

Chapter 1: Operating System Overview

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.

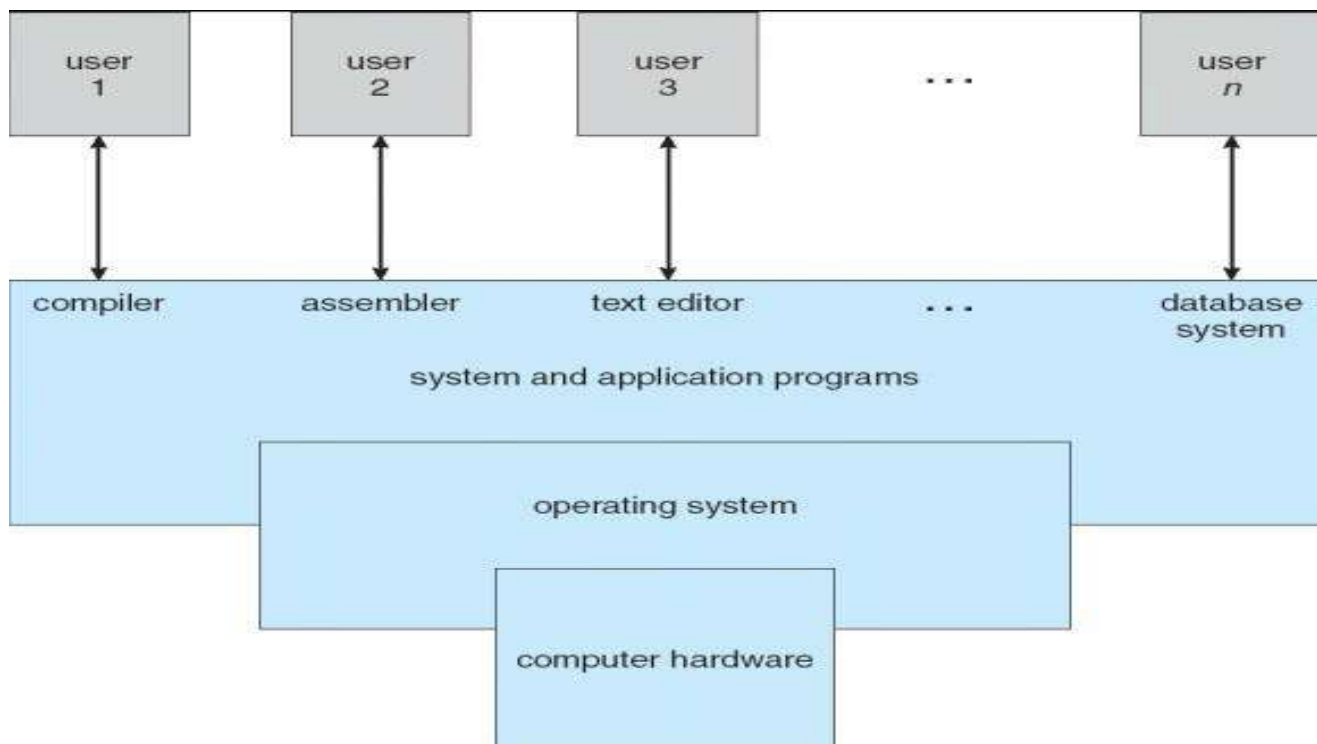


Fig: Abstract view of the components of a Computer System.

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 above.

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.

Hence, an operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware in other words, “**The software that controls the hardware.**” Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, OS/2, MacOS, VMS, MVS, and VM.

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 from 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.

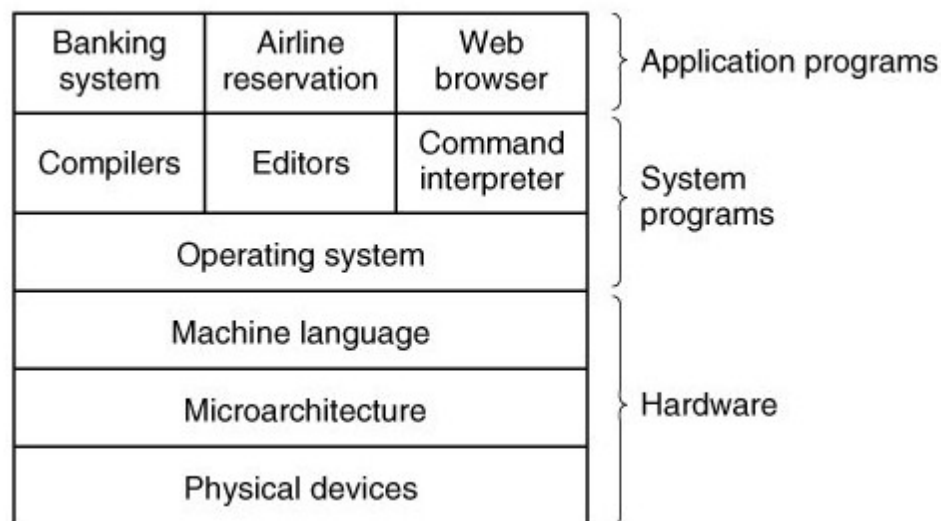


Fig: Computer system consists of Hardware, system program and application program

- **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 below 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

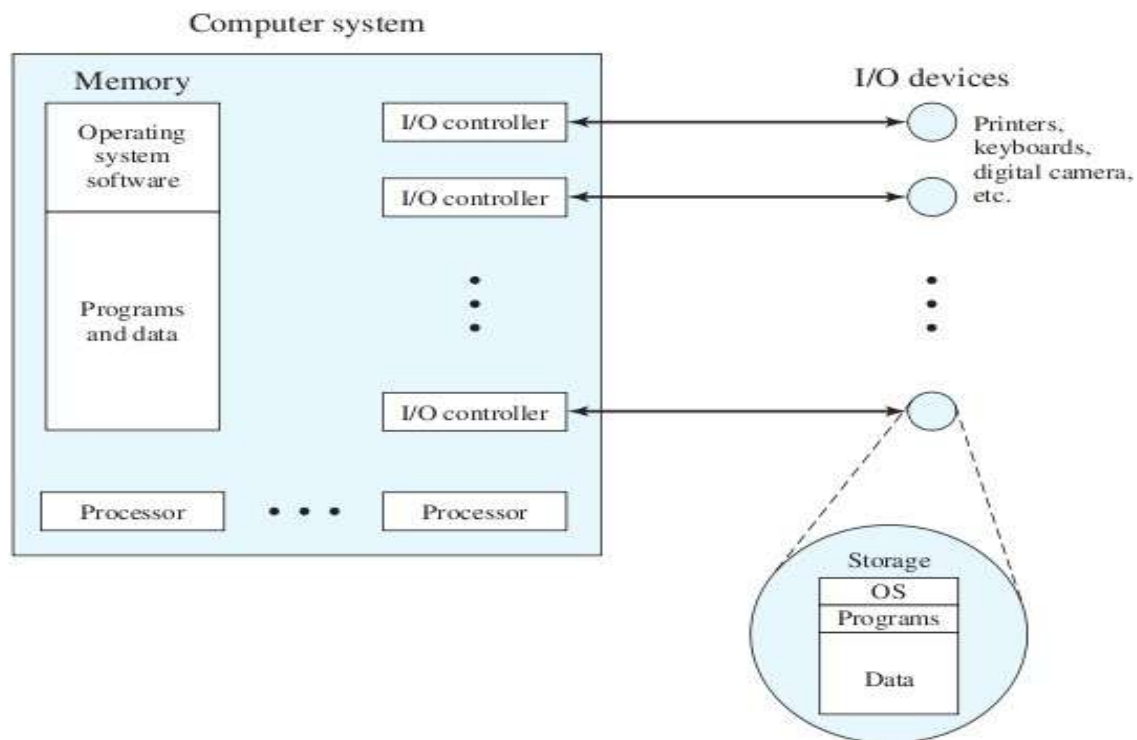


Fig: The Operating system as Resource manger

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

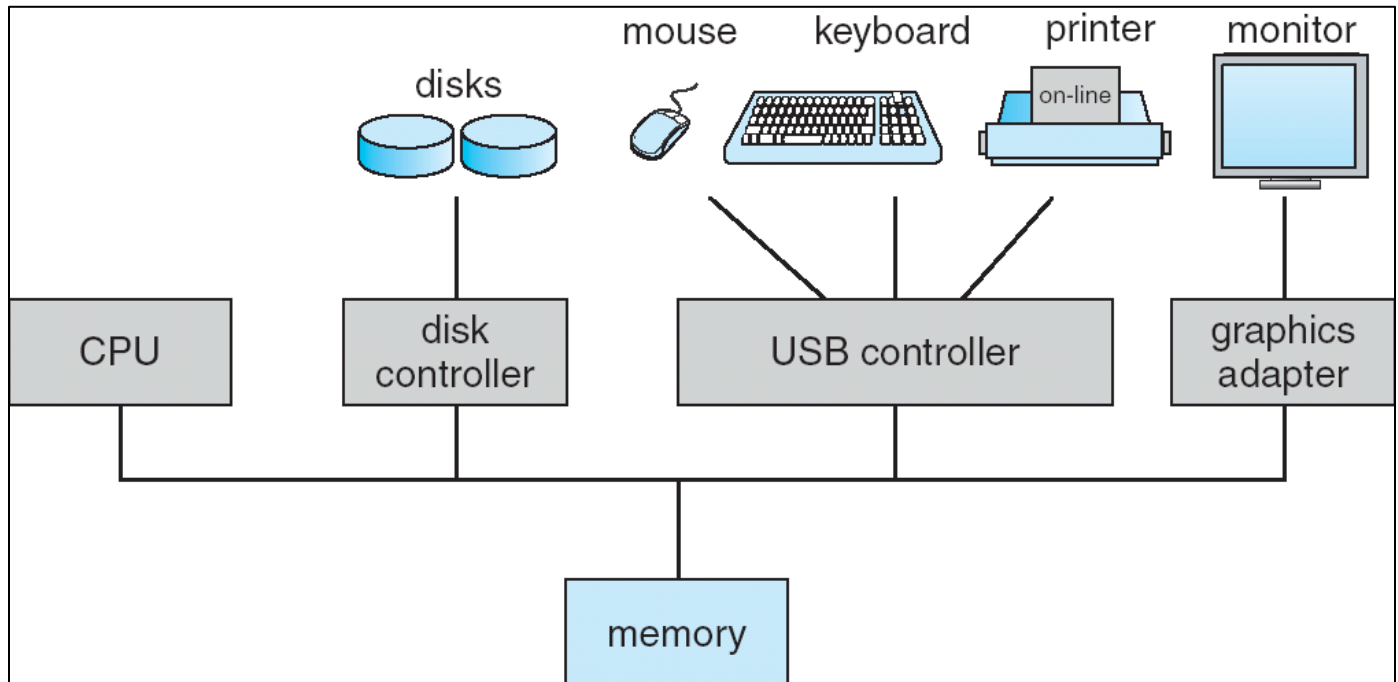


Fig: A Modern Computer System

Function of Operating system:

- Implementing the user interface
- Sharing hardware among users
- Allowing users to share data among themselves
- Preventing users from interfering with one another
- Scheduling resources among users
- Facilitating input/output
- Recovering from errors
- Accounting for resource usage
- Facilitating parallel operations
- Organizing data for secure and rapid access
- Handling network communications.

These above functions can be summarized as,

- Memory management function
- Processors management function
- I/O Device management function
- File management function

Objectives

- *Convenience:* An operating system makes a computer more convenient to use.
- *Efficiency:* An operating system allows the computer system resources to be used in an efficient manner.
- *Ability to evolve:* An operating system should permit the effective development, testing, and introduction of new system function without interfering with the service.

Services of operating system

- *Program creation:* Operating system provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs.
- *Program execution:* A number of tasks need be performed to execute a program. The instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be performed. The operating system handles all the tasks.
- *Access to I / O devices:* Each I / O requires its own peculiar set of instructions, or control signals for operation. The operating system takes care of the details.
- *Controlled access to files:* The operating system deals with the file format of the storage medium and the nature of I / O device.
- *System Access:* The operating system controls the access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflict.
- *Error detection and response:* The operating system must take the response that clears the error condition with the least impact on running applications.
- *Accounting:* An operating system collects usage statistics for various resources and monitors performance parameters such as response time to improve performance.

Evolution of Operating System:

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

1. Serial Processing:

- Early computer from late 1940 to the mid 1950
- The programmer interacted directly with the computer hardware.
- These machines 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 systems 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 allotted 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.

2. 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.

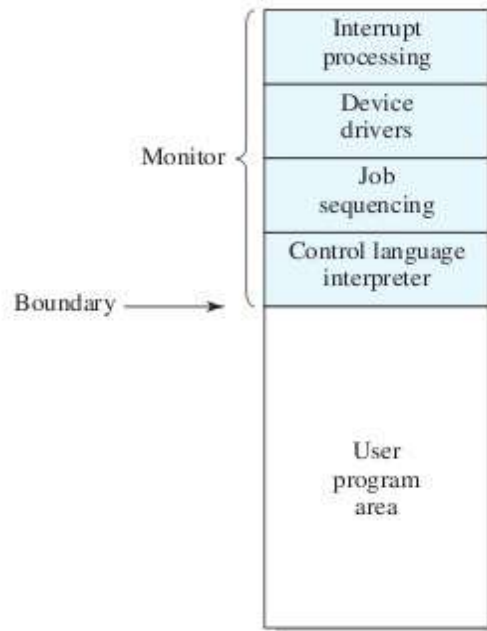


Fig.: Memory Layout for resident memory

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.

3. 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 waiting 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.

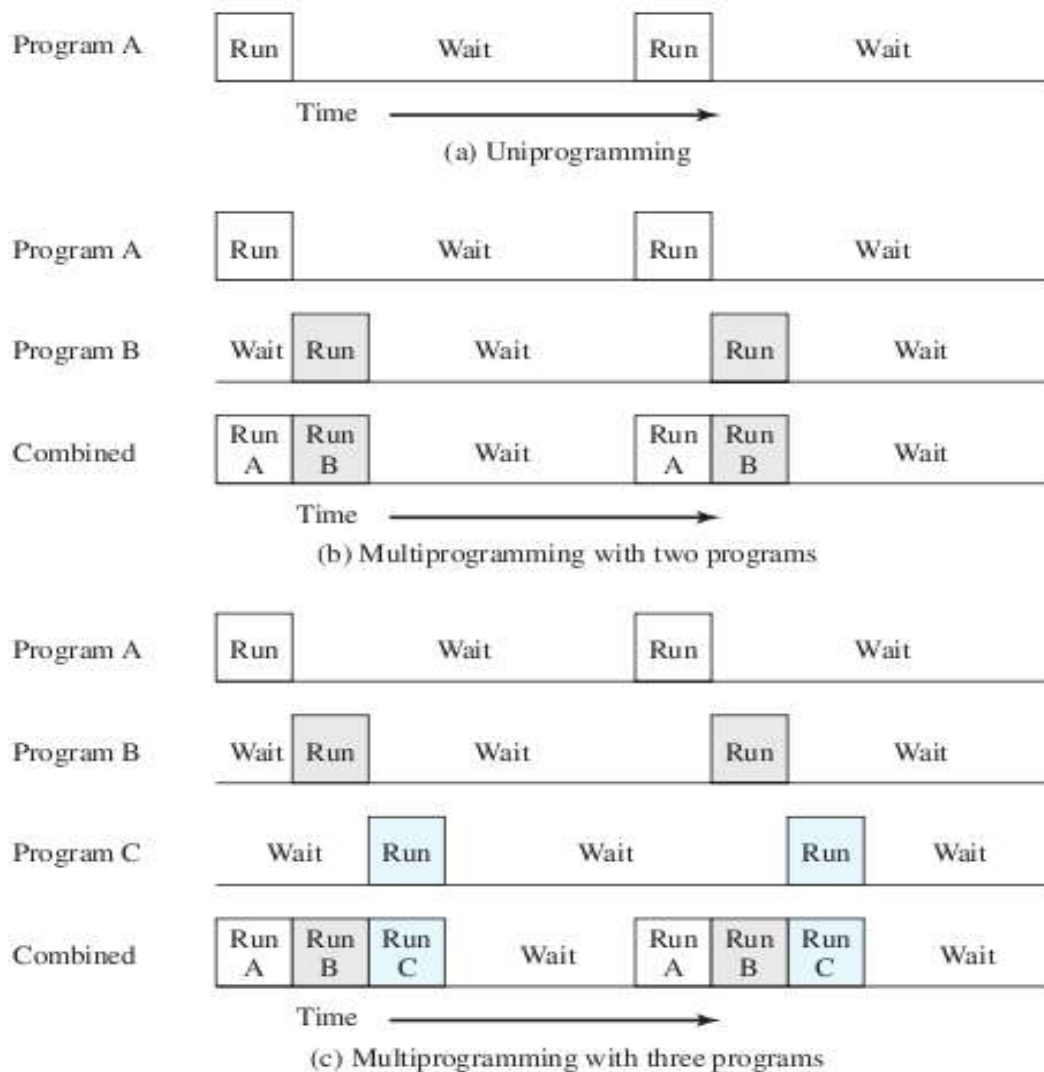


Fig: Multiprogramming example

4. 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.

The History of Operating Systems

Operating systems have evolved through a number of distinct phases or generations, which corresponds roughly to the decades.

The 1940's - First Generations

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programs were often entered one bit at time on rows of mechanical switches (plug boards).

Programming languages were unknown (not even assembly languages). Operating systems were unheard of.

The 1950's - Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

The 1960's - Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.

For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.

Another major feature in third-generation operating system was the technique called SPOOLing (simultaneous peripheral operations on line). In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.

Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected) terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multiprogram large number of simultaneous interactive users.

Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it became possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

OPERATING SYSTEM

Compiled by Er. Arjun Aryal

Chapter 2

Processes

- ***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.
- Generally, Process includes:
 - Current value of Program Counter (PC)
 - Contents of the processors registers
 - Value of the variables
 - The process stack contains temporary data such as subroutine parameter, return address, and temporary variables.
 - A data section that contains global variables.
- **Program:** A set of instructions a computer can interpret and execute

Process and Program

- ***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

Difference between a process and a program

This following 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 ($\text{prod} = \text{prod} * i$) 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

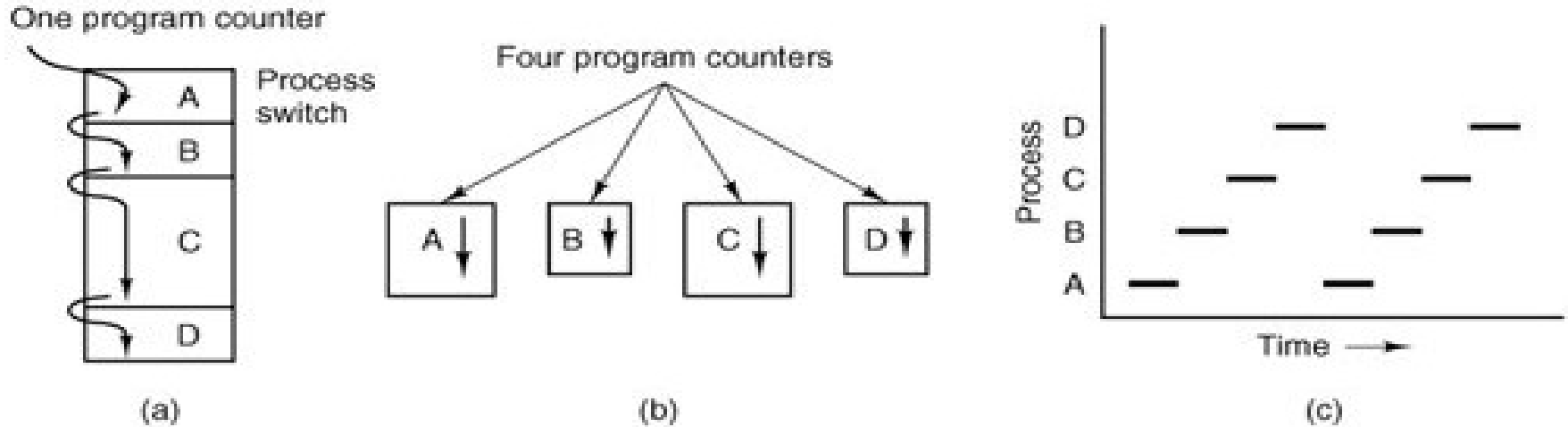


Fig:1.1: (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

- * 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:

- System initialization.
 - Execution of a process creation system call by a running process.
 - A user request to create a new process.
 - 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.

Process Termination

Conditions which terminate processes:

1. Normal exit (voluntary): Most processes terminate because they have done their job. This call exists in UNIX
2. Error exit (voluntary): When process discovers a fatal error. For example, a user tries to compile a program that does not exist.
3. Fatal error (involuntary): An error caused by process due to a bug in program for example, executing an illegal instruction, referring non-existing memory or dividing by zero.
4. Killed by another process (involuntary): A process executes a system call telling the Operating Systems to terminate some other process. In UNIX, this call is kill. In some systems when a process kills all processes it created are killed as well.

Process Hierarchies

- Parent creates a child process, child processes can create its own process
Forms a hierarchy
 - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
 - all processes are created equal

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.

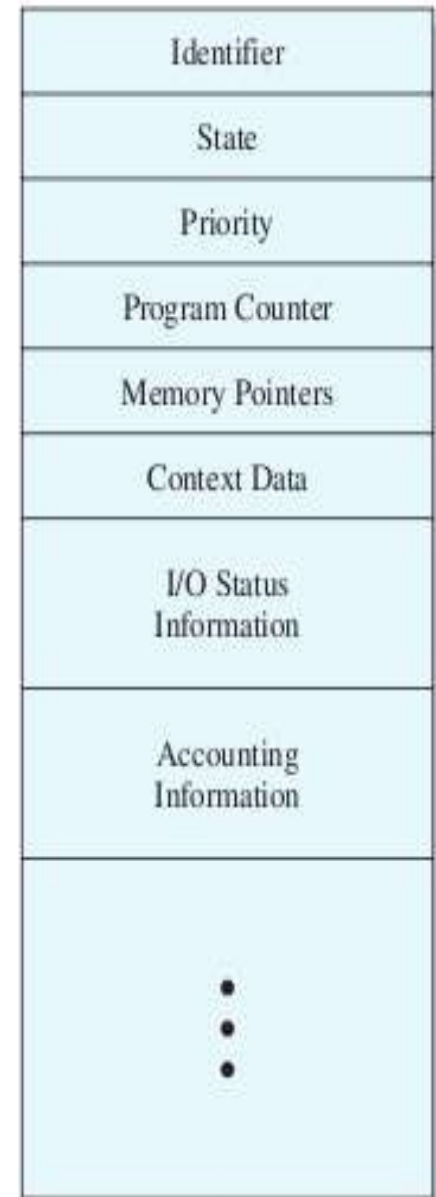
PCB Contains

- **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.

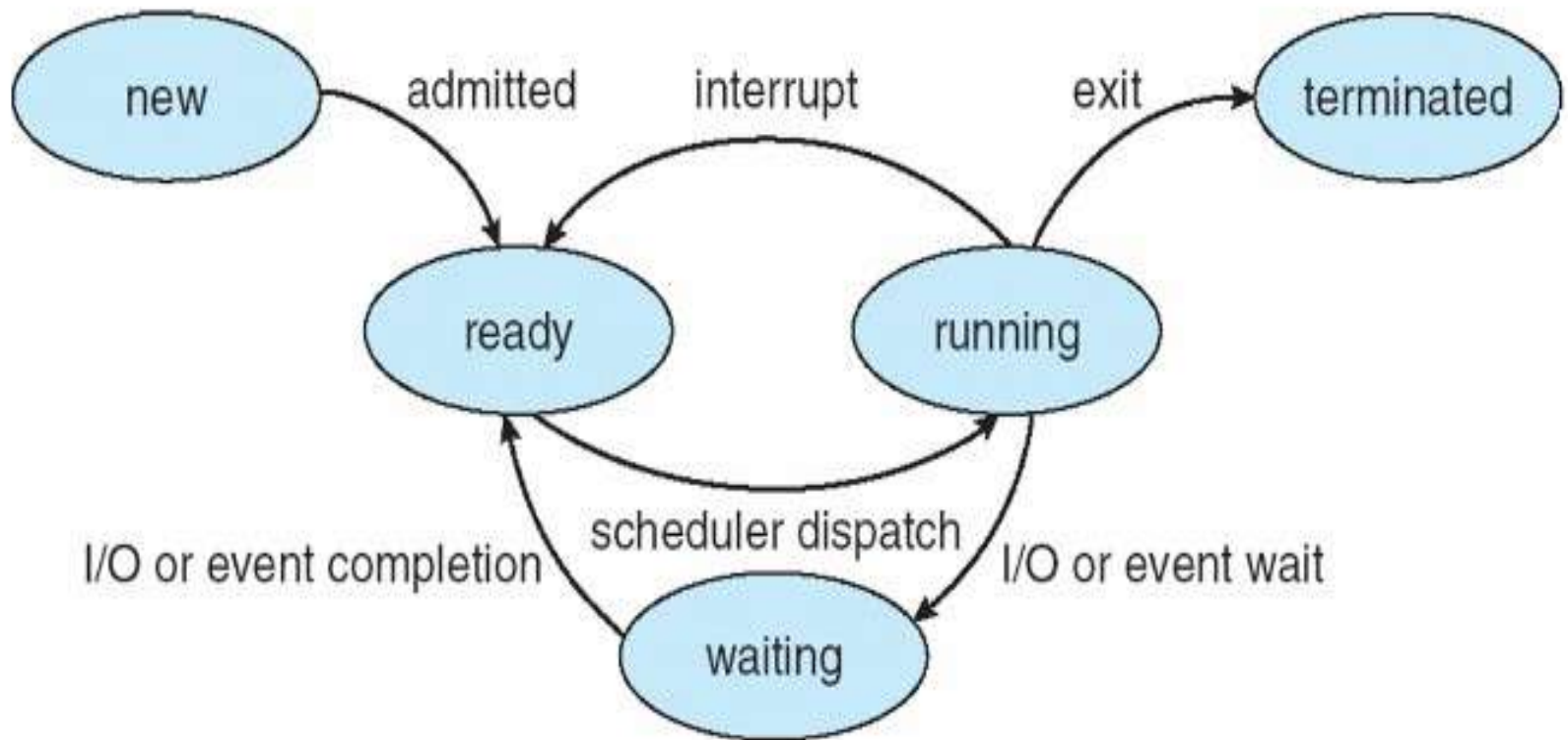
Identifier
State
Priority
Program Counter
Memory Pointers
Context Data
I/O Status Information
Accounting Information
⋮

PCB Contains cntd..

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



Process States



Compiled by Engr. Arjun Arora, Lec 9, PPT
Fig. Process state Transition diagram

Process States

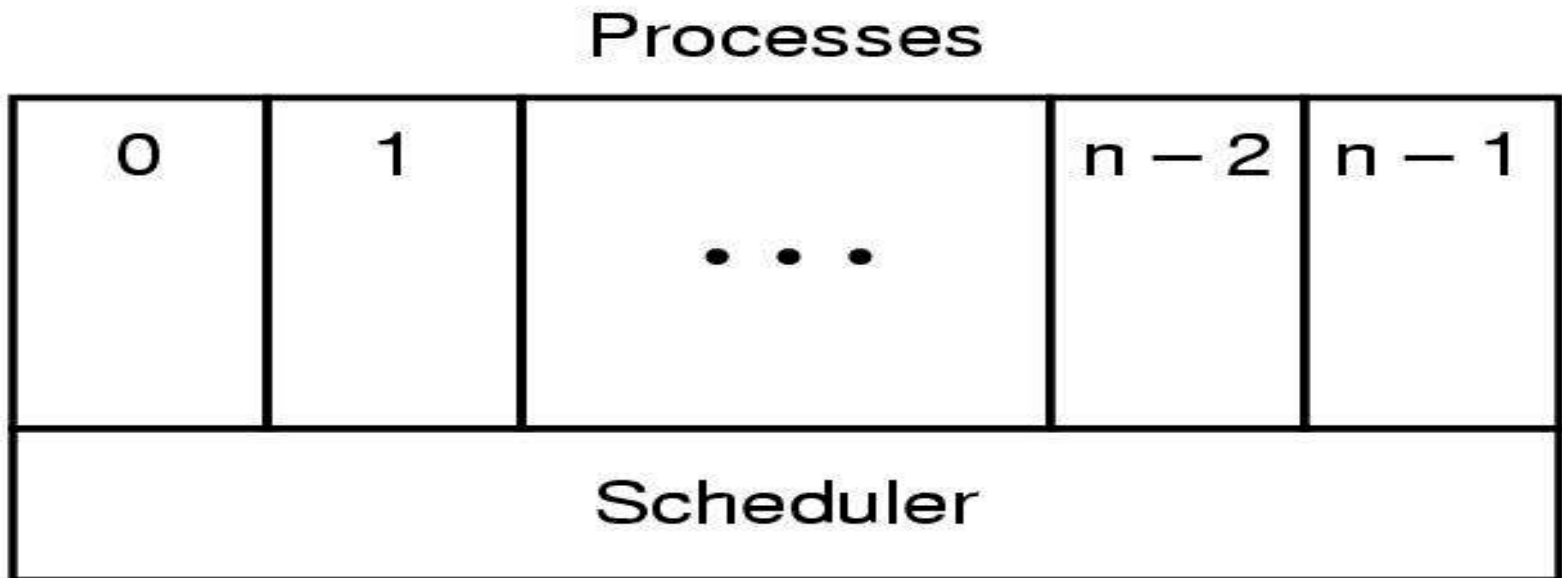
- Each process may be in one of the following states:
 - **New**: The process is being created.
 - **Running**: Instructions are being executed.
 - **Waiting**: The process is waiting for some event to occur (such as I/O completion or reception of a signal)
 - **Ready**: The process is waiting to be 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

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters 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 Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Implementation of Process:



- Lowest layer of process-structured OS
 - handles interrupts, scheduling
- Above that layer are sequential processes

Interrupt handling and scheduling

- Can be summarize as :
 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.

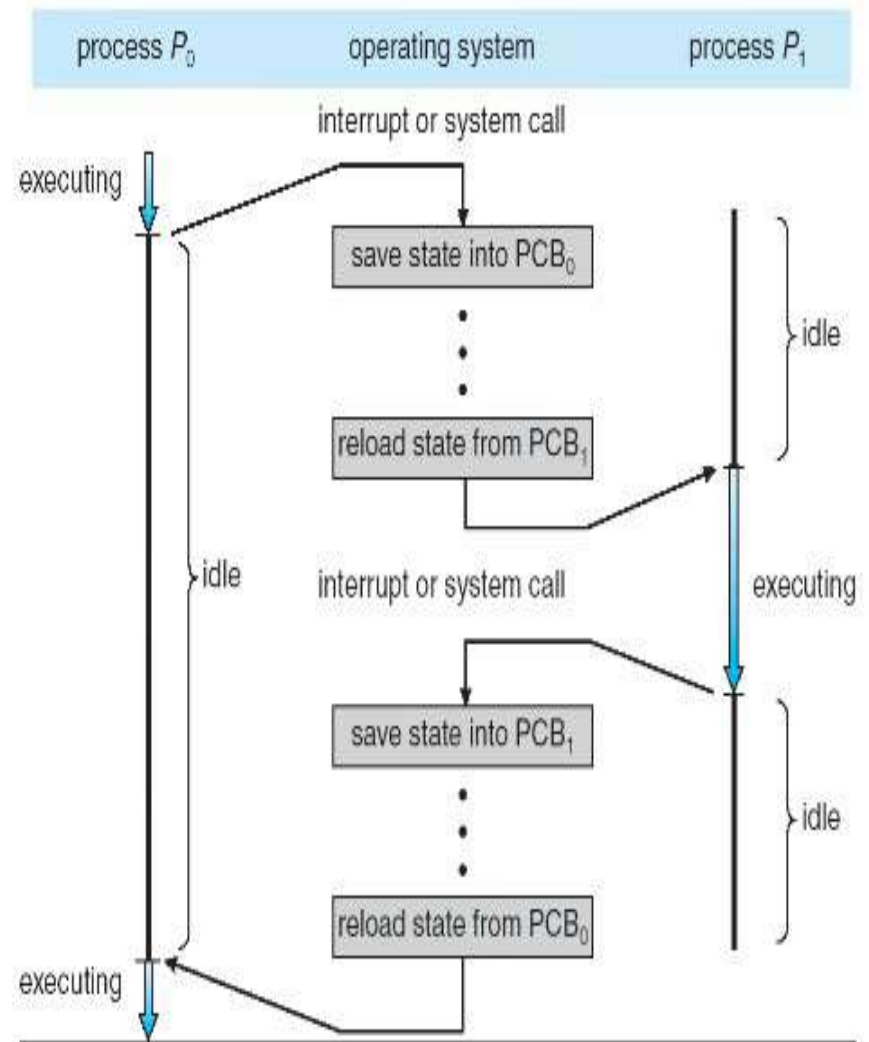
The above points lists the Skeleton of what the lowest level of the operating system does when an interrupt occurs.

Interrupt Vector

- 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

Context Switching:

- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as context switch.
- When a context switch occurs,
 - the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run
- 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.



The OS Kernel

- Basic set of objects, primitives, data structures, processes
- Rest of OS is built on top of kernel
- Kernel provides *mechanism* to implement various *policies*
 - process/thread management
 - interrupt/trap handling
 - resource management
 - input/output
- The Purpose is
 - To make the rest of the operating system completely independent of the hardware and thus highly portable.

Kernal Vs OS

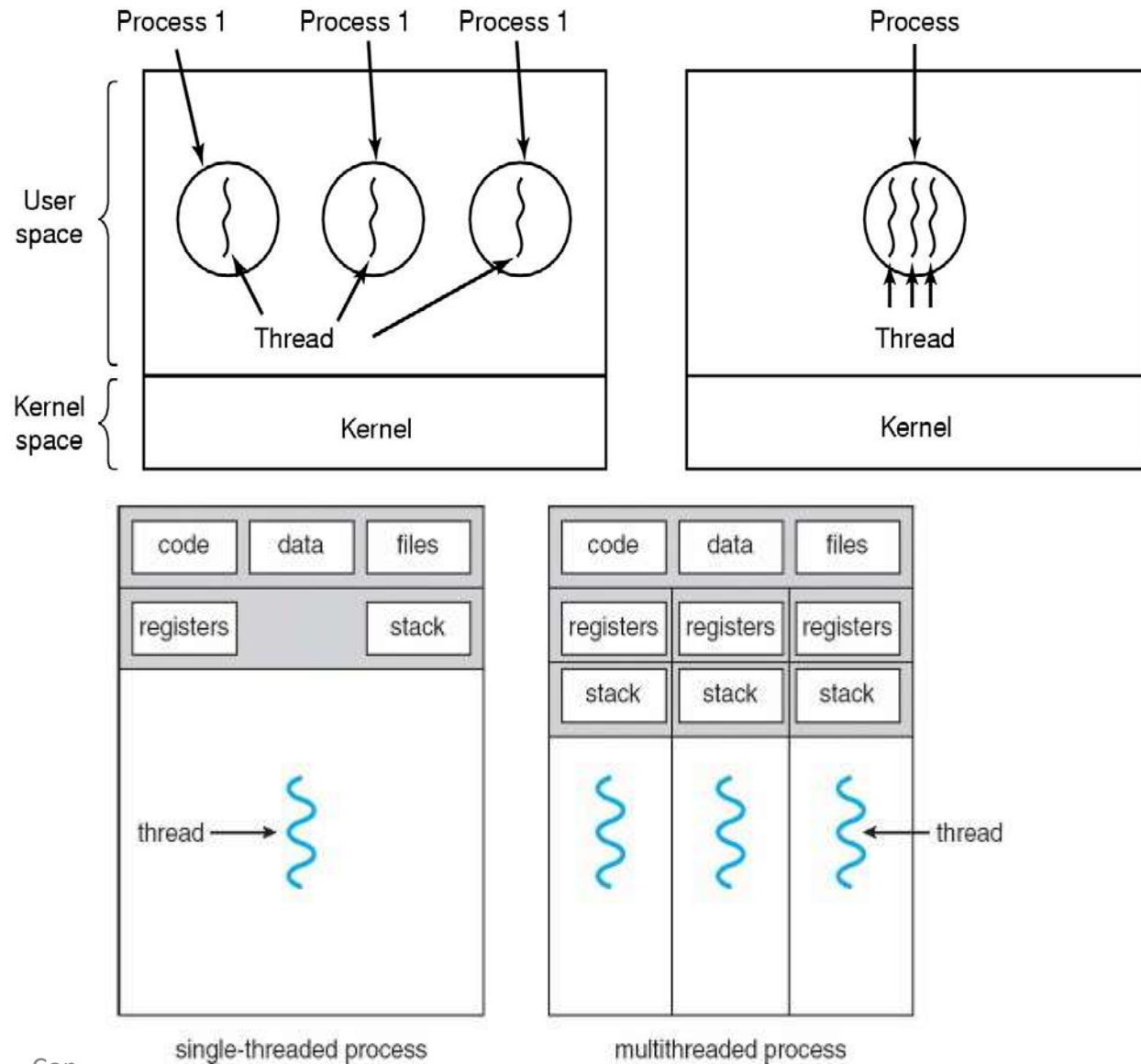
- The technical definition is "platform that consists of specific set of libraries and infrastructure for applications to be built upon and interact with each other". A kernel is an operating system in that sense.
- The end-user definition is usually something around "a software package that provides a desktop, shortcuts to applications, a web browser and a media player". A kernel doesn't match that definition.

