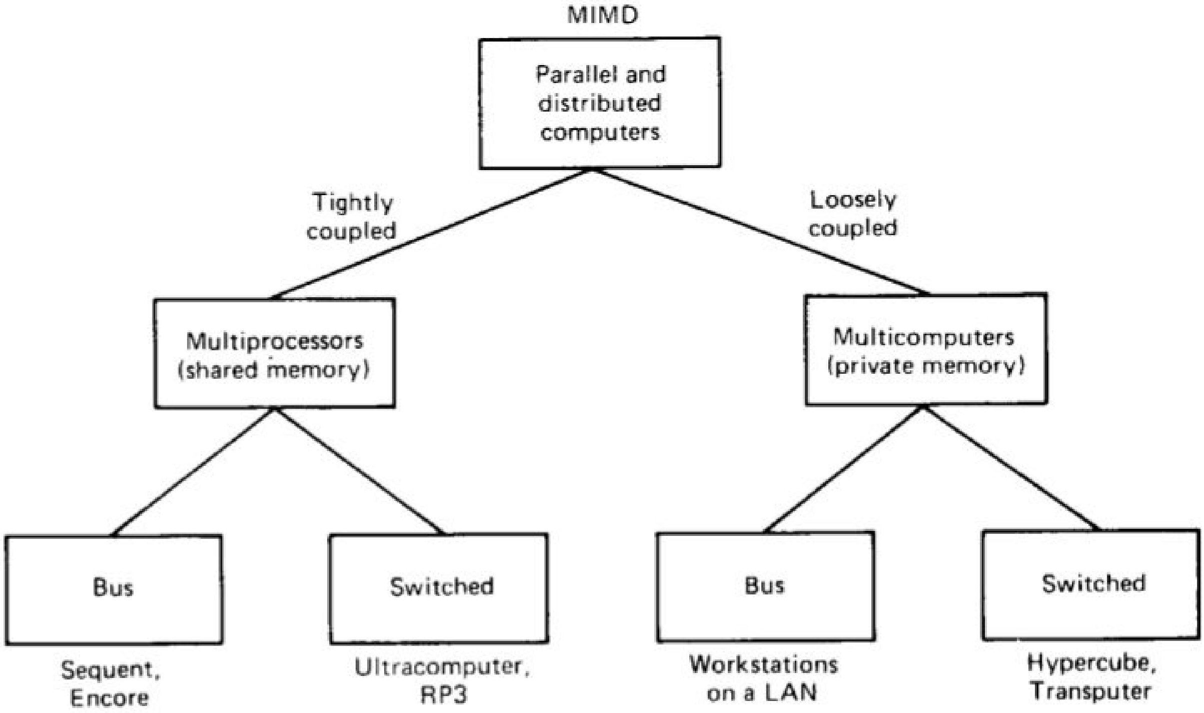


Fig: A taxonomy of parallel and distributed computer systems.

We divide all MIMD computers into two groups: those that have shared memory, usually called **Multiprocessors** and those that do not, sometimes called **Multicomputers.** The essential difference is this: in a multiprocessor, there is a single virtual address space that is shared by all CPUs. If any CPU writes, for example the address value 44 to address 1000, any other CPU subsequently reading from its address 1000 will get the value 44. All the machine share the same memory.

In contrast, in a multicomputer, every machine has its own private memory. If one CPU writes the value 44 to the address 1000, when another CPU reads the address 1000 it will get whatever value was there before. The write of 44 doesn't affect its memory at all. A common example of multicomputers is the collection of personal computers connected by the network.

Each of these category can be further divided into bus and switched based on architecture of the interconnection network.

By bus we mean that there is is single network, backbone, bus, cable, or other medium that connects all machine. Cable television uses a scheme like this, the cable company runs the cable down the street, and all the subscribers have taps running to it from their television sets.

