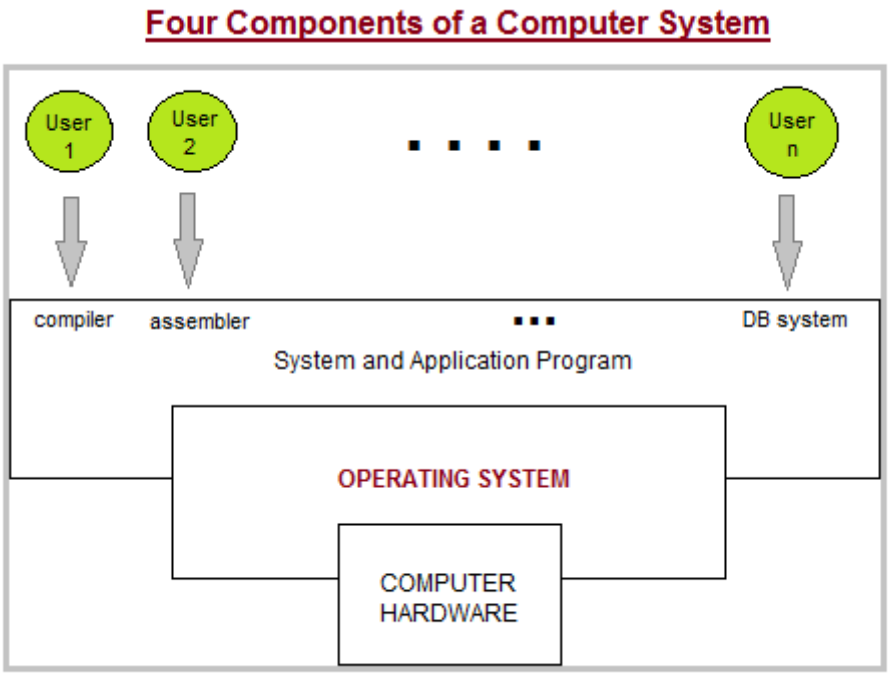
UNIT – 1

Introduction to Operating System Concept

Introduction to os:

Introduction to Operating Systems

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



Two Views of Operating System

1. User's View
2. System View

Operating System: User View

The user view of the computer refers to the interface being used. Such systems are designed for one user to monopolize its resources, to maximize the work that the user is performing. In

these cases, the operating system is designed mostly for ease of use, with some attention paid to performance, and none paid to resource utilization.

Operating System: System View

Operating system can be viewed as a resource allocator also. A computer system consists of many resources like - hardware and software - that must be managed efficiently. The operating system acts as the manager of the resources, decides between conflicting requests, controls execution of programs etc.

Operating System Management Tasks

1. **Processor management** which involves putting the tasks into order and pairing them into manageable size before they go to the CPU.
2. **Memory management** which coordinates data to and from RAM (random-access memory) and determines the necessity for virtual memory.
3. **Device management** which provides interface between connected devices.
4. **Storage management** which directs permanent data storage.
5. **Application** which allows standard communication between software and your computer.
6. **User interface** which allows you to communicate with your computer.

Functions of Operating System

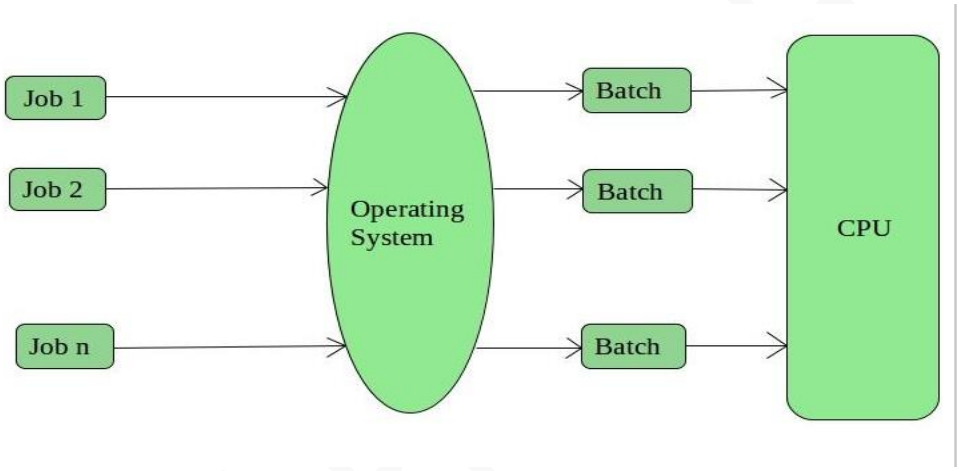
1. It boots the computer
2. It performs basic computer tasks e.g. managing the various peripheral devices e.g. mouse, keyboard
3. It provides a user interface, e.g. command line, graphical user interface (GUI)
4. It handles system resources such as computer's memory and sharing of the central processing unit(CPU) time by various applications or peripheral devices.
5. It provides file management which refers to the way that the operating system manipulates, stores, retrieves and saves data.
6. Error Handling is done by the operating system. It takes preventive measures whenever required to avoid errors.