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.
- 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.

Threads

- 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. Figure aside illustrate the difference between single threaded process and a multi-threaded process.



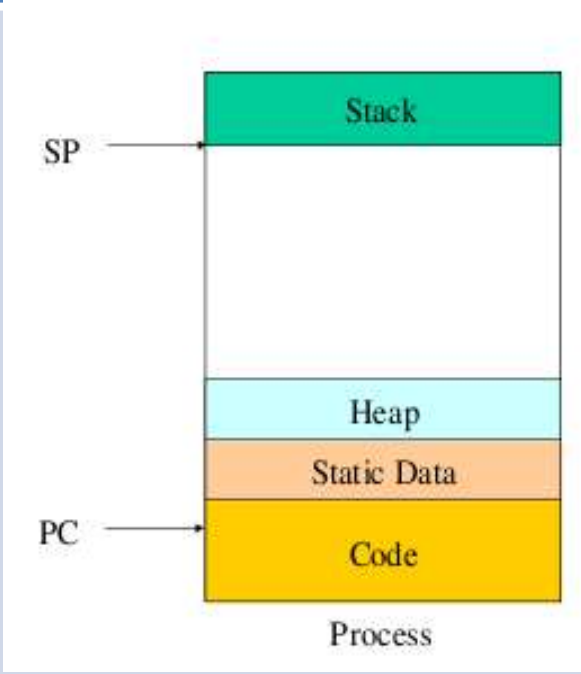
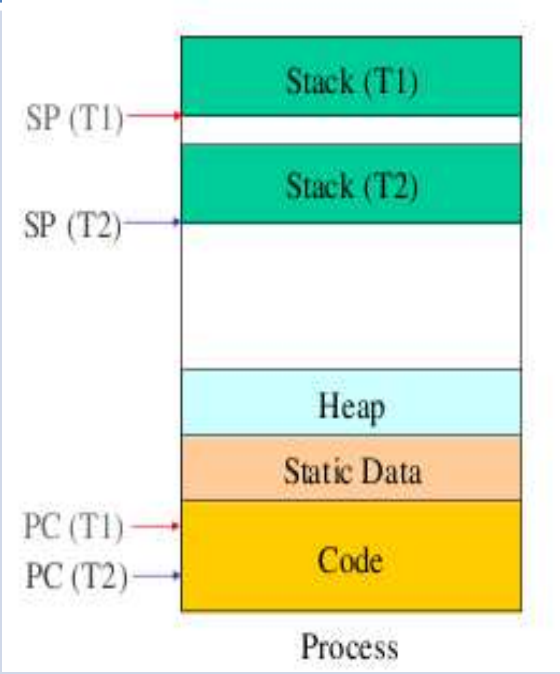
Multithreading

- **Multithreading refers to the ability of an operating system to support multiple threads of execution within a single process.**
- **Within a process, there may be one or more threads, each with the following:**
 - **A thread execution state**
 - **A saved thread context when not running, one way to view a thread is as an independent program counter operating within a process.**
 - **An execution stack**
 - **Some per-thread static storage for local variables**
 - **Access to the memory and resources of its process, shared with all other threads in that process.**

Benefits of Multi-threading

- **Responsiveness:** Multithreaded 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 writhing 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. Multithreading on a multi-CPU increases concurrency.

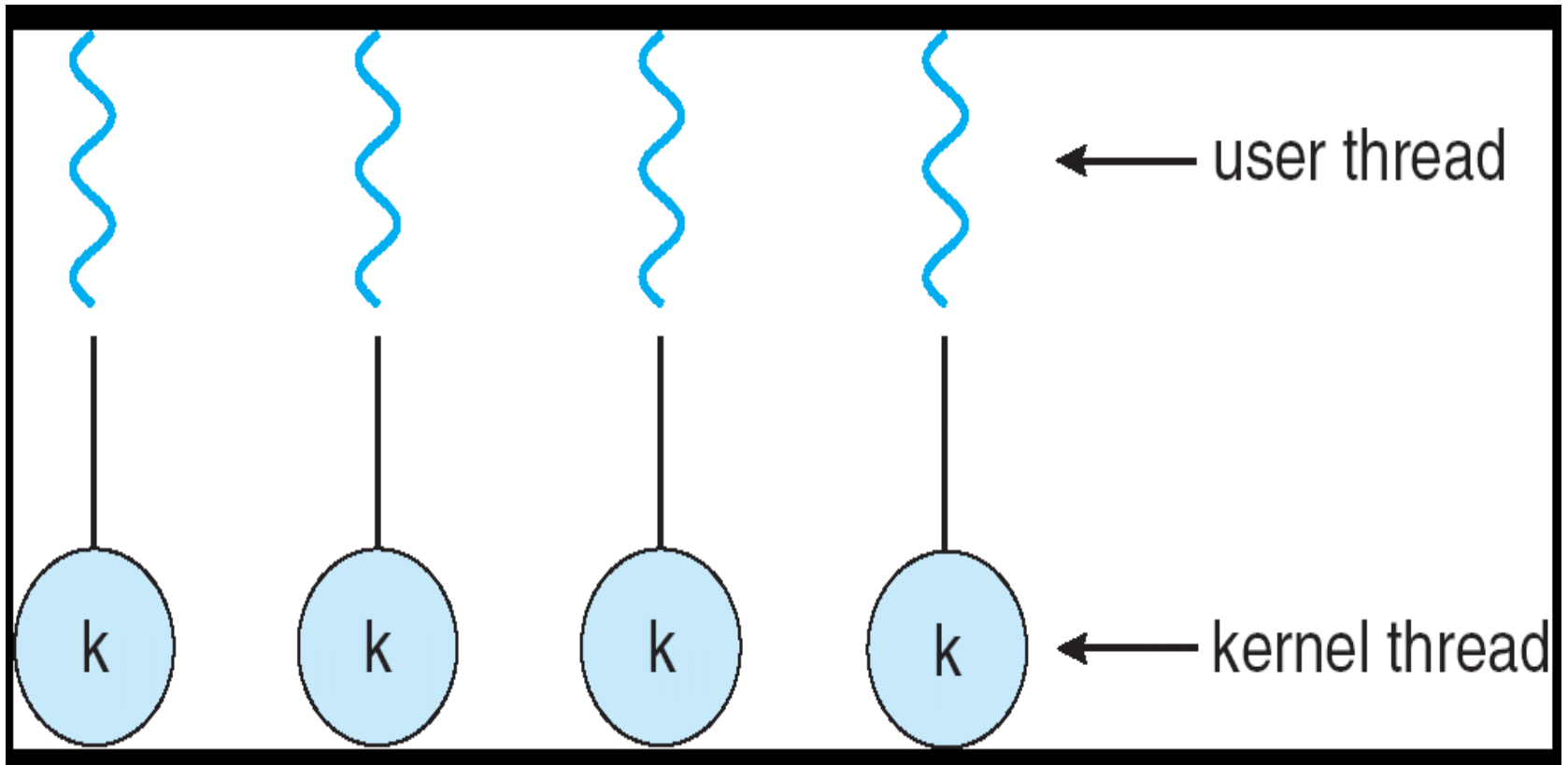
Process VS Thread

Process	Thread
	
<ul style="list-style-type: none"> • Heavy weight 	<ul style="list-style-type: none"> • Light weight
<ul style="list-style-type: none"> • Unit of Allocation <ul style="list-style-type: none"> – Resources, privileges etc 	<ul style="list-style-type: none"> • Unit of Execution <ul style="list-style-type: none"> – PC, SP, registers <p>PC—Program counter, SP—Stack pointer</p>

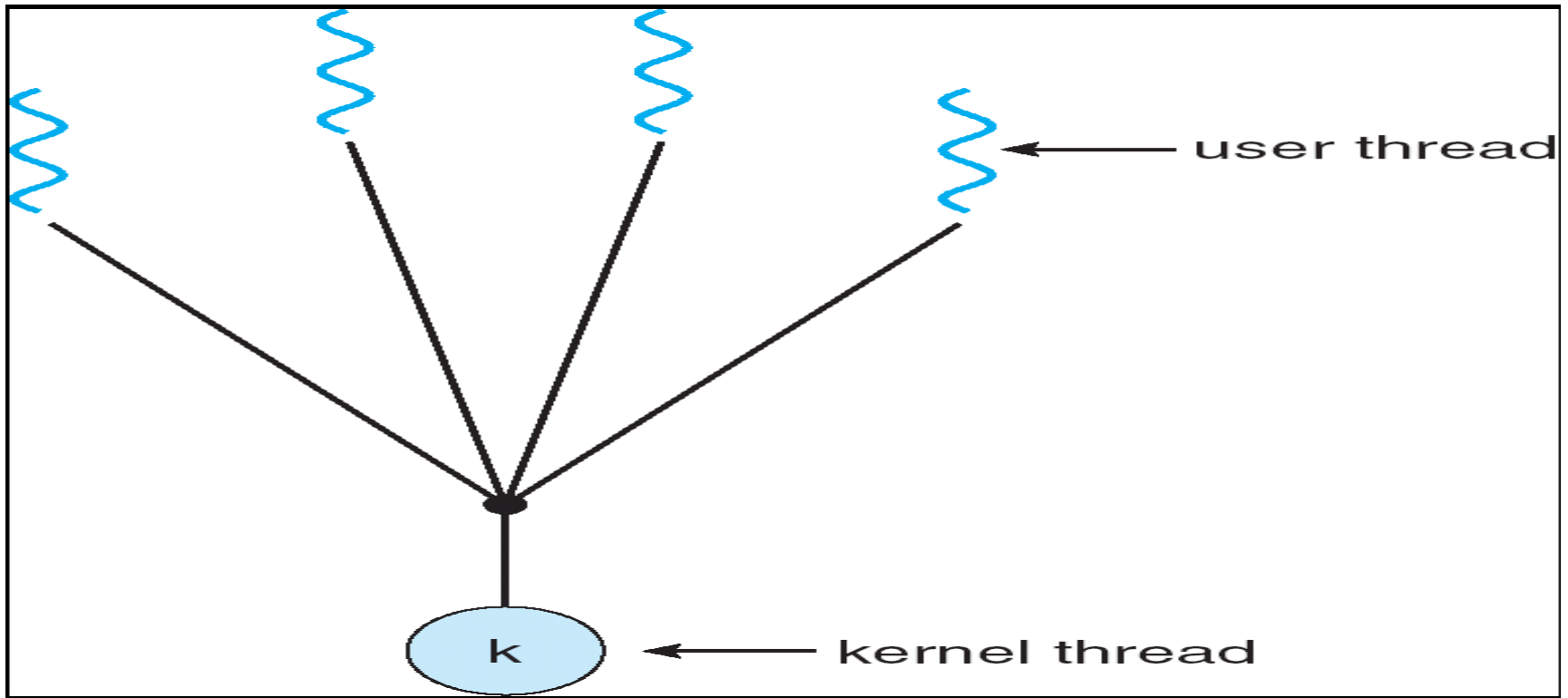
Process VS Thread

<i>Process</i>	<i>Thread</i>
<ul style="list-style-type: none"> • Inter-process communication is expensive: need to context switch • Secure: one process cannot corrupt another process 	<ul style="list-style-type: none"> • 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
<ul style="list-style-type: none"> • Process are Typically independent 	<ul style="list-style-type: none"> • Thread exist as subsets of a process
<ul style="list-style-type: none"> • Process carry considerable state information 	<ul style="list-style-type: none"> • Multiple thread within a process share state as well as memory and other resources
<ul style="list-style-type: none"> • Processes have separate address space 	<ul style="list-style-type: none"> • Thread share their address space
<ul style="list-style-type: none"> • processes interact only through system-provided inter-process communication mechanisms. 	<ul style="list-style-type: none"> • Context switching between threads in the same process is typically faster than context switching between processes

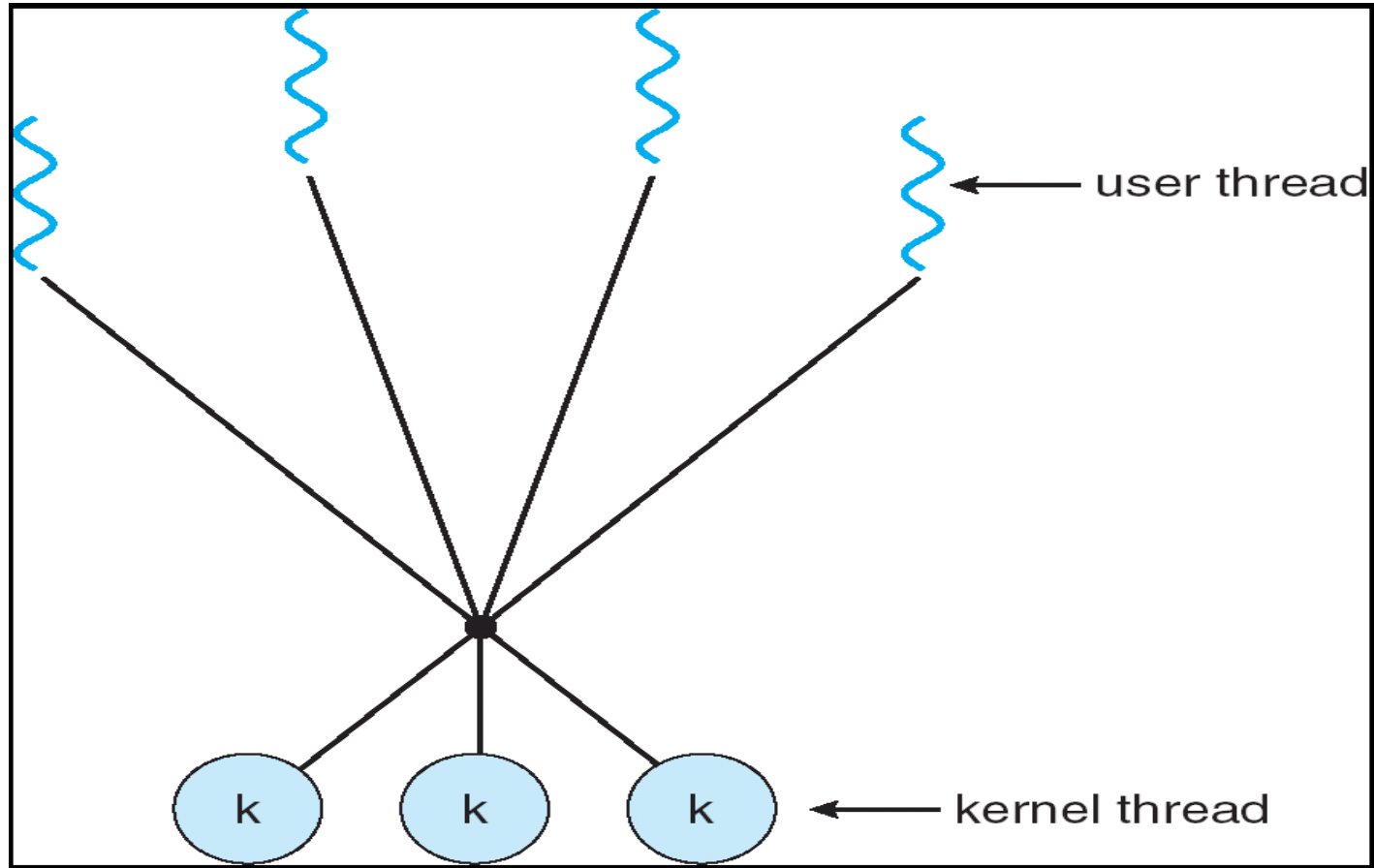
Threads Model



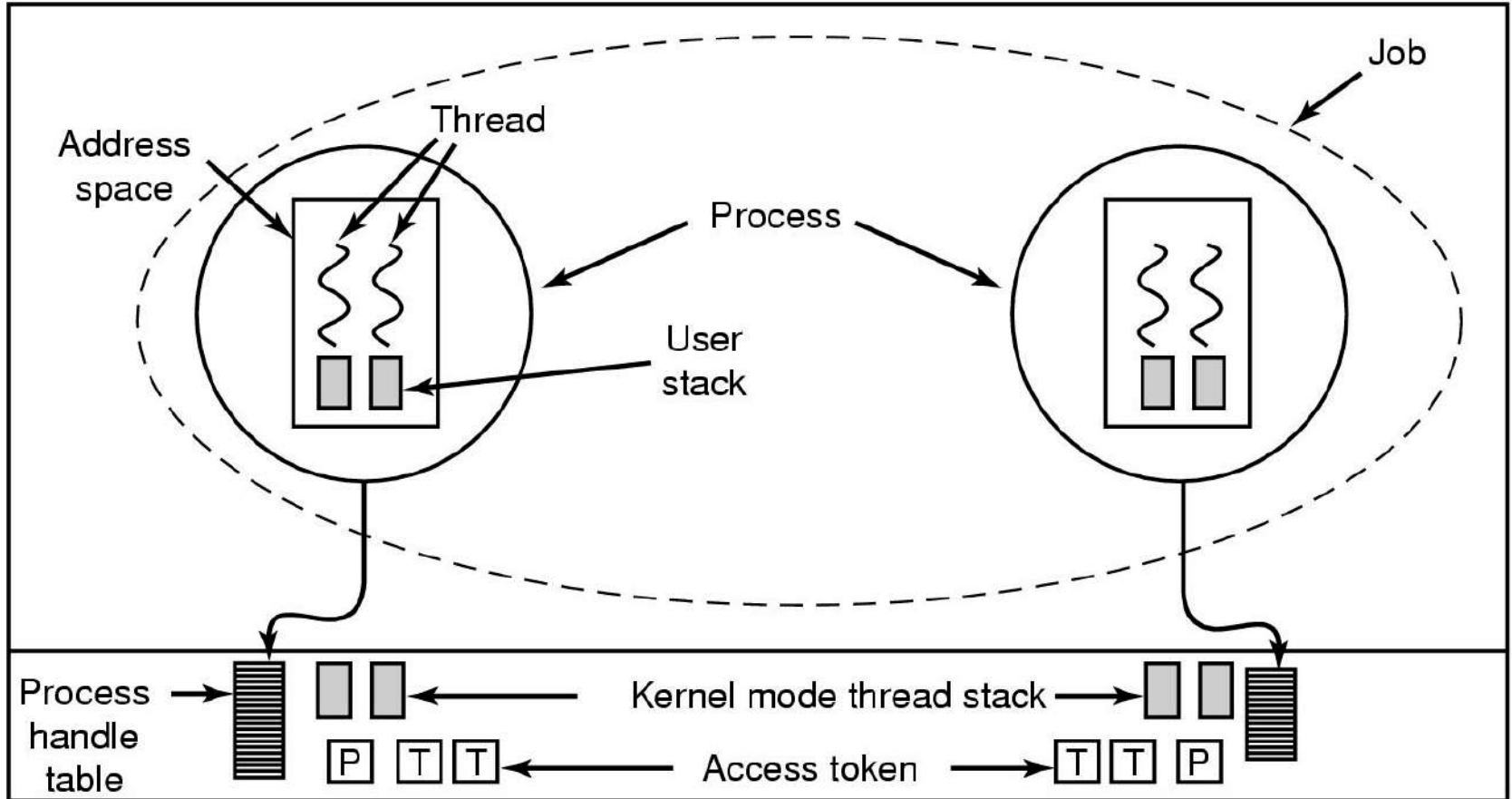
Many to one



Many to Many



Processes and Threads



Relationship between jobs, processes, threads, and fibers

Inter process Communication

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 affect other or be affected by the other process. Any process that shares data with other process is called co-operating process.