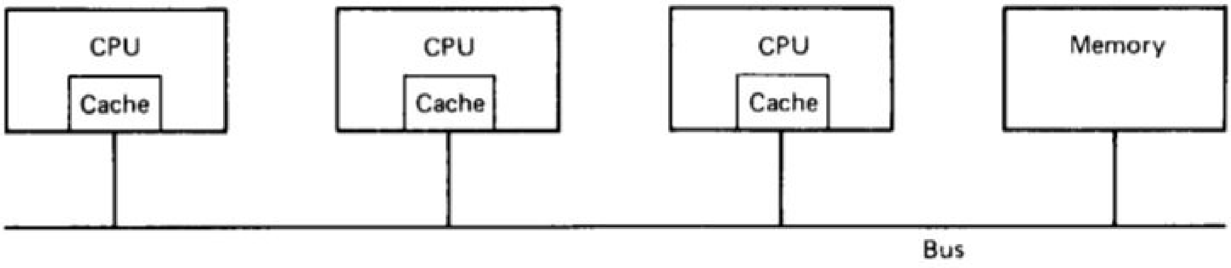
Switched system do not have a single backbone like cable television, instead there are individual wires from machine to machine, with many different wiring patterns in use. Messages move along the wires, with an explicit switching decisions made at each step to route the message along one of the out going wires. The world-wide public telephone system is organized in this way.

In a tightly coupled system, the delay experienced when a message is sent from one computer to other another is short, and the data rate is high; that is the number of bits per second that can be transferred is large. In a loosely coupled system the opposite is true: the inter-machine delay is large and the date rate is low. For example two CPUs chips on the same printed board and connected by wires are likely to be tightly coupled, whereas two computers connected by a 2400 bits/sec modem over the telephone system are certain to be loosely coupled.

Tightly coupled system tend to be used more as parallel systems (working on a single problem) and loosely coupled system tend to be used as distributed system (working on many unrelated problems).

Bus based multiprocessor

Bus based multiprocessor consists of some number of CPUs all connected to a common bus, along with a memory module.



A simple configuration is to have a high speed backplane or motherboard into which CPU or memory cards can be inserted. A typical bus has 32 or 64 address lines, 32 or 64 data lines, and perhaps 32 or more control lines, all of which operate in parallel.

The problem with this scheme is that with as few as 4 or 5 CPUs , the bus will usually be overloaded and performance will drop drastically. The solution is to add the high speed cache memory between the CPUs and the bus as shown in the fig. The cache holds the most recently accessed words. All memory requests goes though the cache. If the word requested is in the cache, the cache itself responds to the CPU and no bus request is made. If the cache is large enough, the probability of success, called the hit rate, will be high and the amount of bus traffic per CPU will drop dramatically, allowing many more CPUs in the system.

However the introduction of cache also bring the serious problem with it suppose that two CPUs A and B each read the same word into their respective caches. Then A overwrites the word. When B next reads that word, it gets the old value from its cache, not the value A just wrote. The memory is now incoherent and the system is difficult to program.

One of the solution for it is to implement write-through cache and snoopy cache. In write-through cache, cache memories are designed so that whenever a word is written to the cache, it is written through to memory as well. In addition, all caches constantly monitor the bus. Whenever a cache sees a write occurring to a memory address present in its cache, it either removes that entry from its cache or updates the cache entry with the new value. Such a cache is called **snoopy cache.** because it is always snooping on the bus.

Using it, it is possible to put about 32 or possibly 64 CPUs on a single bus.

Switched Multiprocessor

To build a multiprocessor with more than 64 processors, a different method is needed to connect the CPUs with the memory. Two switching techniques are employed for it.