Types of Operating Systems:

An Operating System performs all the basic tasks like managing file, process, and memory. Thus operating system acts as manager of all the resources, i.e. resource manager. Thus operating system becomes an interface between user and machine.

Types of Operating Systems: Some of the widely used operating systems are as follows-

1. ***Batch Operating System*** –
   This type of operating system do not interact with the computer directly. There is an operator which takes similar jobs having same requirement and group them into batches. It is the responsibility of operator to sort the jobs with similar needs.



Advantages of Batch Operating System:

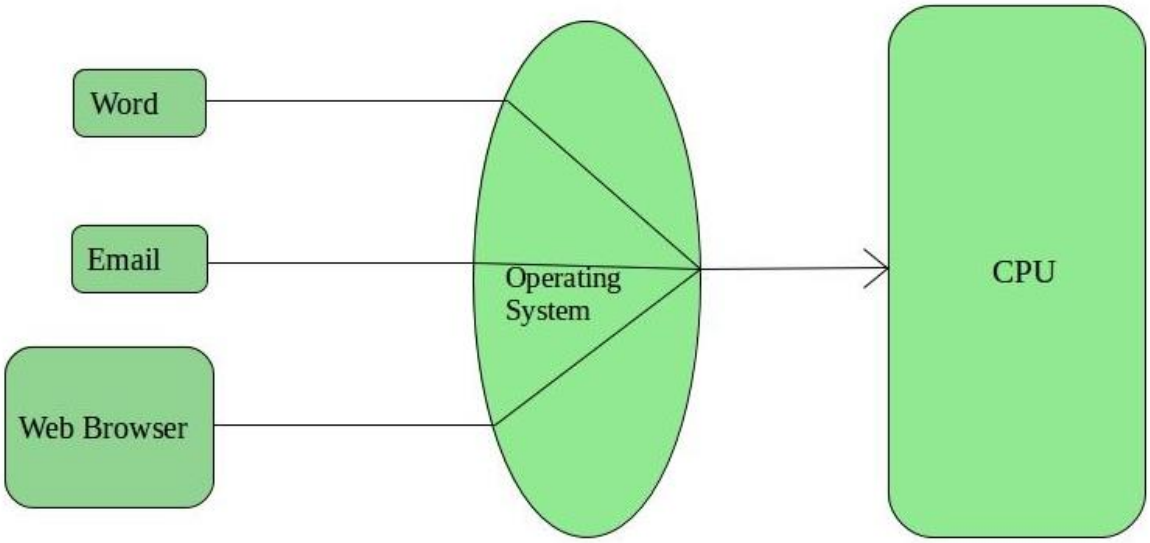
- It is very difficult to guess or know the time required by any job to complete. Processors of the batch systems knows how long the job would be when it is in queue
- Multiple users can share the batch systems
- The idle time batch system is very less
- It is easy to manage large work repeatedly in batch systems

Disadvantages of Batch Operating System:

- The computer operators should be well known with batch systems
- Batch systems are hard to debug
- It is sometime costly
- The other jobs will have to wait for an unknown time if any job fails

Examples of Batch based Operating System: Payroll System, Bank Statements etc.

2. **Time-Sharing Operating Systems –**
   Each task has given some time to execute, so that all the tasks work smoothly. Each user gets time of CPU as they use single system. These systems are also known as Multitasking Systems. The task can be from single user or from different users also. The time that each task gets to execute is called quantum. After this time interval is over OS switches over to next task.