- reasons for providing an environment for process co-operation
 - **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.
 - **Computation Speedup:** Breakup tasks into sub-tasks.
 - **Modularity:** construct a system in a modular fashion.
 - **Convenience:**

co-operating Process

co-operating process requires IPC. There are two fundamental ways of IPC.

- **Shared Memory**
- **Message Passing**

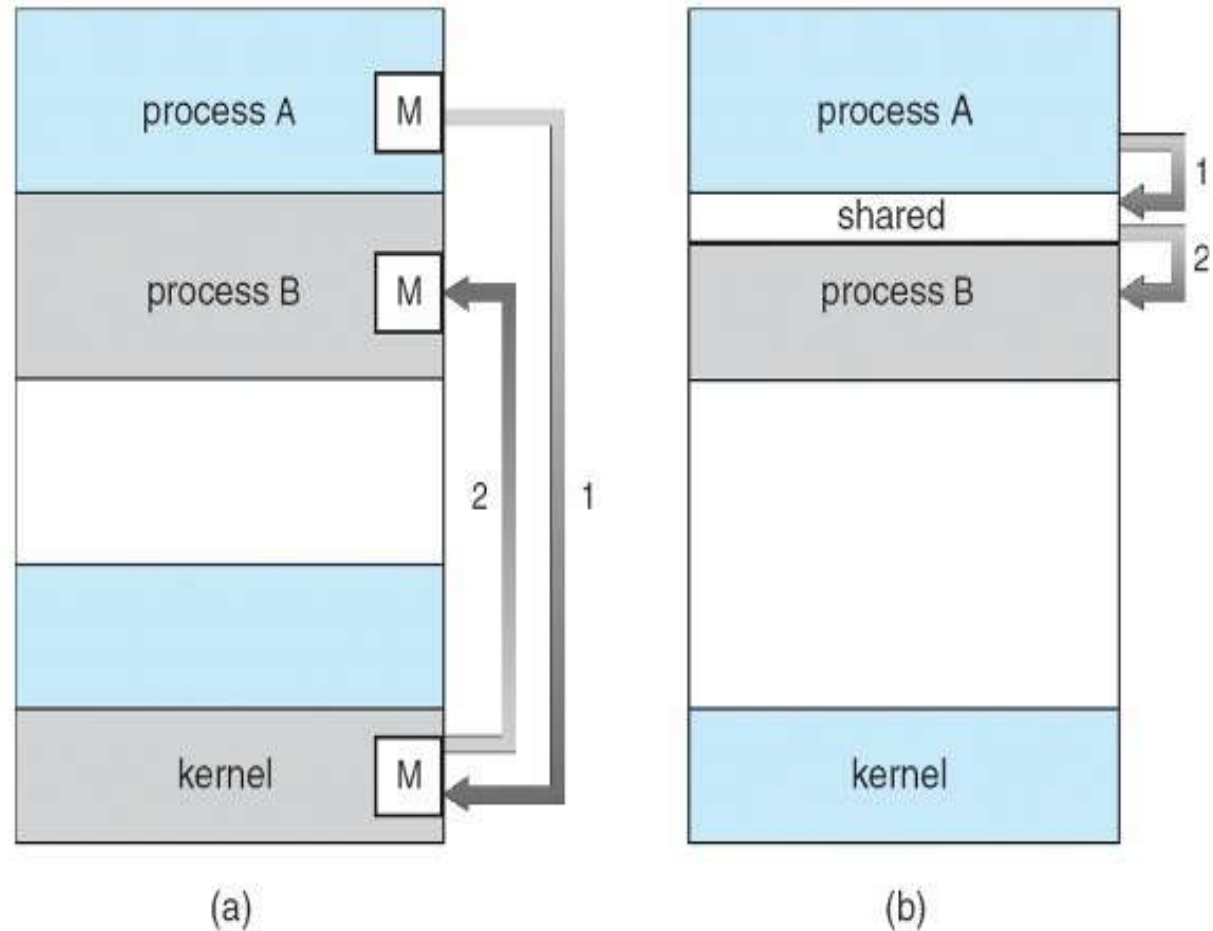


Fig: Communication Model. a) message passing b. shared memory

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

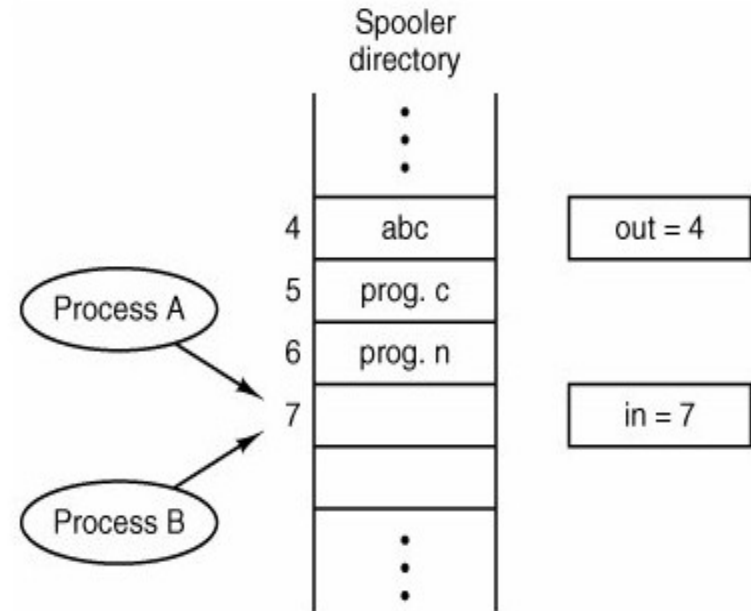
- communication takes place by means of messages exchanged between the co-operating process
- Message passing is useful for exchanging the smaller amount of data since no conflict need to be avoided.
- Easier to implement than shared memory.
- Slower than that of Shared memory as message passing system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

Inter process 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)**.

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 inter process communication works in practice, let us consider a simple but common example, a print spooler.

When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and 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.

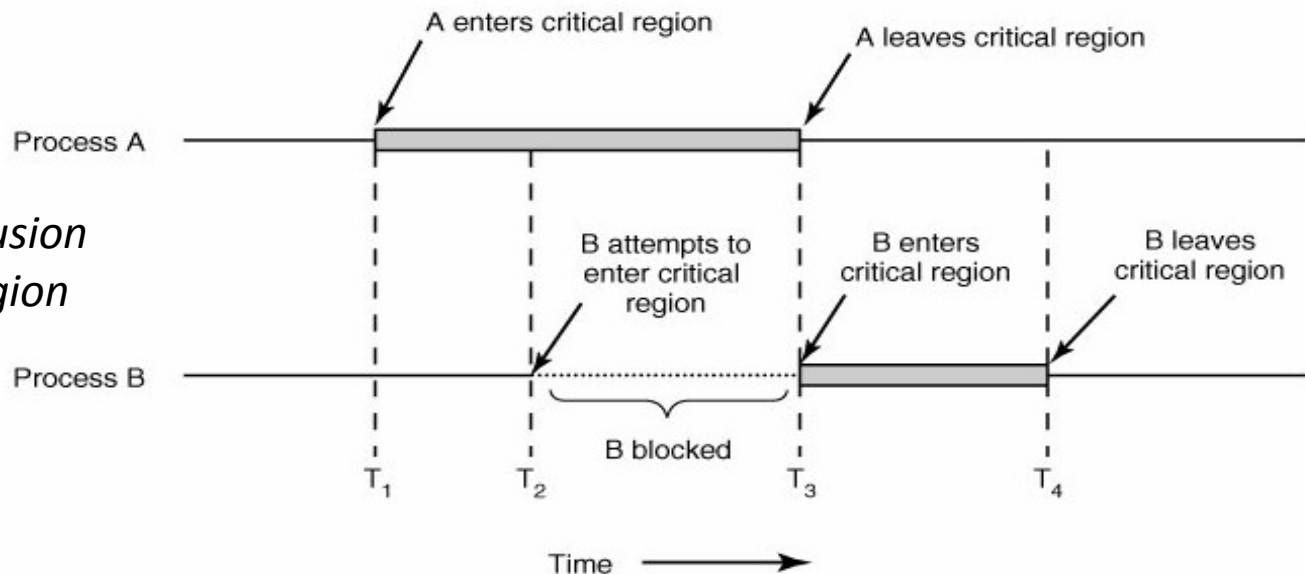
Avoiding Race Conditions

1. Critical Section

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

Solution to Critical section problem:

- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.



*Fig: Mutual Exclusion
using Critical Region*

Mutual Exclusion with Busy Waiting

- We will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble
- **Disabling Interrupts**
- **Lock Variables**
- **Strict Alteration**
- **Peterson's Solution**
- **The TSL Instruction**

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:***
 - 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?
 - 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.

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.

Strict Alteration

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

(a)

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

(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.

Strict Alteration

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

(a)

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

(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.

- **Drawbacks:**

- Taking turn 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.

Peterson's Solution

- 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.

```
#define FALSE 0
#define TRUE 1
#define N 2
/* number of processes */
/* whose turn is it? */
int turn;
/* all values initially 0 (FALSE) */
/* process is 0 or 1 */
int interested[N];
void enter_region(int process)
{
    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 */
}
```

- 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.

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 indivisible no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

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

- 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.

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.

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
 - 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.
 - 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.

Producer-consumer problem cntd..

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.

```
#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.

Producer-consumer problem cntd..

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.
8. 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

```
#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.

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. A semaphore generally takes one of two forms: **binary and counting**. A binary semaphore is a simple "true/false" (locked/unlocked) flag that controls access to a single resource. A counting semaphore is a counter for a set of available resources. Either semaphore type may be employed to prevent a race condition.
- 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.)

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```

Fig: The producer-consumer problem using semaphore

Semaphore cntd..

- This solution uses three semaphore.
 - Full: For counting the number of slots that are full, initially 0
 - Empty: For counting the number of slots that are empty, initially equal to the no. of slots in the buffer.
 - 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.

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

- 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.

Monitors

monitor *example*

integer *i*;

condition *c*;

procedure *producer*();

.

.

.

end;

procedure *consumer*();

.

.

.

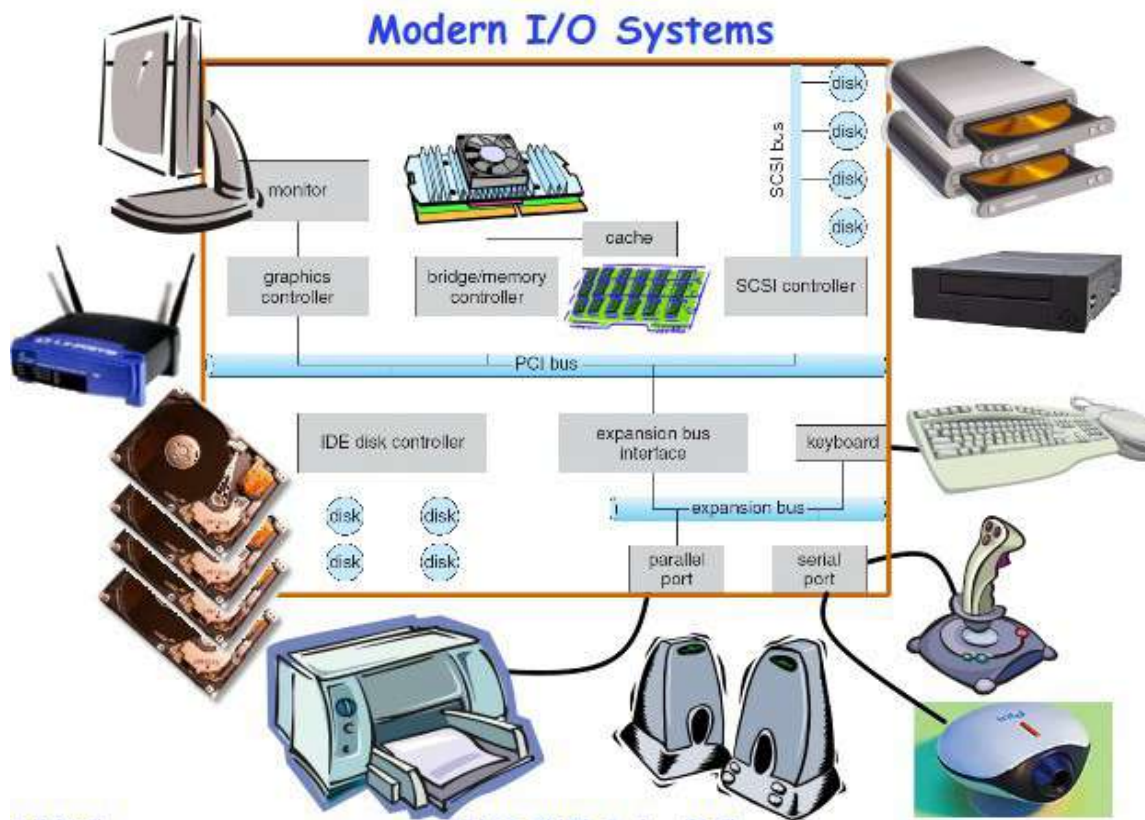
end;

end 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.

Chapter:3 Input/Output



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 through input/output bus. A device controller usually can control several input/output devices.

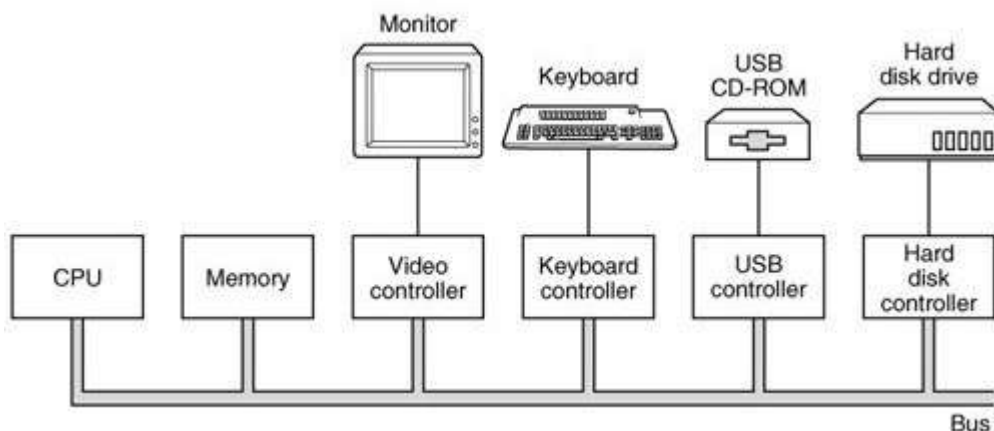


Fig: A model for connecting the CPU, memory, controllers, and I/O devices

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.

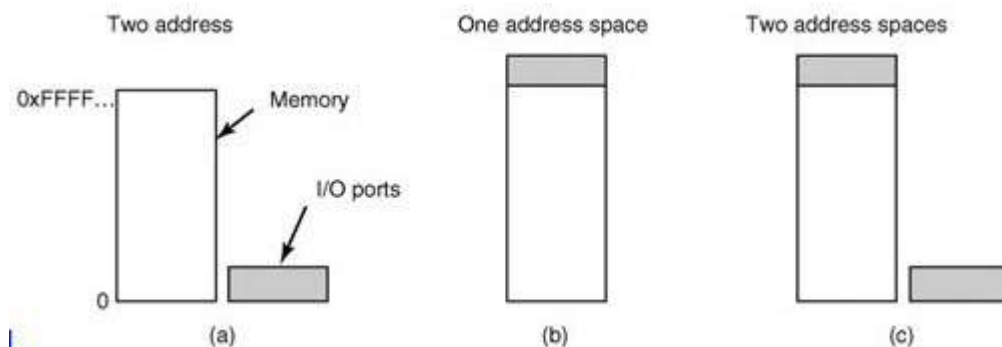


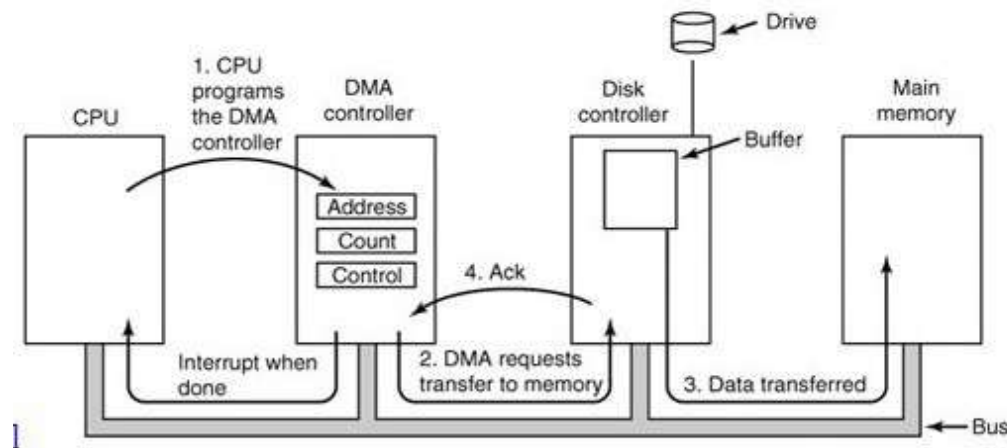
Fig:(a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

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.