a. Crossbar Switch
b. Omega Switch

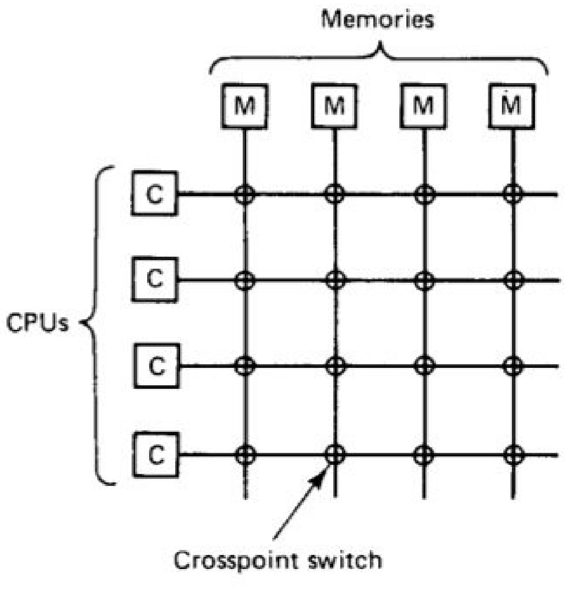


Fig a). Crossbar Switch

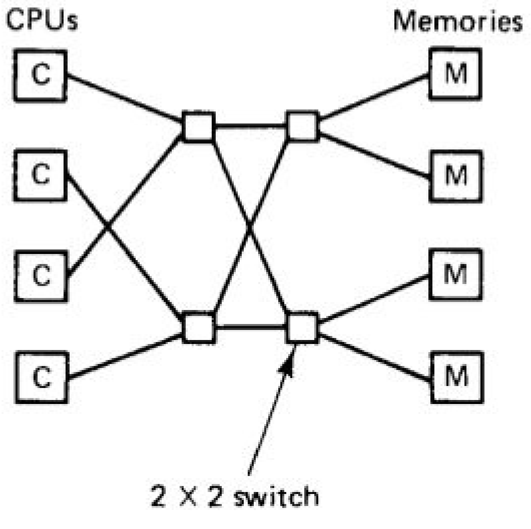


Fig b). An Omega Switching Network

Crossbar Switch:

Memory is divided into the modules and are connected to the CPUs with the crossbar switch. Each CPU and each memory has a connection coming out of it, as shown. At every intersection is a tiny electronic crosspoint switch that can be opened and closed in hardware. When a CPU wants to access a particular memory, the crosspoint switch connecting them is closed, to allow the access to take place. If two CPUs try to access the same memory simultaneously, one of them will have to wait.
The downside of the crossbar switch is that with n CPUs and n memories, n^2 crosspoint switches are needed. For large n this number can be prohibitive.

Omega Switching network

It contains 2x2 switches, each having two inputs and two outputs. Each switch can route either input to either output. A careful look at the figure will show that with proper setting of the switches, every CPU can access every memory. In general case, with n CPUs and n memories, the omega network requires $\log_2 n$ switching stages, each containing n/2 switches, for a total of $(n\log_2 n)/2$ switches.

Bus based multicomputers

Building a multicomputer (i.e no shared memory) is easy. Each CPU has a direct connection to its own local memory. The only problem left is how CPUs communicate with each other.

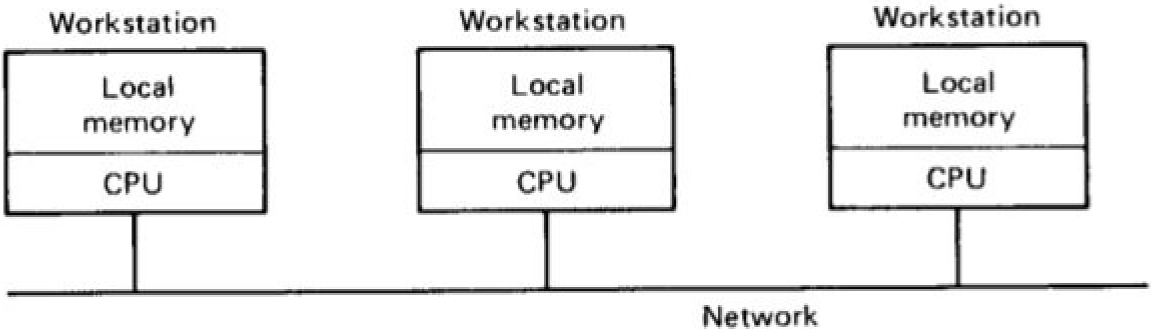


Fig. A multicomputer consisting of workstations on a LAN.

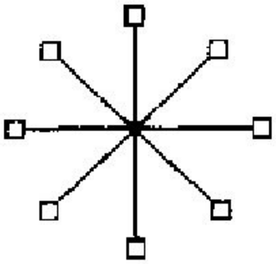
CPUs are connected to each other via LAN (10 or 100Mbps).