Advantages of Time-Sharing OS:

- Each task gets an equal opportunity
- Less chances of duplication of software
- CPU idle time can be reduced

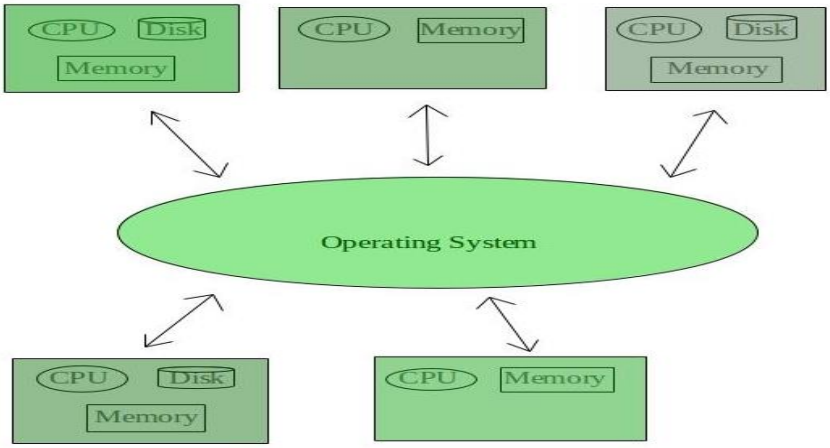
Disadvantages of Time-Sharing OS:

- Reliability problem
- One must have to take care of security and integrity of user programs and data
- Data communication problem

Examples of Time-Sharing OSs are: Multics, Unix etc.

3. Distributed Operating System –

These types of operating system is a recent advancement in the world of computer technology and are being widely accepted all-over the world and, that too, with a great pace. Various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU. These are referred as loosely coupled systems or distributed systems. These systems processors differ in sizes and functions. The major benefit of working with these types of operating system is that it is always possible that one user can access the files or software which are not actually present on his system but on some other system connected within this network i.e., remote access is enabled within the devices connected in that network.



Advantages of Distributed Operating System:

- Failure of one will not affect the other network communication, as all systems are independent from each other
- Electronic mail increases the data exchange speed
- Since resources are being shared, computation is highly fast and durable
- Load on host computer reduces
- These systems are easily scalable as many systems can be easily added to the network
- Delay in data processing reduces

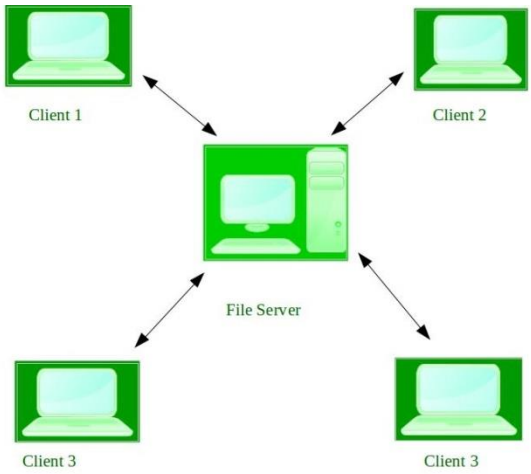
Disadvantages of Distributed Operating System:

- Failure of the main network will stop the entire communication
- To establish distributed systems the language which are used are not well defined yet
- These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet

Examples of Distributed Operating System are- LOCUS etc.

4. Network Operating System –

These systems runs on a server and provides the capability to manage data, users, groups, security, applications, and other networking functions. These type of operating systems allows shared access of files, printers, security, applications, and other networking functions over a small private network. One more important aspect of Network Operating Systems is that all the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that's why these computers are popularly known as tightly coupled systems.



Advantages of Network Operating System:

- Highly stable centralized servers
- Security concerns are handled through servers
- New technologies and hardware up-gradation are easily integrated to the system
- Server access are possible remotely from different locations and types of systems

Disadvantages of Network Operating System:

- Servers are costly
- User has to depend on central location for most operations
- Maintenance and updates are required regularly

Examples of Network Operating System are: Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD etc.

5. Real-Time Operating System –

These types of OSs serves the real-time systems. The time interval required to process and respond to inputs is very small. This time interval is called response time.

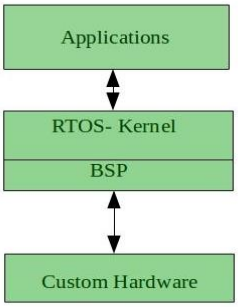
Real-time systems are used when there are time requirements are very strict like missile systems, air traffic control systems, robots etc.