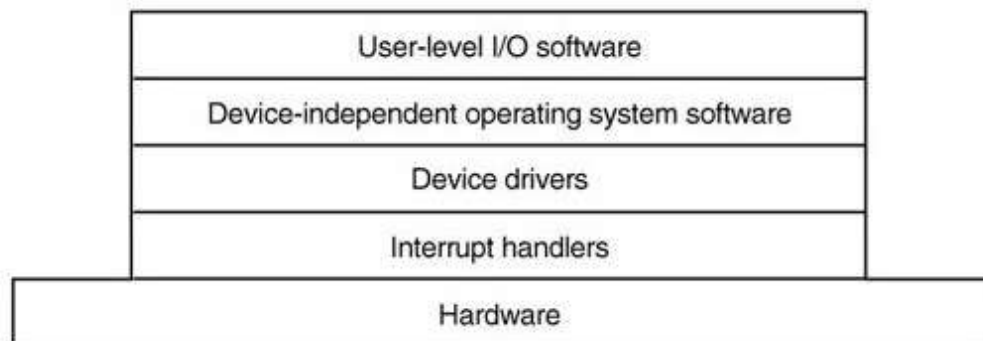
Direct Memory Access (DMA) is a method of allowing data to be moved from one location to another in a computer without intervention from the central processor (CPU).



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:



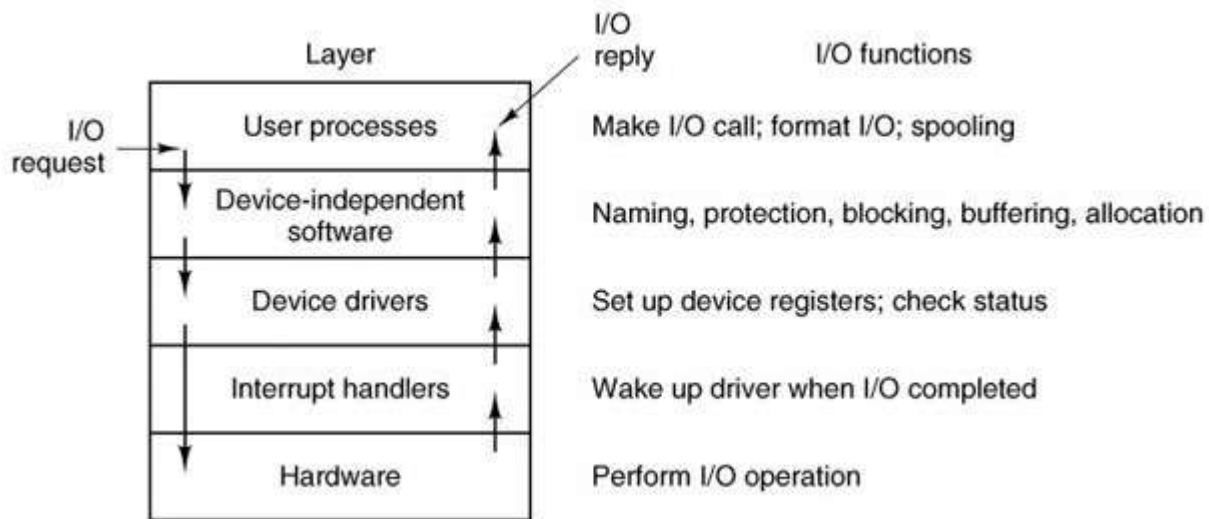


Fig. Layers of the I/O system and the main functions of each layer.

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.

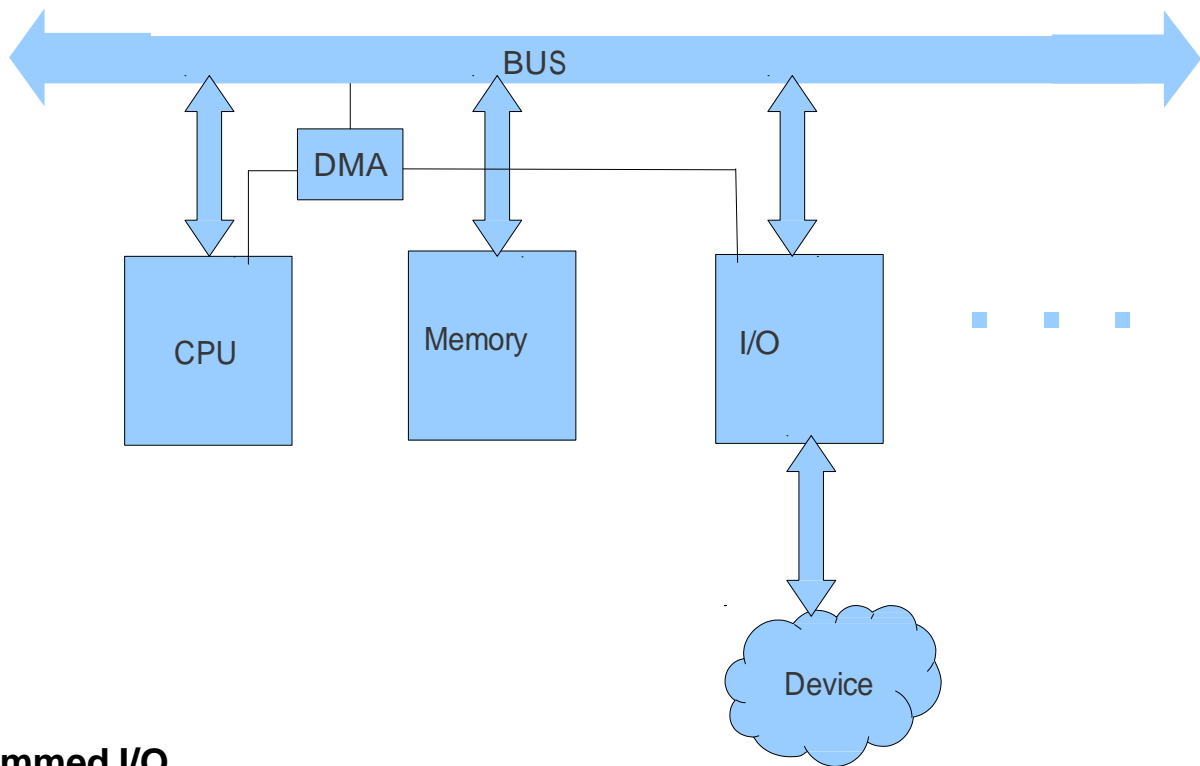
Each device driver normally handles one device type, or one class of closely related devices. For example, it would probably be a good idea to have a single mouse driver, even if the system supports several different brands of mice. As another example, a disk driver can usually handle multiple disks 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.

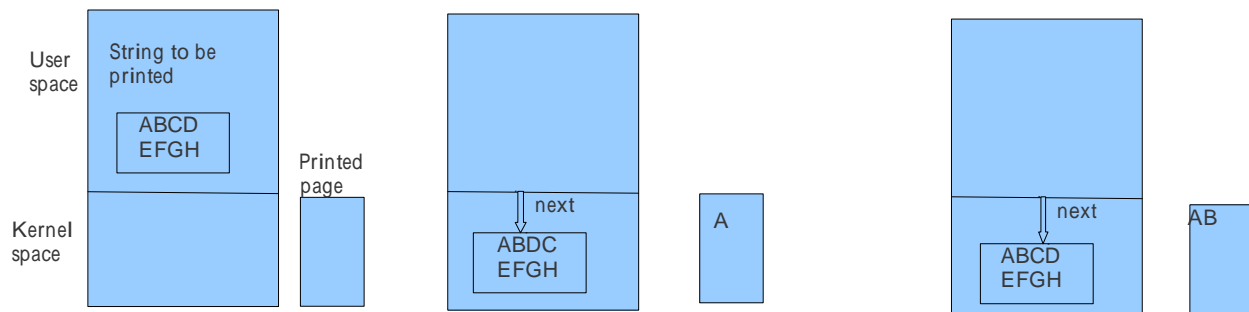


Fig. Steps in Printing a string

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 follows.

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.

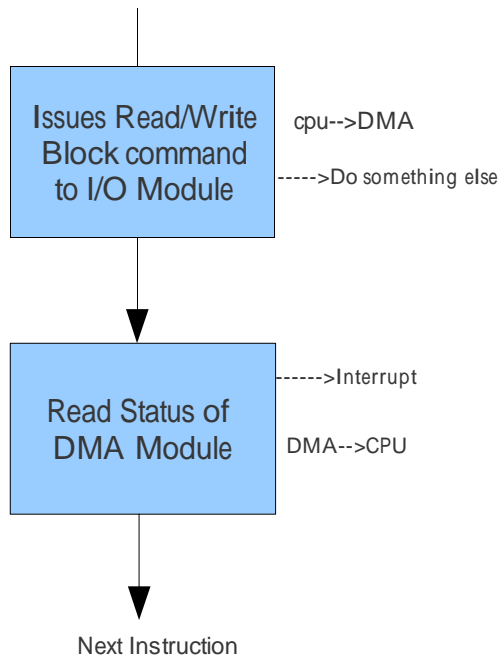


Fig:DMA

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 a 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 must 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.

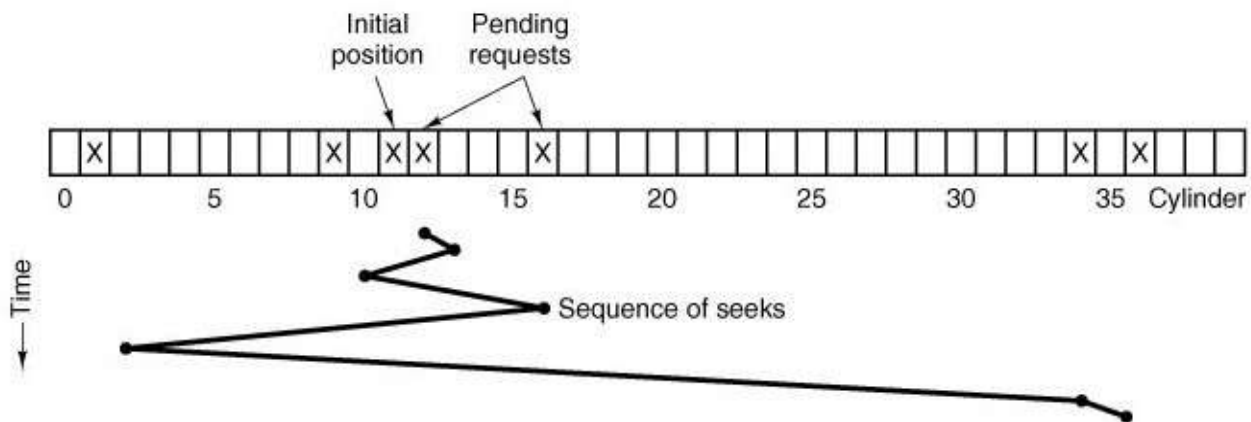
Disk Arm Scheduling Algorithms:

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.

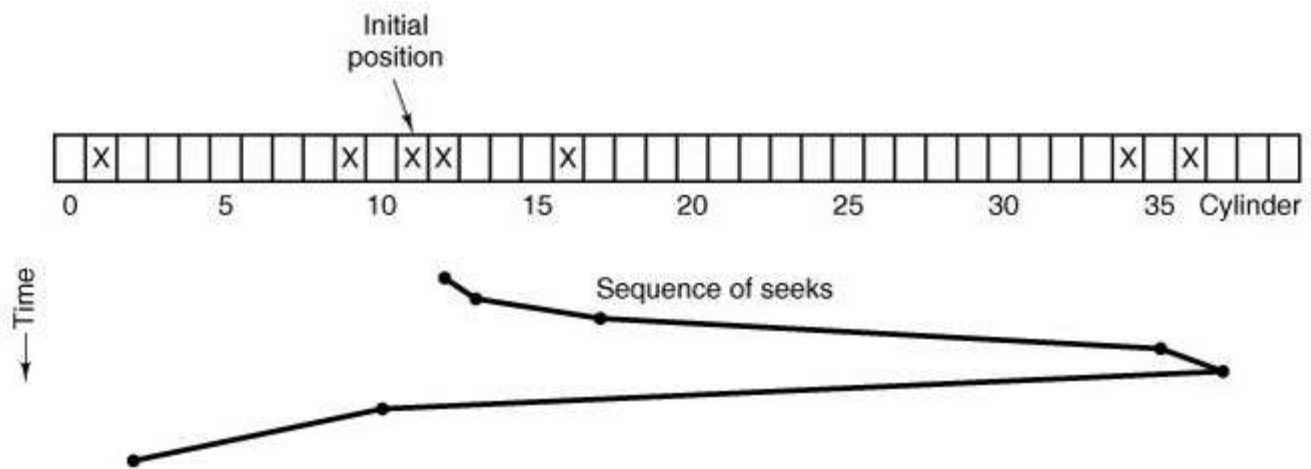


Total no of Cylinder movement: 61

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:

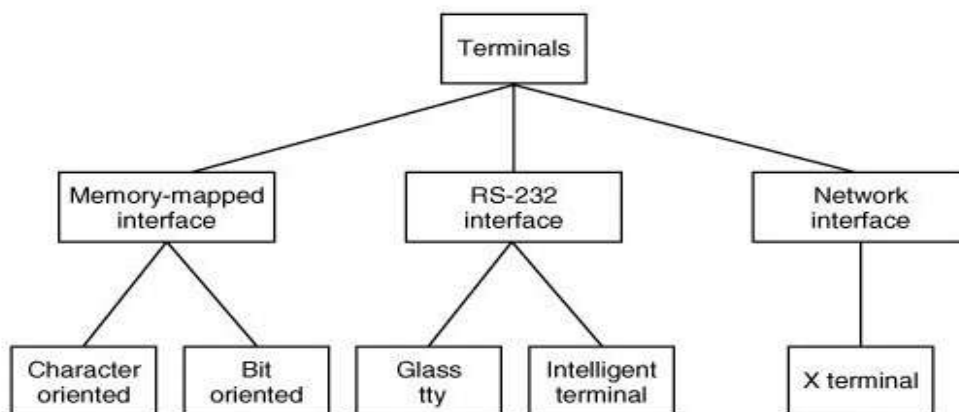
The disk requests come in to the disk driver for cylinders 10, 20, 22, 2, 40, 6 and 30, in the order. A seek takes 6 msec/cylinder moved. How much seek time is needed for:

- FCFS
- Shortest Seek First
- 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 an 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.

Chapter 4:Deadlock:

Resources

– passive entities needed by threads to do their work

CPU time, disk space, memory

Two types of resources:

Preemptable – can take it away

CPU, Embedded security chip, Memory

Non-preemptable – must leave it with the thread

Disk space, printer, chunk of virtual address space, Critical section

Resources may require exclusive access or may be sharable

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

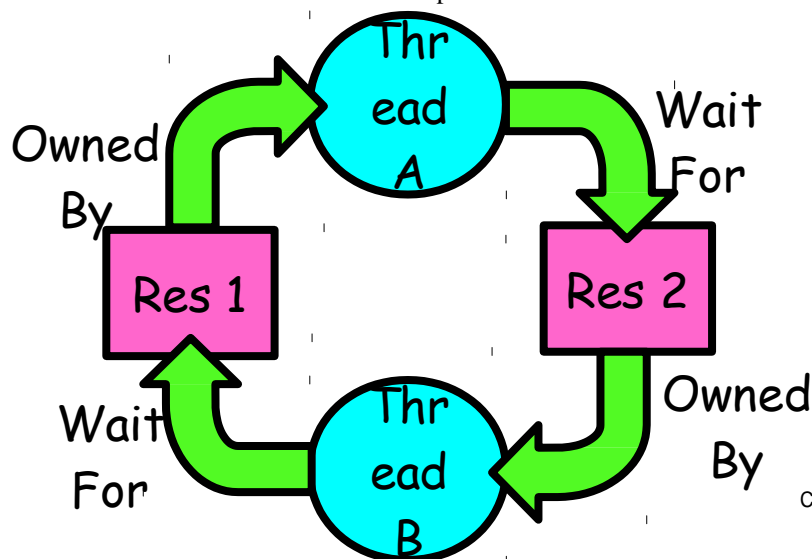
Use the resource

2. 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.



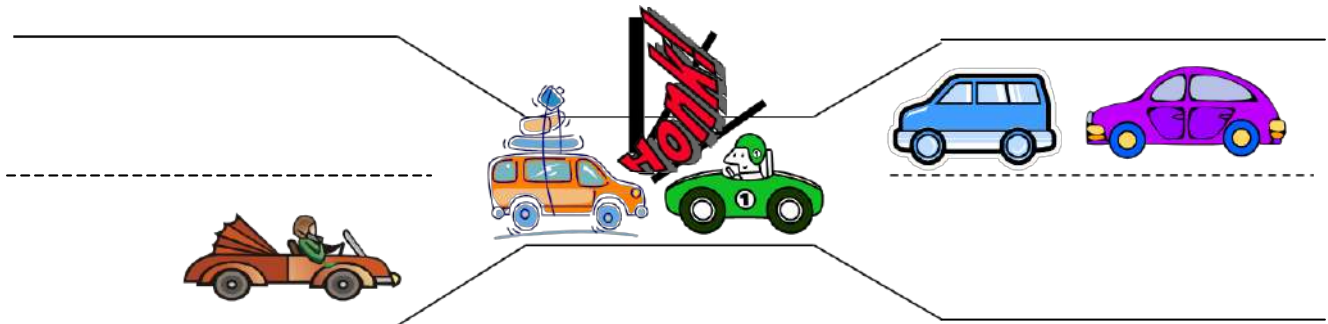
Compiled by: Arjun Aryal

Example 1:

- Two process A and B each want to record 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 asks 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 $\{P_1, \dots, P_n\}$ of waiting processes. P_1 is waiting for a resource that is held by P_2

P_2 is waiting for a resource that is held by P_3

...

P_n is waiting for a resource that is held by P_1

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

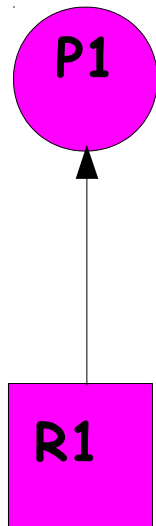
E . V is partitioned into two types:

$P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system.