Switch Multicomputers

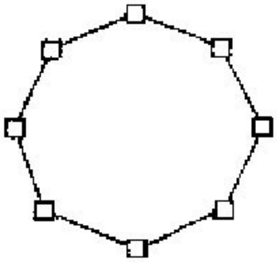
Switched multicomputers do not have a single bus over which all traffic goes. Instead, they have a collection of point-to-point connections. In we see two of the many designs that have been proposed and built: a grid and a hypercube. A grid is easy to understand and easy to lay out on a printed circuit board or chip. This architecture is best suited to problems that are two dimensional in nature (graph theory, vision,etc.)
Another design is a hypercube which is an n -dimensional cube. One can imagine a 4-dimensional hypercube as a pair of ordinary cubes with the corresponding vertices connected, as shown in fig. Similarly, a 5-dimensional hypercube can be represented as two copies of Fig.4.5(b), with the

corresponding vertices connected, andso on. In general, an n -dimensional hypercube has 2 n vertices, each holding one CPU. Each CPU has a fan-out of n , so the interconnection complexity grows logarithmically with the number of CPU.

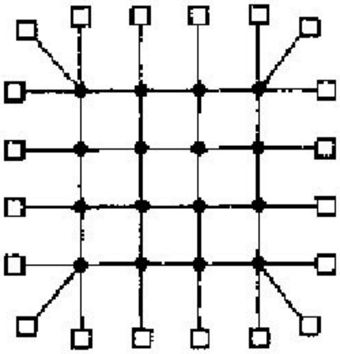
Various interconnection have been proposed and built, but all have the property that each CPU has direct and exclusive access to its own private memory.