Two types of Real-Time Operating System which are as follows:

- ***Hard Real-Time Systems:***

  These OSs are meant for the applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident. Virtual memory is almost never found in these systems.
- ***Soft Real-Time Systems:***

  These OSs are for applications where for time-constraint is less strict.



Advantages of RTOS:

- Maximum Consumption: Maximum utilization of devices and system,thus more output from all the resources
- Task Shifting: Time assigned for shifting tasks in these systems are very less. For example in older systems it takes about 10 micro seconds in shifting one task to another and in latest systems it takes 3 micro seconds.
- Focus on Application: Focus on running applications and less importance to applications which are in queue.
- Real time operating system in embedded system: Since size of programs are small, RTOS can also be used in embedded systems like in transport and others.
- Error Free: These types of systems are error free.
- Memory Allocation: Memory allocation is best managed in these type of systems.

Disadvantages of RTOS:

- Limited Tasks: Very few task run at the same time and their concentration is very less on few applications to avoid errors.
- Use heavy system resources: Sometimes the system resources are not so good and they are expensive as well.
- Complex Algorithms: The algorithms are very complex and difficult fro the designer to write on.
- Device driver and interrupt signals: It needs specific device drivers and interrupt signals to response earliest to interrupts.
- Thread Priority: It is not good to set thread priority as these systems are very less pron to switching tasks.

Examples of Real-Time Operating Systems are: Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

Topic 3: os services

An Operating System supplies different kinds of services to both the users and to the programs as well. It also provides application programs (that run within an Operating system) an environment to execute it freely. It provides users the services run various programs in a convenient manner.

Here is a list of common services offered by an almost all operating systems:

- User Interface
- Program Execution
- File system manipulation
- Input / output Operations
- Communication
- Resource Allocation
- Error Detection
- Accounting
- Security and protection

This chapter will give a brief description of what services an operating system usually provide to users and those programs that are and will be running within it.

User Interface of Operating System

Usually Operating system comes in three forms or types. Depending on the interface their types have been further sub divided. These are:

- Command line interface
- Batch based interface
- Graphical User Interface

Let's get to know in brie about each of them.

The command line interface (CLI) usually deals with using text commands and a technique for entering those commands. The batch interface (BI): commands and directives are used to manage those commands that are entered into files and those files get executed. Another type is the graphical user interface (GUI): which is a window system with a pointing device (like mouse or track-ball) to point to the I/O, choose from menus driven interface and to make choices viewing from a number of lists and a keyboard to entry the texts.

Program Execution in Operating System

The operating system must have the capability to load a program into memory and execute that program. Furthermore, the program must be able to end its execution, either normally or abnormally / forcefully.

File System Manipulation in Operating System

Programs need has to be read and then write them as files and directories. File handling portion of operating system also allows users to create and delete files by specific name along with extension, search for a given file and / or list file information. Some programs comprise of permissions management for allowing or denying access to files or directories based on file ownership.

I/O operations in Operating System

A program which is currently executing may require I/O, which may involve file or other I/O device. For efficiency and protection, users cannot directly govern the I/O devices. So, the OS provide a means to do I/O Input / Output operation which means read or write operation with any file.

Communication System of Operating System

Process needs to swap over information with other process. Processes executing on same computer system or on different computer systems can communicate using operating system support. Communication between two processes can be done using shared memory or via message passing.

Resource Allocation of Operating System

When multiple jobs running concurrently, resources must need to be allocated to each of them. Resources can be CPU cycles, main memory storage, file storage and I/O devices. CPU scheduling routines are used here to establish how best the CPU can be used.

Error Detection

Errors may occur within CPU, memory hardware, I/O devices and in the user program. For each type of error, the OS takes adequate action for ensuring correct and consistent computing.

Accounting

This service of operating system keeps track of which users are using how much and what kinds of computer resources has been used for accounting or simply to accumulate usage statistics.

Protection and Security

Protection includes in ensuring all access to system resources in a controlled manner. For making a system secure, the user needs to authenticate him or her to the system before using (usually via login ID and password).