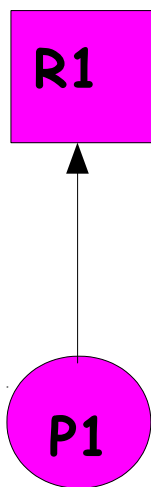
$R = \{R_1, R_2, \dots, R_m\}$, the set consisting of all resource types in the system.

request edge – directed edge $P_i \rightarrow R_j$

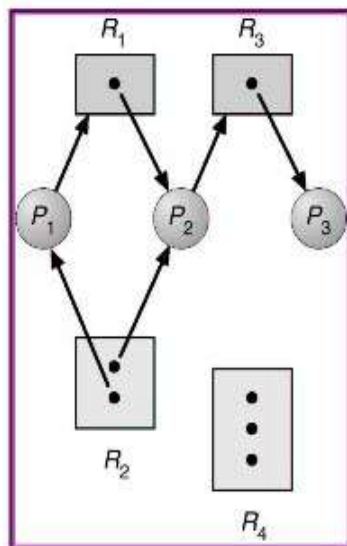
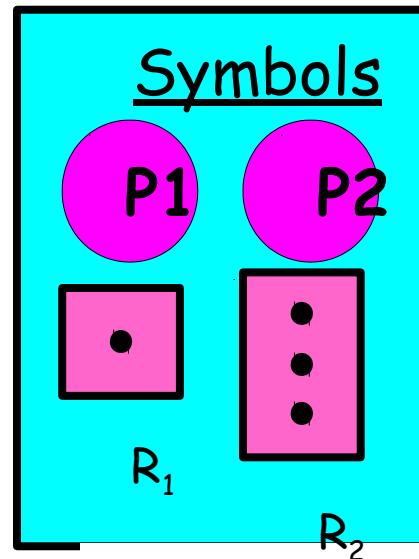
assignment edge – directed edge $R_j \rightarrow P_i$



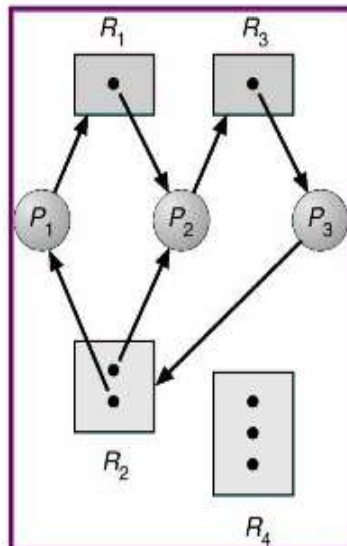
a). P_1 is



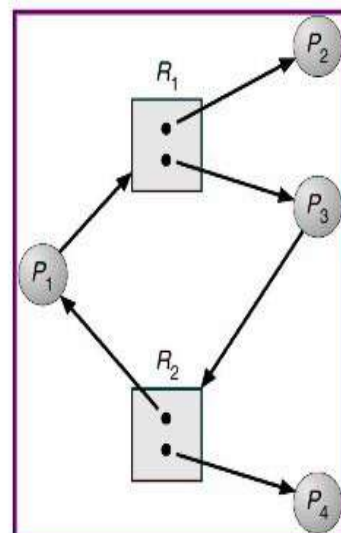
b). P_1



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 \Rightarrow no deadlock.

If graph contains a cycle \Rightarrow

- 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 occupy 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 $r_1 = 1$, $r_2 = 2$, $r_3 = 3$, and $r_4 = 4$. With these priorities, if process P wants to use r_1 and r_3 , it should first request r_1 , then r_3 .

Another protocol is “Whenever a process requests a resource r_j , it must have released all resources r_k with $\text{priority}(r_k) \geq \text{priority}(r_j)$).

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:

Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe for the current allocation state if, for each P_i , the resources which P_i can still request can be satisfied by

- the currently available resources plus
- the resources held by all of the P_j '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 \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
- Avoidance \Rightarrow 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:

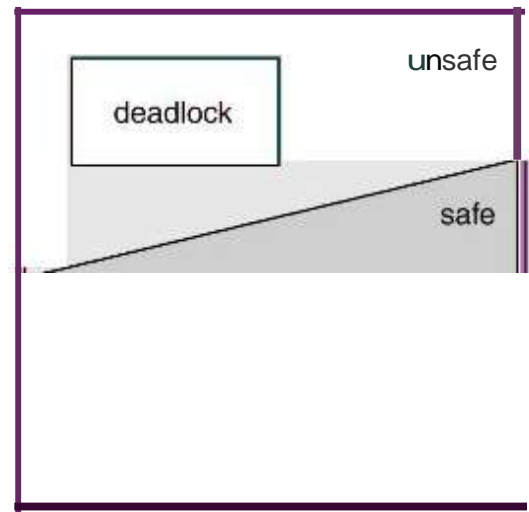
B requests one more unit and is granted.

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.

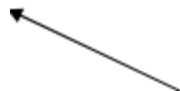
It is important to note that an unsafe state does not imply the existence or even eventual existence of a deadlock. What an unsafe does imply is that some unfortunate sequence of events might lead a deadlock.



	has max		has max		has max		has max		has max		
A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0		B	0	
C	2	7	C	2	7	C	2	7	C	7	7
Free:3			Free: 1			Free: 5			Free:0		

state is safe

	h:m:>x		h:>o	m:>		n:>o	m:>		n:>o	m:>v	
A	3	9	A	4	9	A	4	9	A	3	9
B	2	4	B	2	4	B	4	4	B	0	
C	2	7	C	2	7	C	2	7	C	2	7
Free:3			Free: 2			Free:0			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					Resources still needed				
A	3	0	1	1	A	1	1	0	0
B	0	1	0	0	B	0	1	1	2
C	1	1	1	0	C	3	1	0	0
D	1	1	0	1	D	0	0	1	0
E	0	0	0	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.

$$\Rightarrow A = (0, 0, 0, 0) + (2, 1, 2, 1) = (2, 1, 2, 1)$$

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	R0	R1	R2	R3
P0	0	0	1	0
P1	2	0	0	1
P1	0	1	2	0

Allocation Request Matrix				
Process	R0	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. $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$

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 alog.

For P_i if $\text{Need}_i \leq \text{Available}$, then p_i is in Safe sequence,
 $\text{Available} = \text{Available} + \text{Allocation}_i$

For P0, $\text{need}_0 = 7, 4, 3$

$\text{Available} = 3, 3, 2$

\Rightarrow Condition is false, So P0 must wait.

For P1, $\text{need}_1 = 1, 2, 2$

$\text{Available} = 3, 3, 2$

$\text{need}_1 < \text{Available}$

So P1 will be kept in safe sequence. & Available will be updated as:

$\text{Available} = 3, 3, 2 + 2, 0, 0 = 5, 3, 2$

For P2, $\text{need}_2 = 6, 0, 0$

$\text{Available} = 5, 3, 2$

\Rightarrow condition is again false, so P2 also have to wait.

For P3, $\text{need}_3 = 0, 1, 1$

$\text{Available} = 5, 3, 2$

\Rightarrow condition is true, P3 will be in safe sequence.

$\text{Available} = 5, 3, 2 + 2, 1, 1 = 7, 4, 3$

For P4, $\text{need}_4 = 4, 3, 1$

$\text{Available} = 7, 4, 3$

==> condition $\text{Need}_i \leq \text{Available}$ is true, so P4 will be in safe sequence
 $\text{Available} = 7, 4, 3 + 0, 0, 2 = 7, 4, 5$

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

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

Next P0 whose need = 7, 4, 3
 $\text{Available} = 10, 4, 7$
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 5: Memory Management

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.

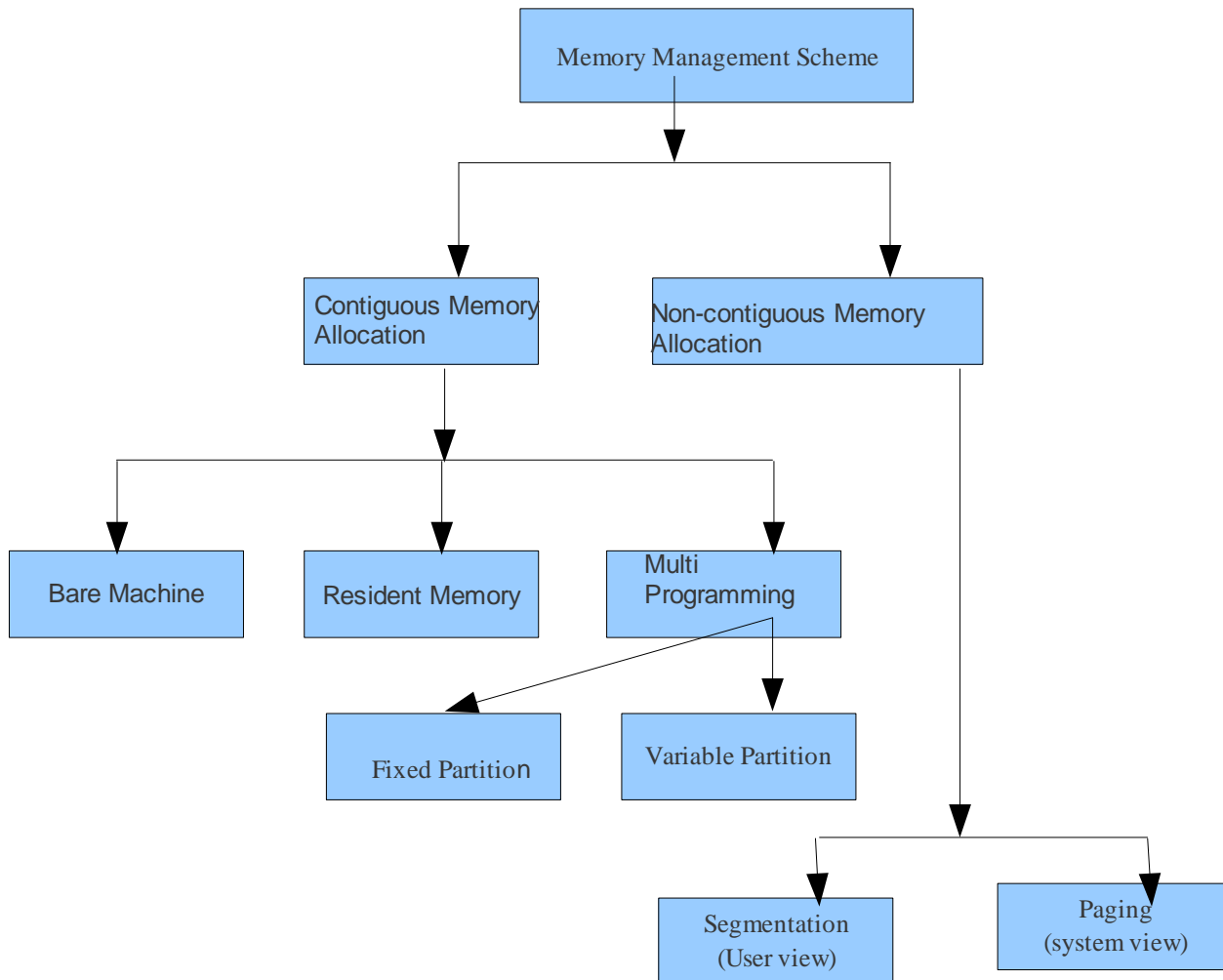


Fig: Types of Memory management

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 manage 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

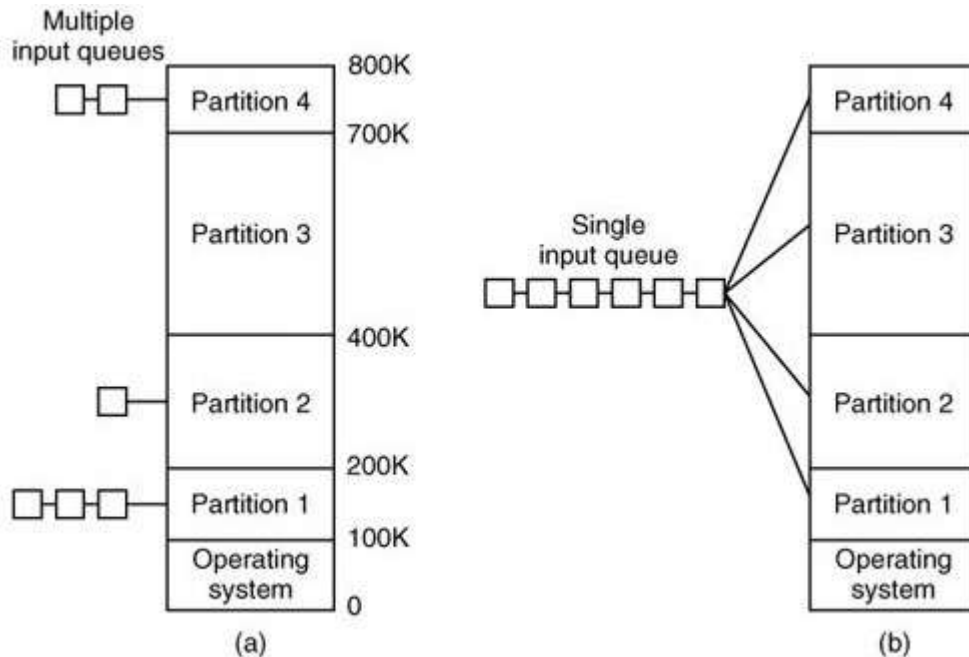
Disadvantages:

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.

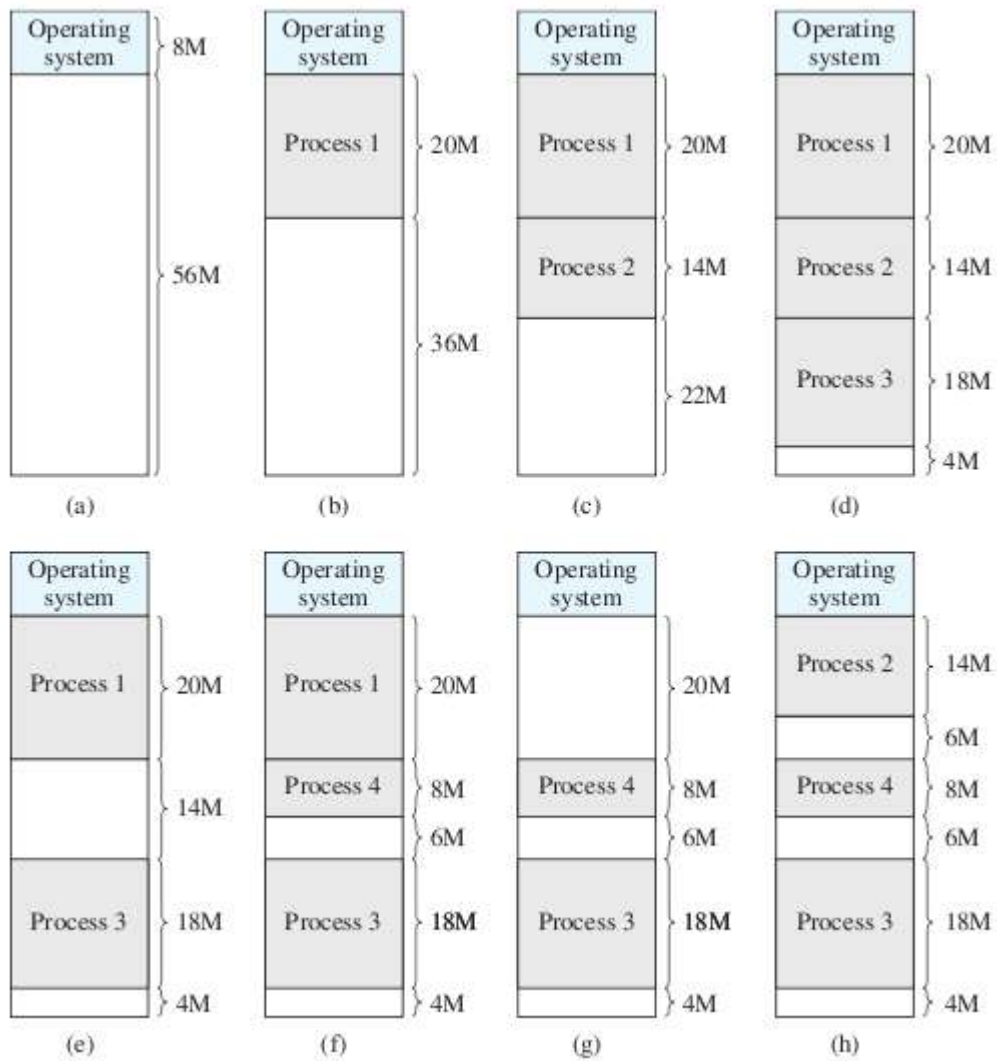


Fig. The Effect of dynamic partitioning

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.

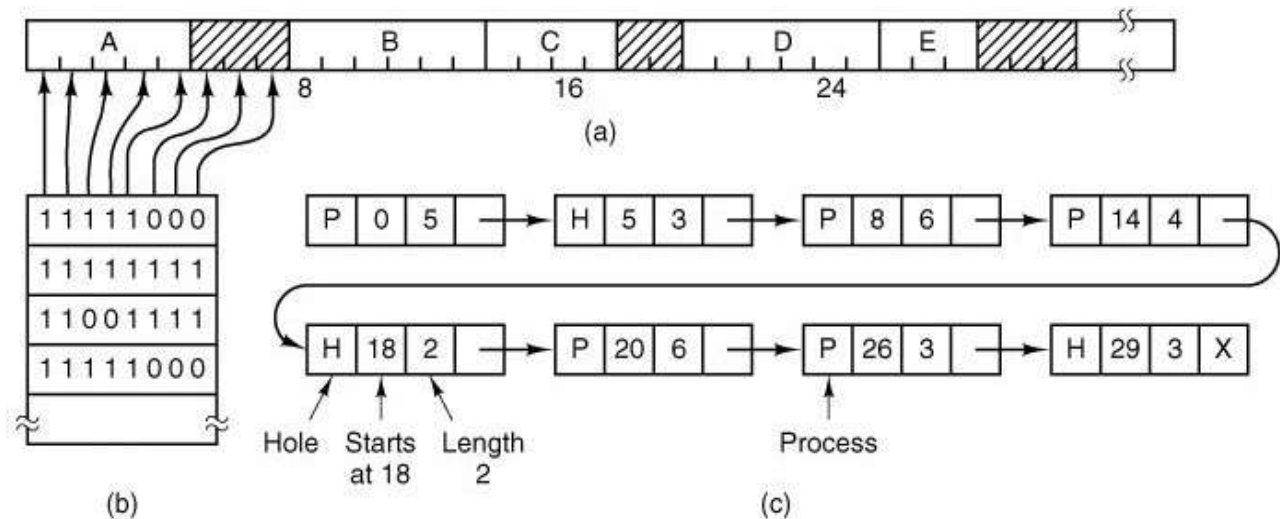


Fig:(a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

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.