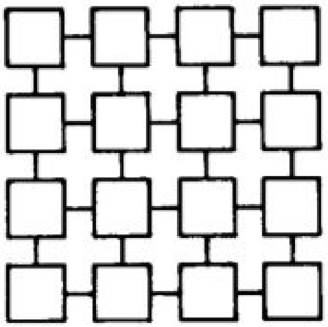
A single switch



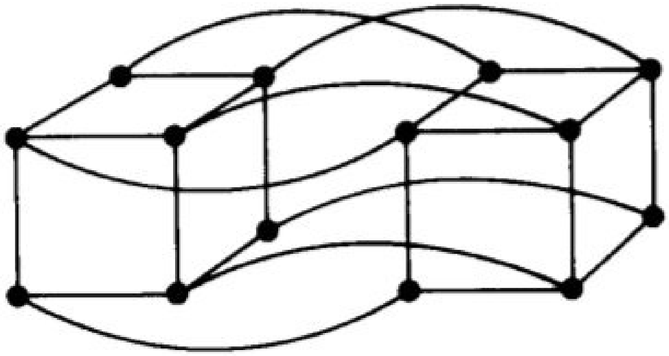
A ring



A grid



(a) A grid



(b) 4-D Hypercube

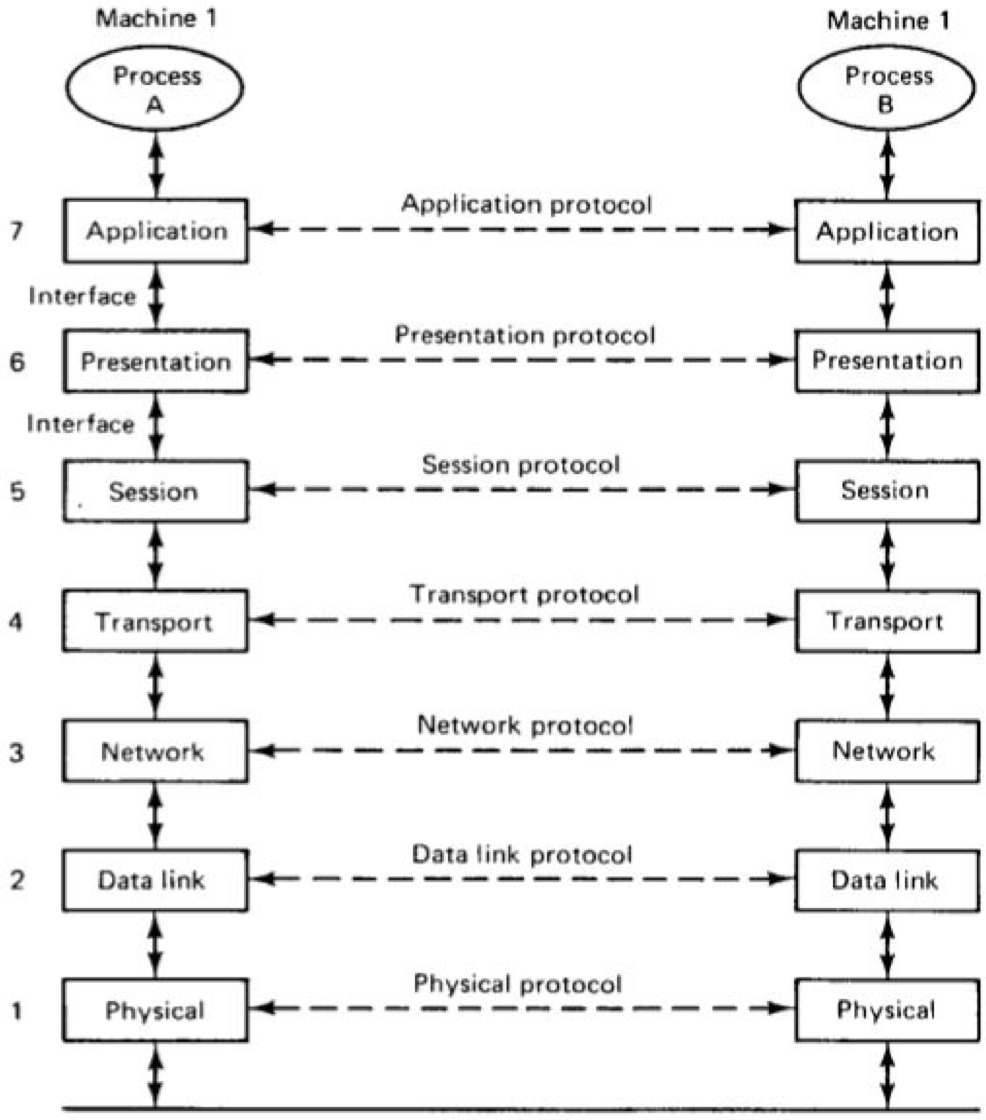
NUMA Machines

Various researchers have proposed intermediate designs that try to capture the desirable properties of both architectures. Most of these designs attempt to simulate shared memory on multicomputers. All of them have the property that a process exe-cuting on any machine can access data from its own memory without any delay,whereas access to data located on another machine entails considerable delay and over-head, as a request message must be sent there and a reply received. Computers in which references to some memory addresses are cheap (i.e., local)and others are expensive (i.e., remote) have become known as NUMA (NonUniform Memory Access) machines. Our definition of NUMA machines is somewhat broader than what some other writers have used. We regard any machine which presents the programmer with the illusion of shared memory, but implements it by sending mes-sages across a network to fetch chunks of data as NUMA.

Communication in Distributed System:

Layered Communication:

All communication in distributed system is based on message passing. When a process A wants to communicate with a process B it first builds a message in its own address space. Then it executes a system call that causes the operating system to fetch the message and send it over the network to B. To make it easier to deal with numerous levels and issues involved in communication, the international Standards Organization has developed a reference model, that clearly identifies the various level involved, gives them standard names and points out which level should do which job. This model is called OSI (Open System Interconnection Reference Model).



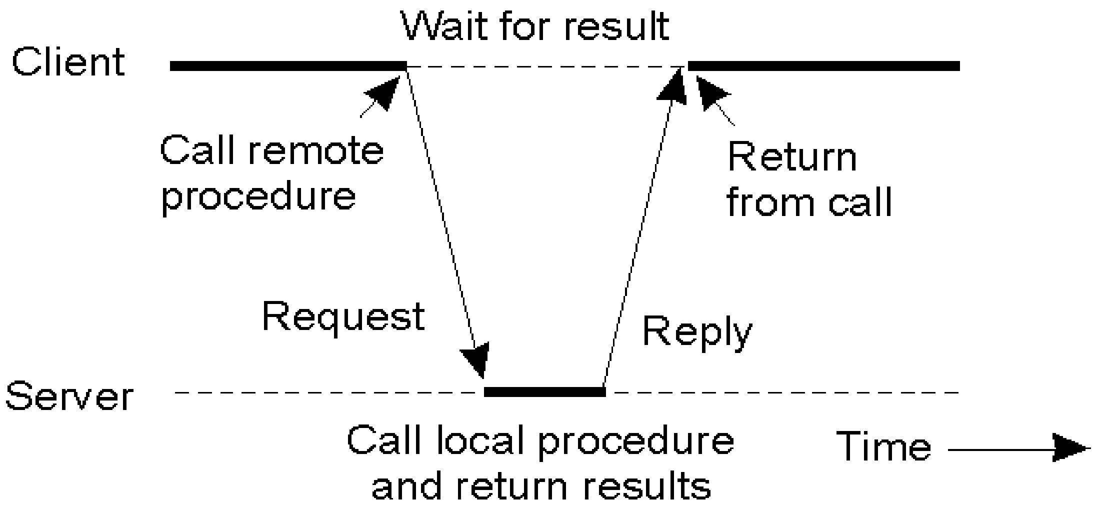
Network

RPC:

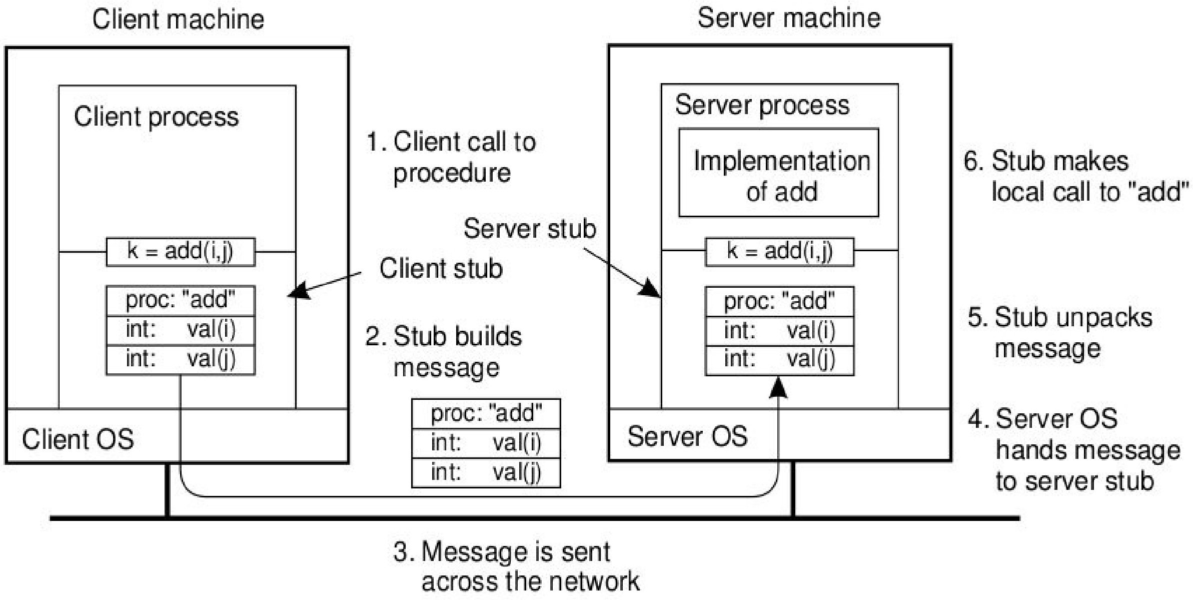
A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation.

Principle of RPC:

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging parameters is called marshalling
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
- Simplifies programmer task



Example of RPC:



a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub, to a local call to the server procedure without either client or server being aware of the intermediate steps.

ATM

The "asynchronous" in ATM means ATM devices do not send and receive information at fixed speeds or using a timer, but instead negotiate transmission speeds based on hardware and information flow reliability. The "transfer mode" in ATM refers to the fixed-size cell structure used for packaging information.

ATM transfers information in fixed-size units called cells. Each cell consists of 53 octets, or bytes as shown in Fig.