Topic 4: Introduction to system call

Introduction of System Call

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface (API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Services Provided by System Calls:

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

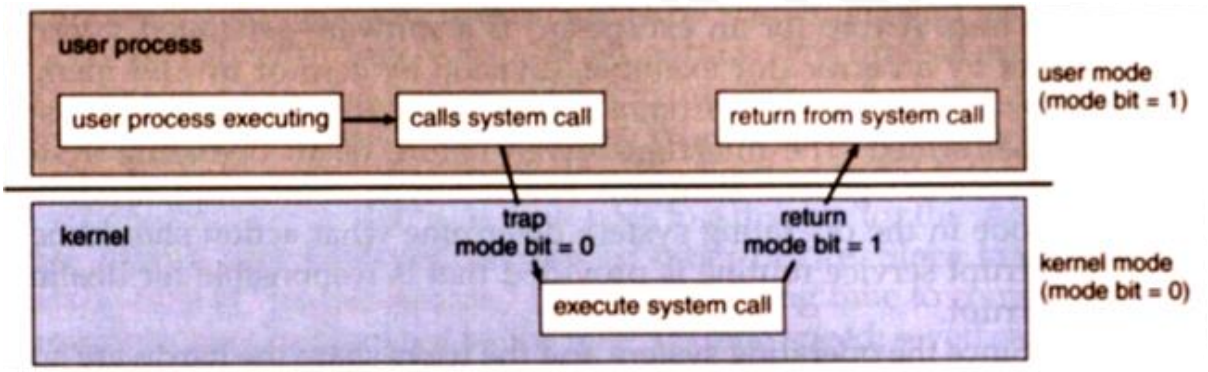
Types of System Calls: There are 5 different categories of system calls –

1. **Process control:** end, abort, create, terminate, allocate and free memory.
2. **File management:** create, open, close, delete, read file etc.
3. **Device management**
4. **Information maintenance**
5. **Communication**

Examples of Windows and UNIX System Calls –

	WINDOWS	**UNIX**
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()

Information Maintenance	GetCurrentProcessID() SetTimer() Sleeo()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()



The system call provides an interface to the operating system services.

Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls. By using the API, certain benefits can be gained:

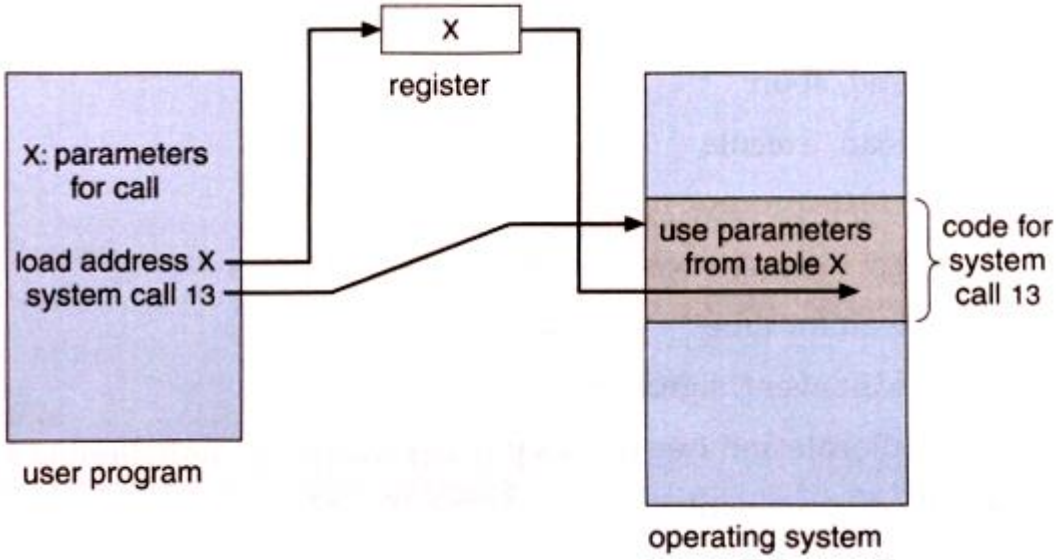
- Portability: as long a system supports an API, any program using that API can compile and run.
- Ease of Use: using the API can be significantly easier then using the actual system call.

1.12.1. System Call Parameters

Three general methods exist for passing parameters to the OS:

1. Parameters can be passed in registers.

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



1.12.2. Types of System Calls

There are 5 different categories of system calls:

process control, file manipulation, device manipulation, information maintenance and communication.

1.12.2.1. Process Control

A running program needs to be able to stop execution either normally or abnormally. When execution is stopped abnormally, often a dump of memory is taken and can be examined with a debugger.

1.12.2.2. File Management

Some common system calls are *create*, *delete*, *read*, *write*, *reposition*, or *close*. Also, there is a need to determine the file attributes – *get* and *set* file attribute. Many times the OS provides an API to make these system calls.