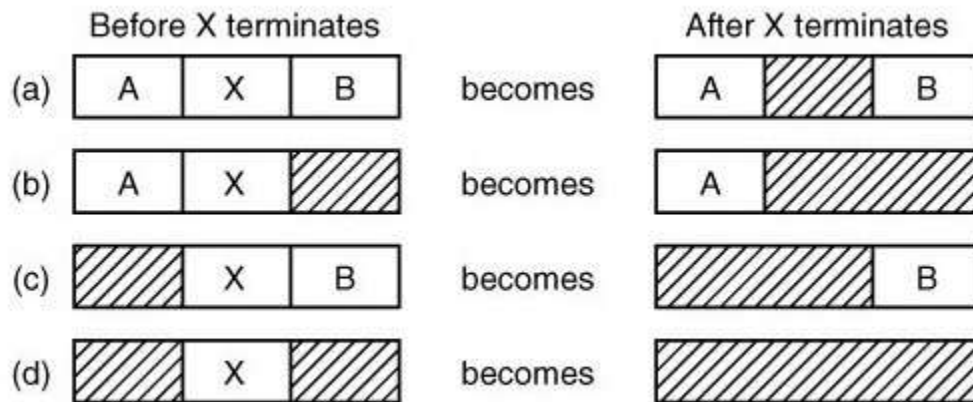


Fig: Four neighbor combinations for the terminating process, X.

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.

	0	128k	256k	512k	1024k
start	1024k				
A=70K	A	128	256	512	
B=35K	A	B 64	256	512	
C=80K	A	B 64	C 128	512	
A ends	128	B 64	C 128	512	
D=60K	128	B D	C 128	512	
B ends	128	64 D	C 128	512	
D ends	256		C 128	512	
C ends	512			512	
end	1024k				

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.

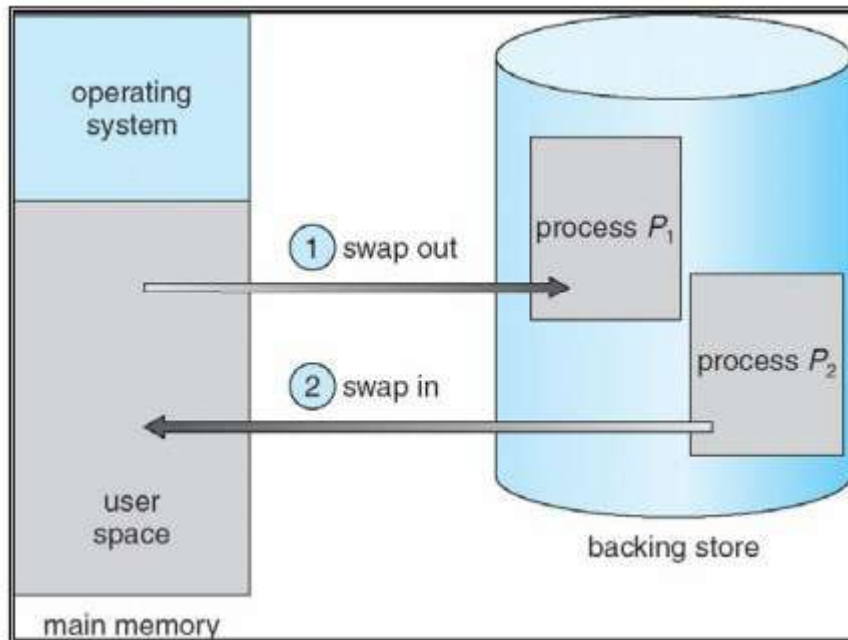


Fig: Swapping of two processes using a disk as a backing store

Logical Address VS Physical Address:

An address generated by the CPU is commonly referred to as a **logical 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 **logical 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 in contiguous memory allocation. We have seen a method called compaction to resolve this problem. Since it's 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 to 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 MEMORY 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 technique 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

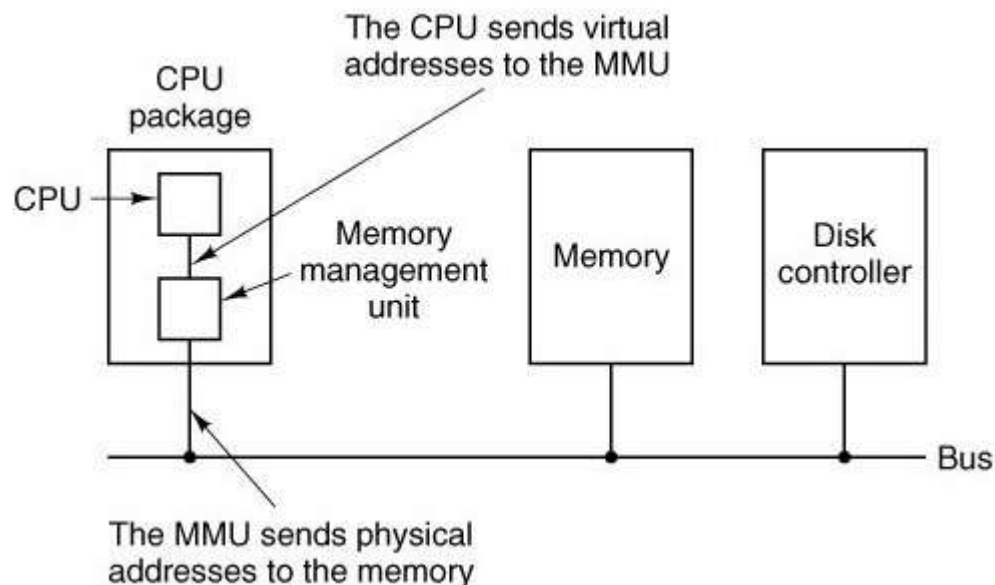


Fig: The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays

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 $12288 + 20 = 12308$.

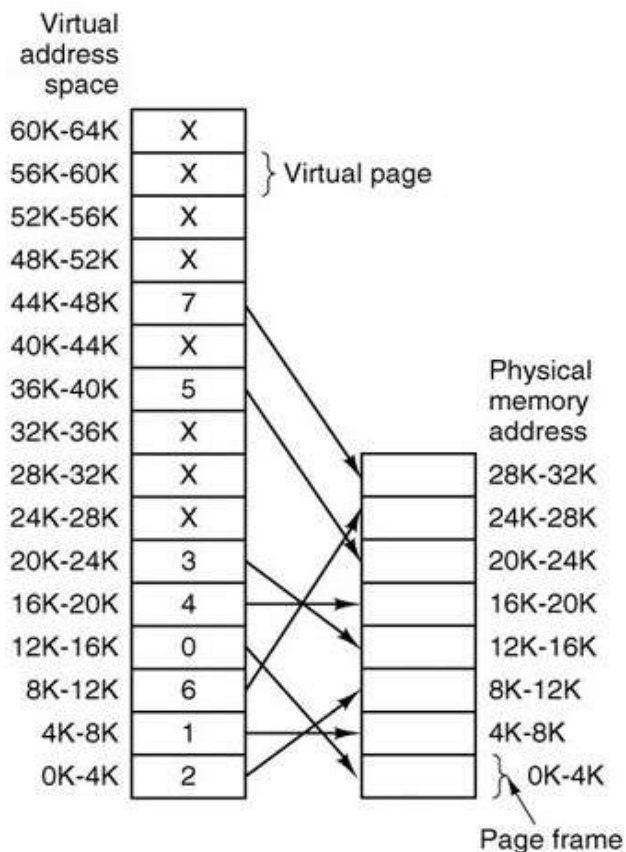


Fig: The relation between virtual addresses and physical memory addresses is given by the page table.

PAGE Fault:

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.

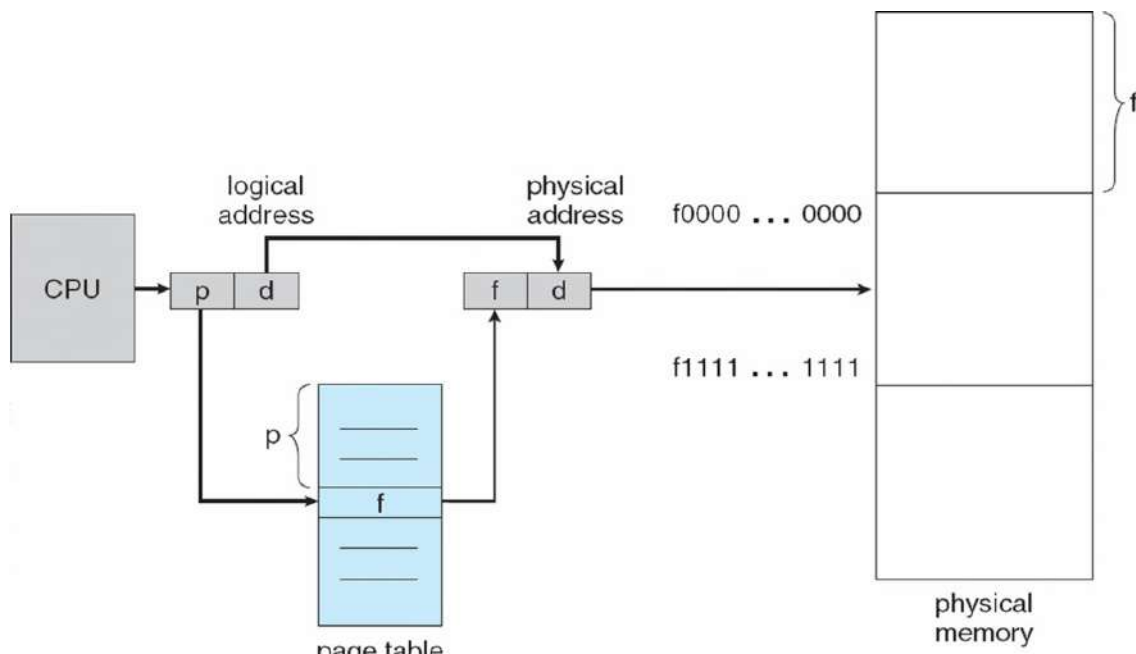


Fig:Paging Hardware

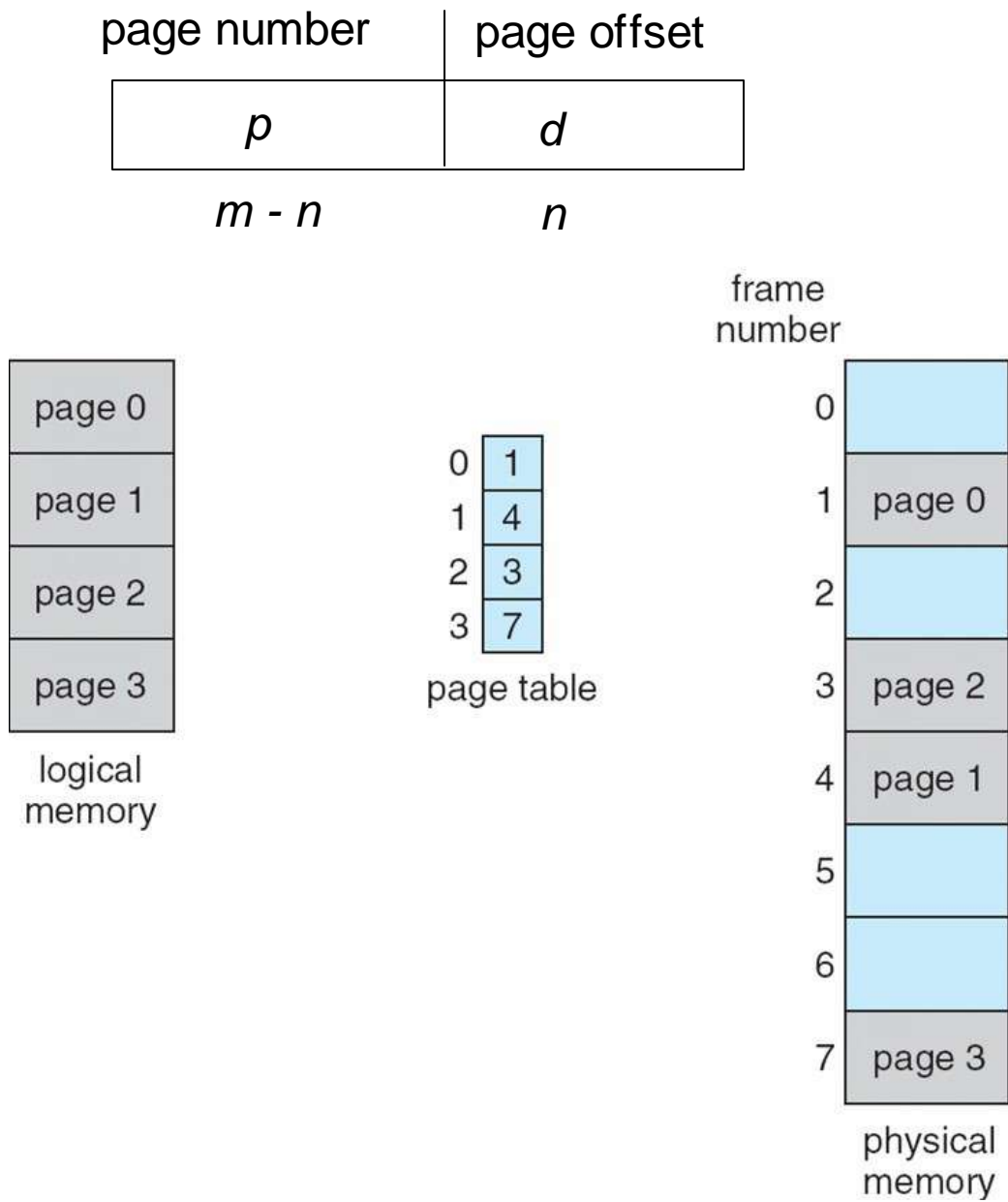


Fig: Paging model of Logical and Physical Memory

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

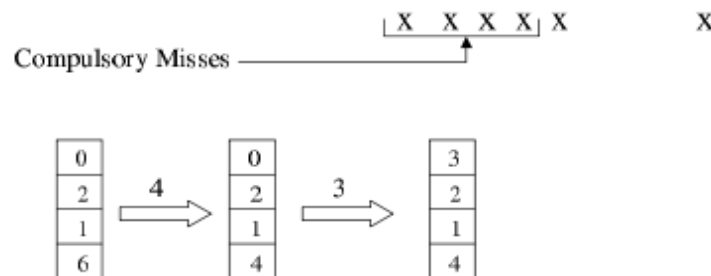
The Optimal Page Replacement Algorithm:

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.

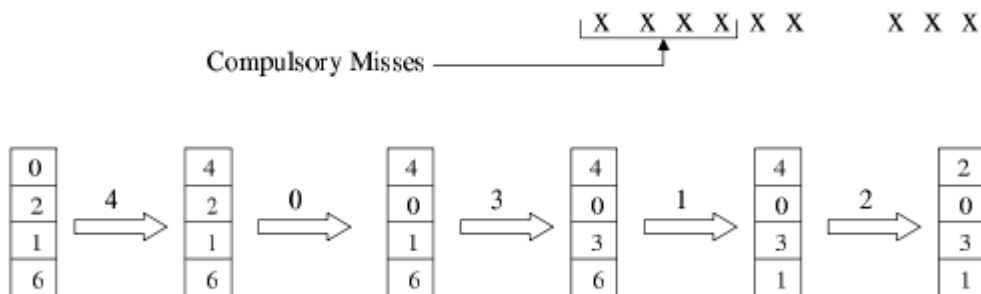
- Very simple to implement.

- 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.

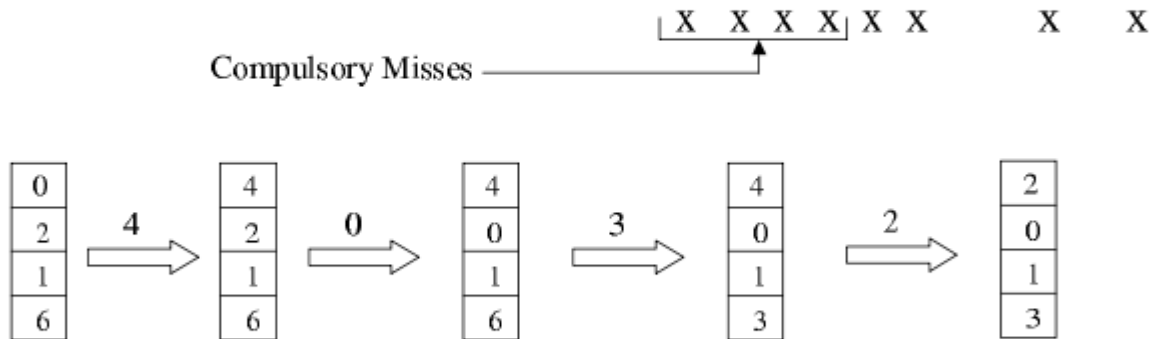
LRU(Least Recently Used):

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



- 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.

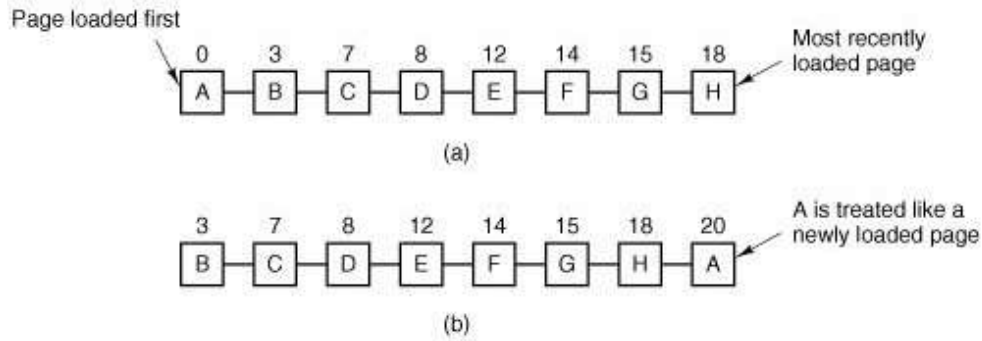
Class 2: referenced, not modified.

Class 3: referenced, modified.

The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class.