Header 5 bytes	**Payload** 48 bytes

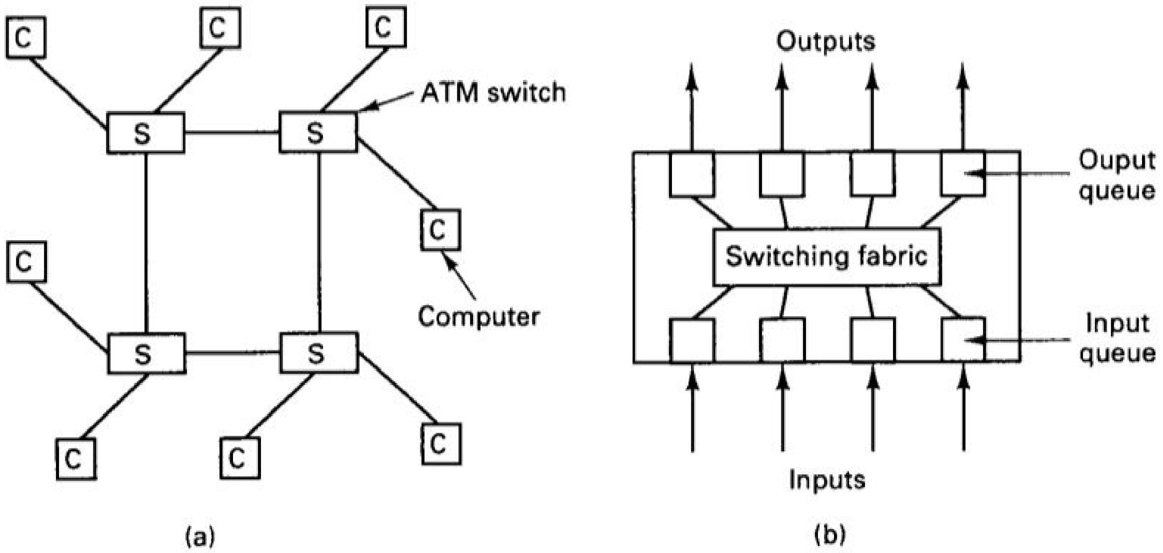
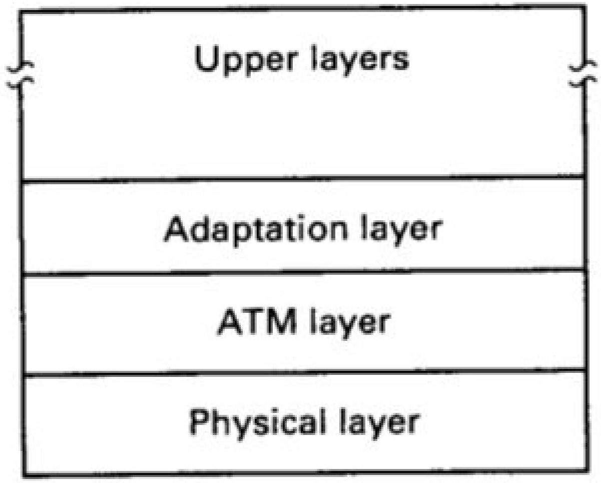


Figure illustrate the ATM network with 4 switches. Each of these switches has 4 ports, each used for both input and output lines. The inside of the generic switch is as shown in fig b.



ATM Layer:

Client-server Model:

The idea behind this model is to structure the operating-system as a group of cooperating process, called servers, that offer services to the users, called clients. The client and server normally all run the same microkernel, with both clients and severs running as user processes. A machine may run a single process or it may run multiple clients, multiple servers or mixture of both.

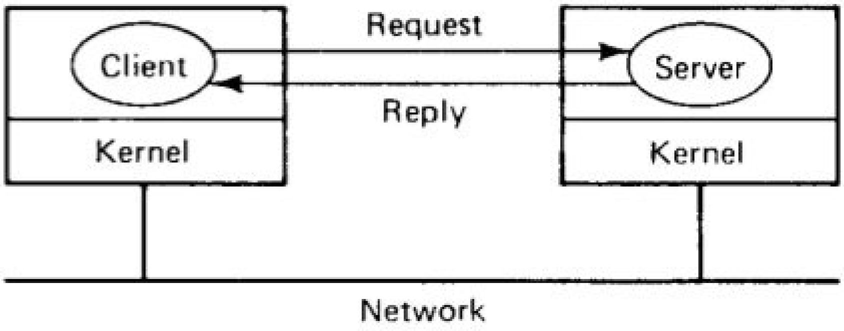


Fig. Client and server model.

To avoid the considerable overhead of the connection oriented protocol such as TCP and OSI, the client-server model is usually based on a simple connectionless request/reply protocol. The client sends a request message to the server asking for some services (eg. read a block of a file). The server does the work and returns the data requested or an error-code indicating why the work could not be performed as shown in the fig above.

References:
1. Modern Operating System by: Andrew S. Tannenbaum
2. Operating System Concepts by: Abraham Silberschatz, G. Gange, P.B Galvin
3. wikipedia