1.12.2.3. Device Management

Process usually require several resources to execute, if these resources are available, they will be granted and control returned to the user process. These resources are also thought of as devices. Some are physical, such as a video card, and others are abstract, such as a file.

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

1.12.2.4. Information Management

Some system calls exist purely for transferring information between the user program and the operating system. An example of this is *time*, or *date*.

The OS also keeps information about all its processes and provides system calls to report this information.

1.12.2.5. Communication

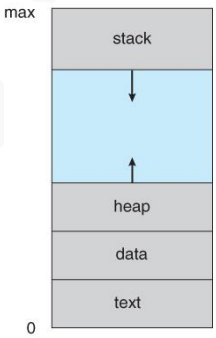
There are two models of interprocess communication, the message-passing model and the shared memory model. Message-passing uses a common mailbox to pass messages between processes. Shared memory use certain system calls to create and gain access to create and gain access to regions of memory owned by other processes. The two processes exchange information by reading and writing in the shared data.

UNIT-2
Process Management

2.1 WHAT IS A PROCESS?

Process is a program in execution. A process is an instance of a program running in a computer. It is close in meaning to task , a term used in some operating systems. In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program).

- On execution, this program may read in some data and output some data.
- Note that when a program is written and a file is prepared, it is still a script (passive entity). It has no dynamics of its own i.e, it cannot cause any input processing or output to happen.
- Once we compile, and later when we run this program, the intended operations take place (active entity). In other words, a program is a text script with no dynamic Behavior.
- When a program is in execution, the script is acted upon. It can result in engaging a processor for some processing and it can also engage in I/O operations.
- It is for this reason a process is differentiated from program.
- While the program is a text script (often called executable file), a program in execution is a process.



2.1.1 THE PROCESS

- Process memory is divided into four sections as shown in Figure 3.1 below:
  - The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
  - The data section stores global and static variables, allocated and initialized prior to executing main.
  - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
  - The stack is used for local variables. Space on the stack is reserved for local variables when they are declared and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.

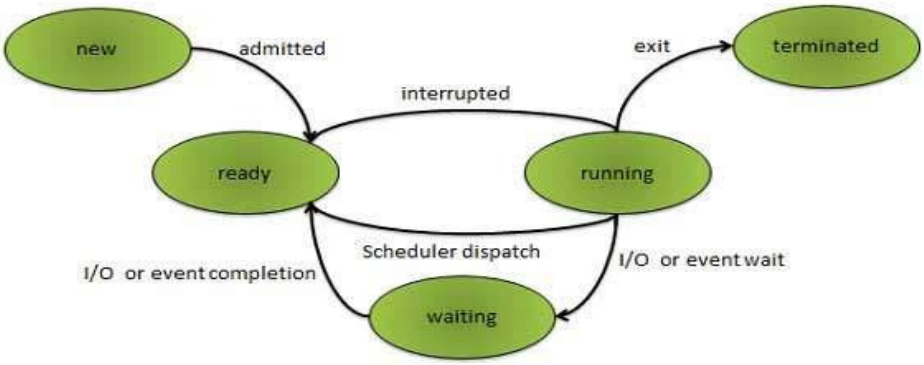
- Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.

2.1.2 Process state:

As a process executes, it changes state. The state of a process is defined as the current activity of the process.

Process can have one of the following five states at a time.

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



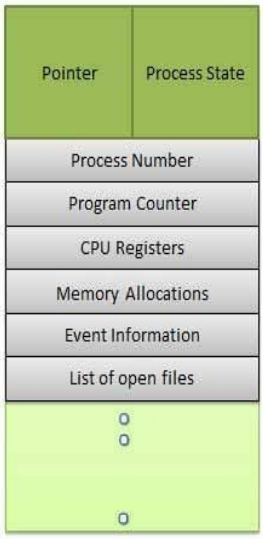
2.1.3 Process Control Block, PCB:

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

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

S.N.	Information & Description
1	**Pointer** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2	**Process State** Process state may be new, ready, running, waiting and so on.
3	**Program Counter** Program Counter indicates the address of the next instruction to be executed for this process.
4	**CPU registers** CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

5	**Memory management information**: This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for deallocating the memory when the process terminates.
6	**Accounting information** This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.