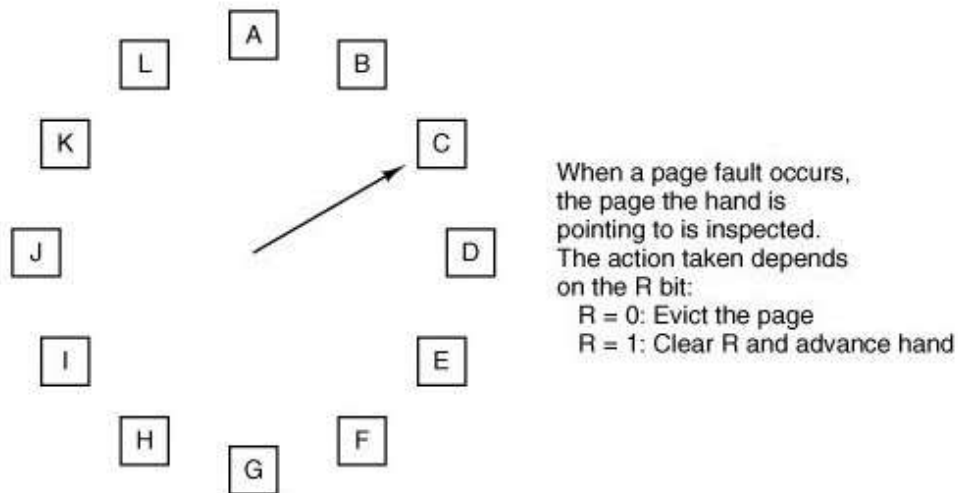
The Second Chance Page Replacement Algorithm:

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.



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).

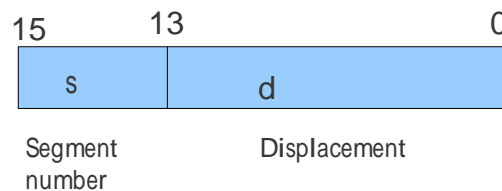
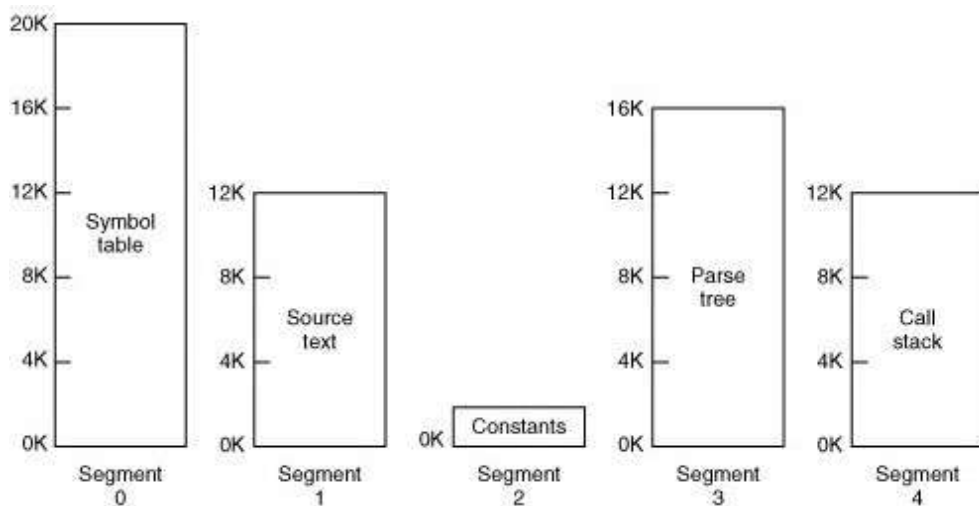
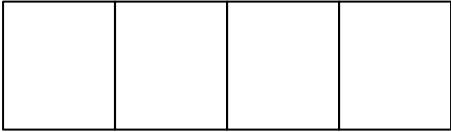
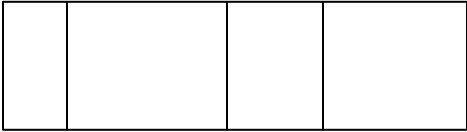


Fig:Virtual address space or logical address space (16 bit)



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

Paging Vs Segmentation:

Sno.	Paging	Segmentation
1	<p>Block replacement easy Fixed-length blocks</p> 	<p>Block replacement hard Variable-length blocks Need to find contiguous, variable-sized, unused part of main memory</p> 
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 data 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 is 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.

Chapter:6 File-systems

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

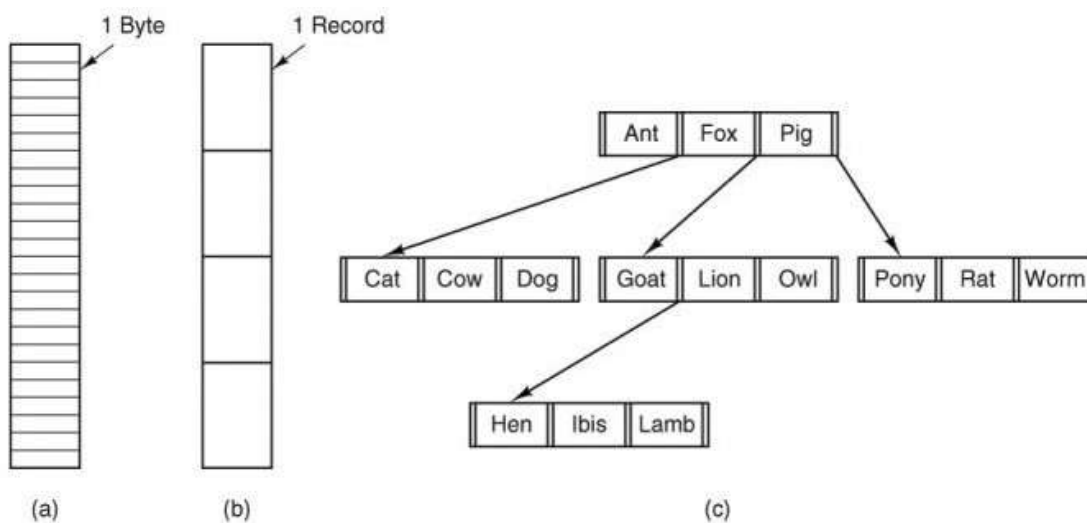
File Operations

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 kept 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 stores 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, **it is simple to implement** because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.
2. 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.

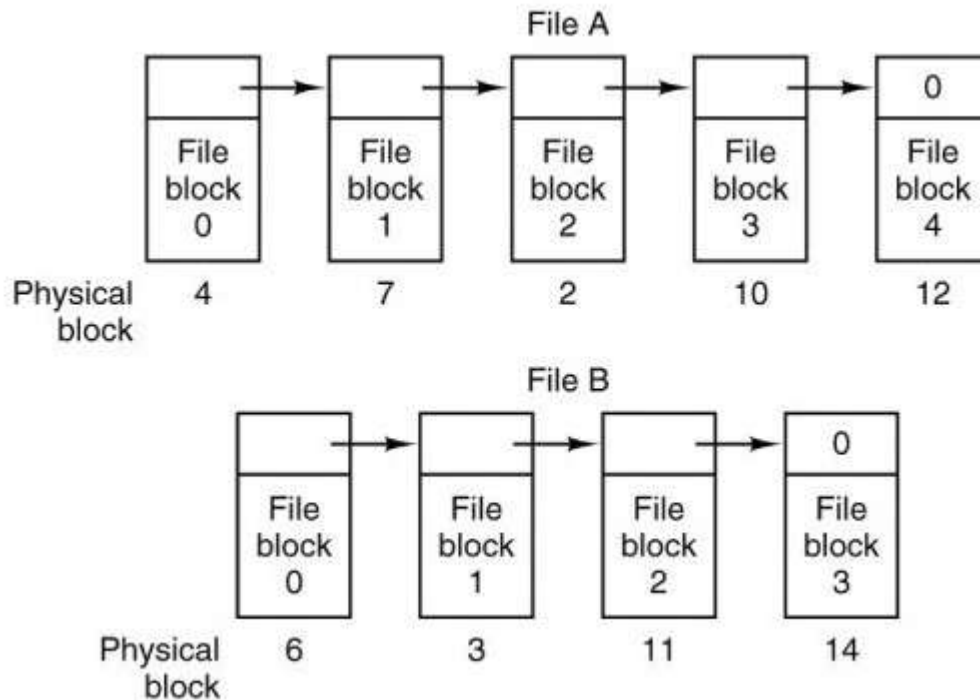


Fig: Storing a file as a linked list of disk blocks.

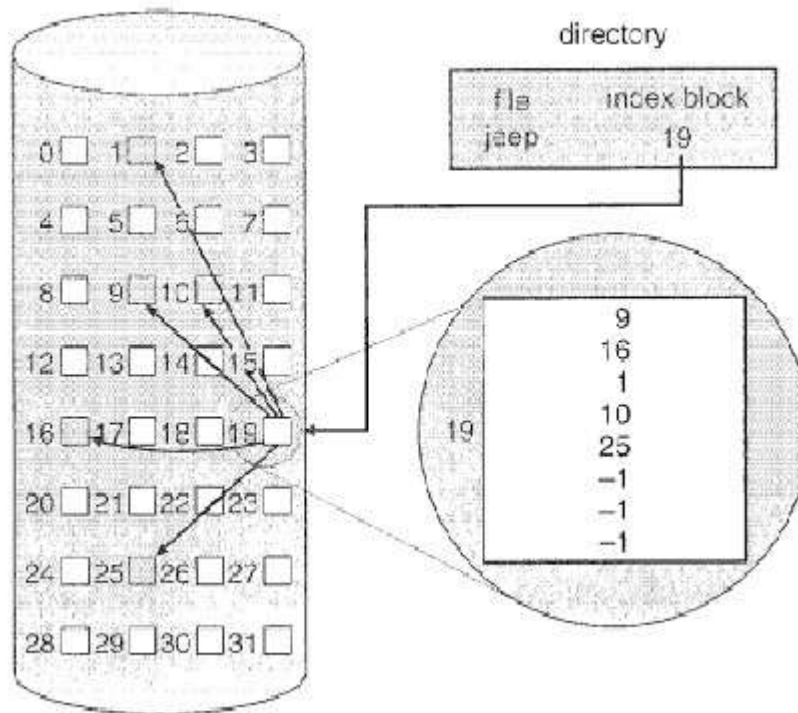
Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation. The major problem with linked allocation is that it can be used only for sequential access files. To find the i^{th} block of a file, we must start at the beginning of that file, and follow 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 i th entry in the index block points to the i th block of the file. The directory contains the address of the index block.



Index allocation of Disk sapce

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

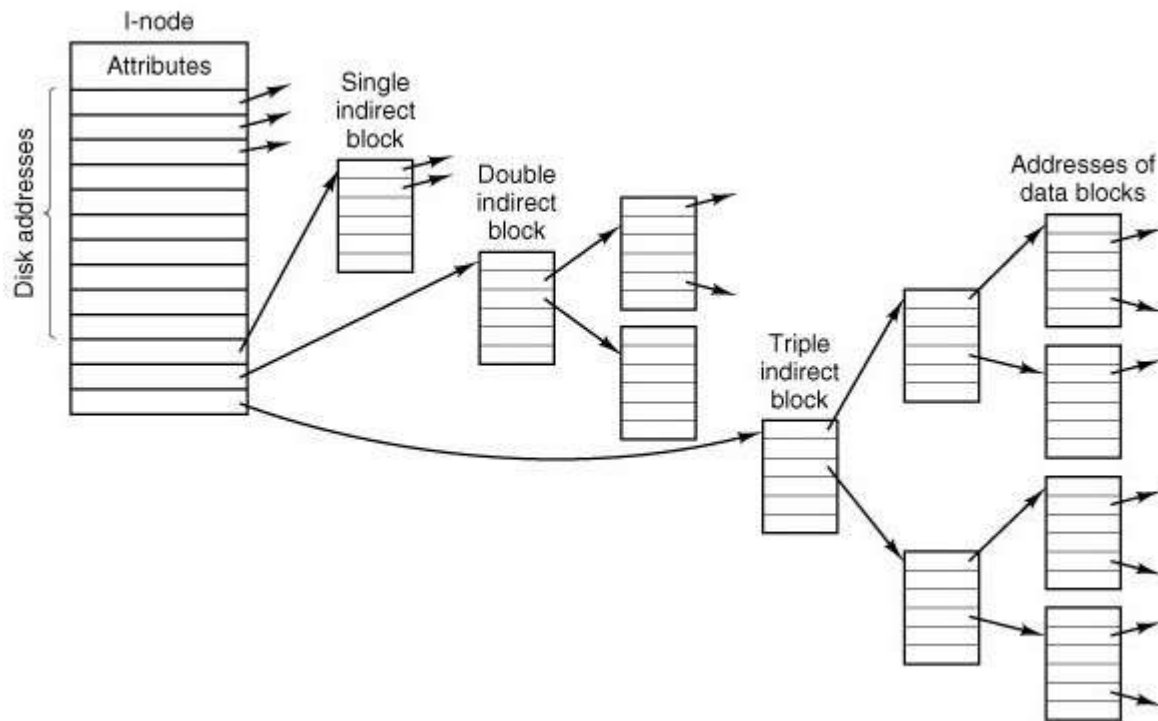


Fig: An i-node with three levels of indirect blocks.

File System Layout:

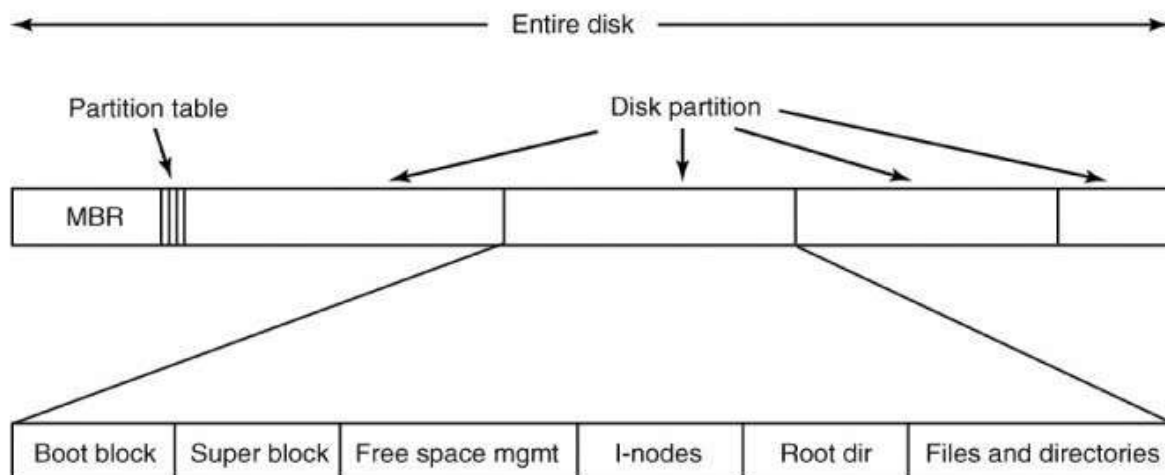


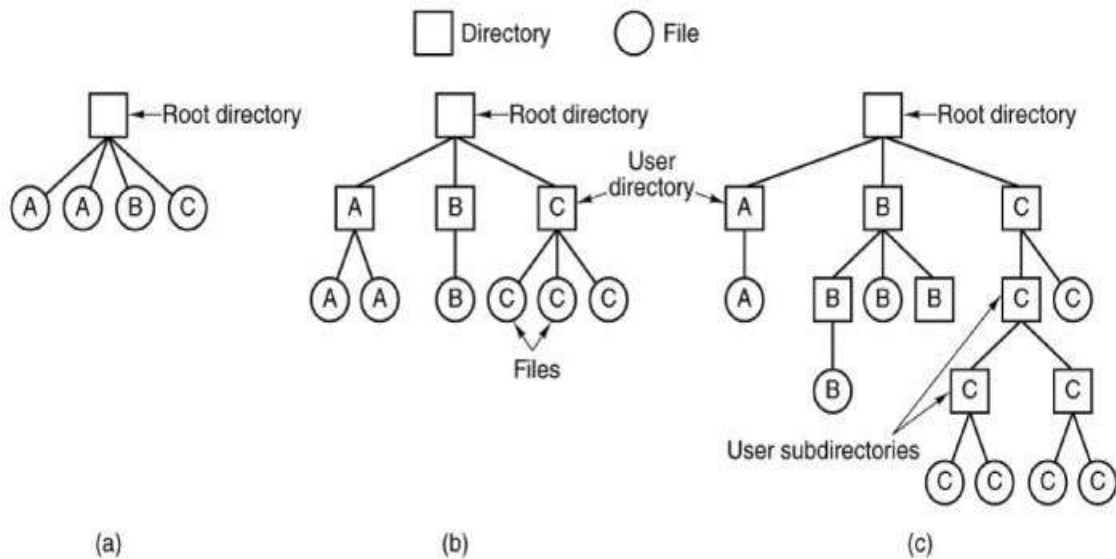
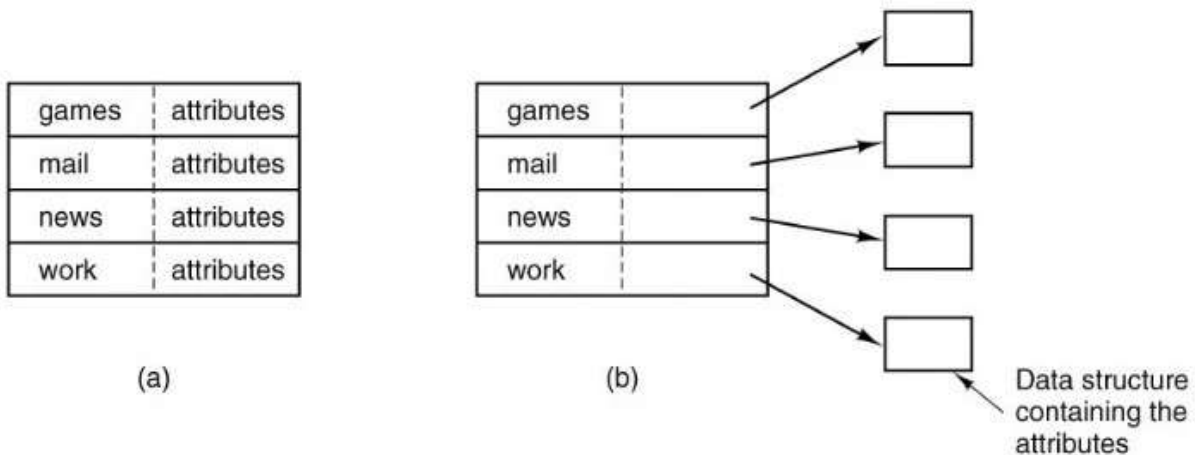
Fig:A possible file system layout.

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.