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

2.2 PROCESS SCHEDULING:

Definition

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

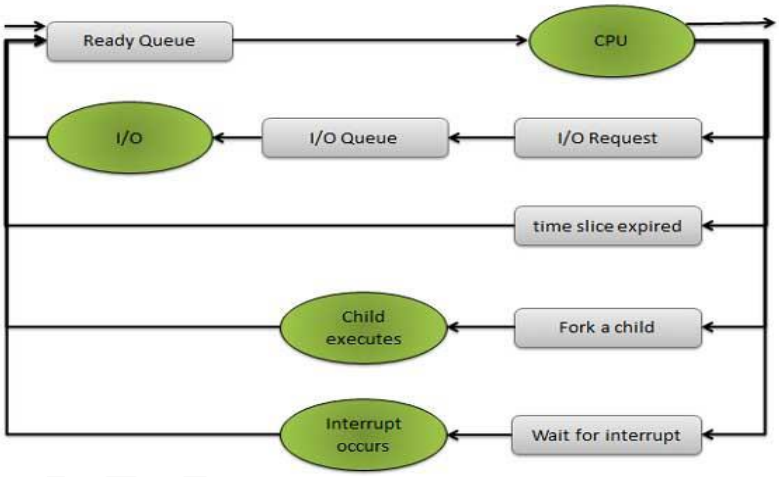
Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

2.2.1 Scheduling Queues

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a **job queue**. This queue consists of all processes in the system. The operating system also maintains other queues such as **device queue**. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Queues are of three types

- Job queue
- Ready queue
- Device queue

A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

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

2.2.2 Schedulers

Schedulers are special system software's which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

Schedulers are of three types

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

Long Term Scheduler

It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into **memory** for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide **a balanced mix of jobs**, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

When process changes the state from new to ready, then there is use of long term scheduler.

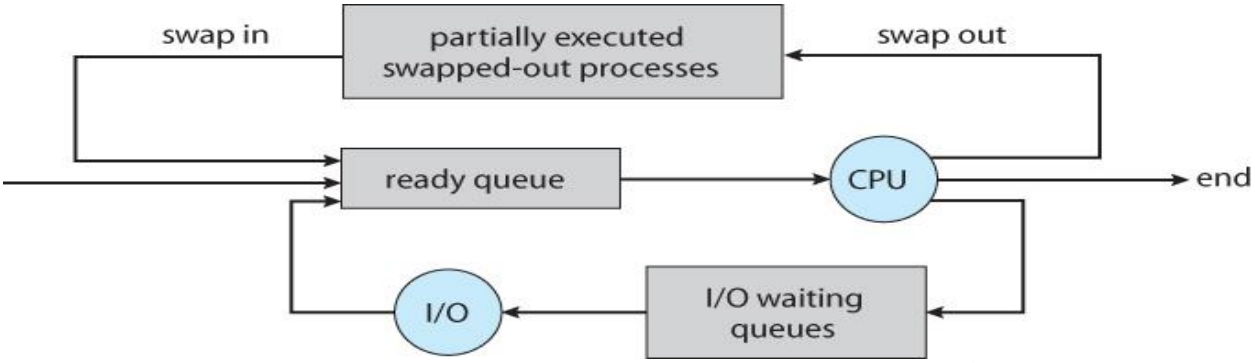
Short Term Scheduler

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

Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

Medium term scheduling is part of the **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.



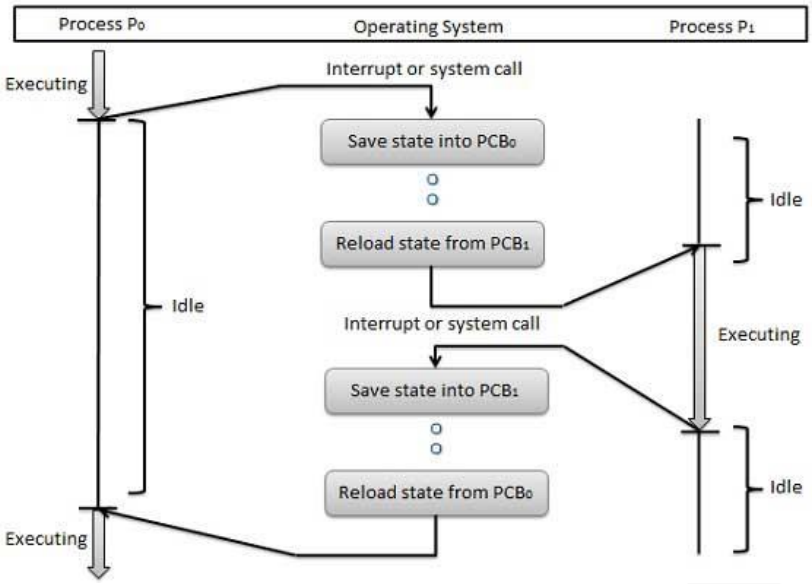
Running process may become **suspended** if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the **secondary storage**. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

2.2.3 Context Switch

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

When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support. Context switch requires (n + m) bxK time units to save the state of the processor with n general registers, assuming b are the store operations are required to save n and m registers of two process control blocks and each store instruction requires K time units.



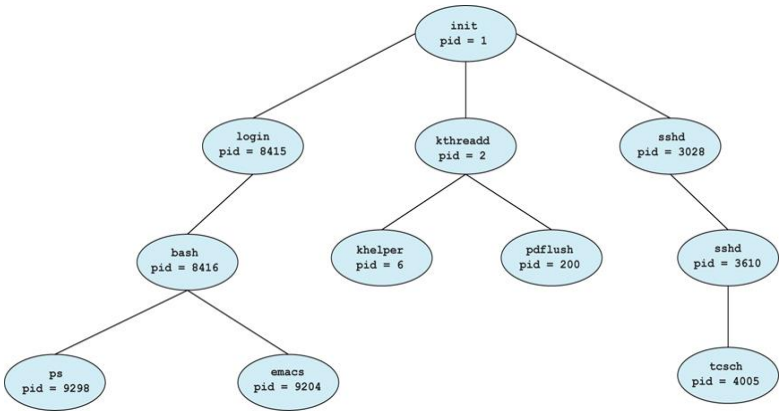
Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

- Program Counter
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting

2.3 Operations on Processes

2.3.1 Process Creation

- Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.
- Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 3.9 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:



- Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.

There are two possibilities for the parent process after creating the child:

1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.

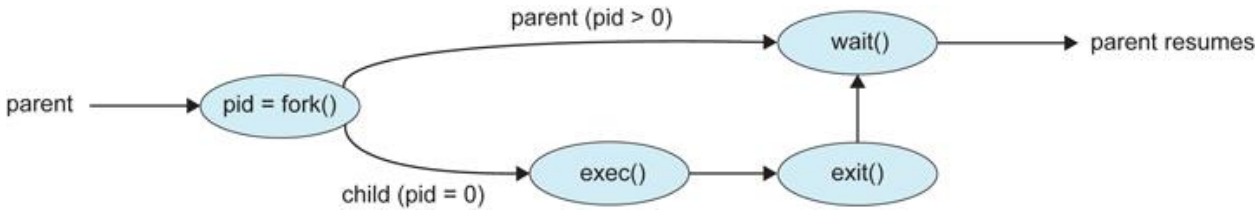
2. The parent waits until some or all its children have terminated

Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)

Two possibilities for the address space of the child relative to the parent:

- The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the fork system call in UNIX.
- The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the spawn system calls in Windows. UNIX systems implement this as a second step, using the exec system call.

Figures below shows the fork and exec process on a UNIX system. Note that the fork system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which. Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.



Process creation using the fork ( ) system call

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
    }
    else if (pid == 0) {/* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
      exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

2.3.2 Process Termination

Processes may request their own termination by making the exit ( ) system call, typically returning an int. This int is passed along to the parent if it is doing a wait ( ), and is typically zero on successful completion and some non-zero code in the event of problems.

Child code:

intexitCode;

exit(exitCode); // return exitCode; has the same effect when executed from main( )

parent code:

pid_tpid;

int status

pid = wait(&status);

// pid indicates which child exited. exitCode in low-order bits of status

// macros can test the high-order bits of status for why it stopped

Processes may also be terminated by the system for a variety of reasons, including:

- The inability of the system to deliver necessary system resources.
- In response to a KILL command, or other un handled process interrupt.
- A parent may kill its children if the task assigned to them is no longer needed.
- If the parent exits, the system may or may not allow the child to continue without a parent.
- When a process terminates, all of its system resources are freed up, open files flushed and closed

2.3 **Inter process Communication:**

There are two types of processes

1. Independent Processes

2. Cooperating Processes

- Independent Processes operating concurrently on systems are those that can neither affect other processes or be affected by other processes.
- Cooperating Processes are those that can affect or be affected by other processes.

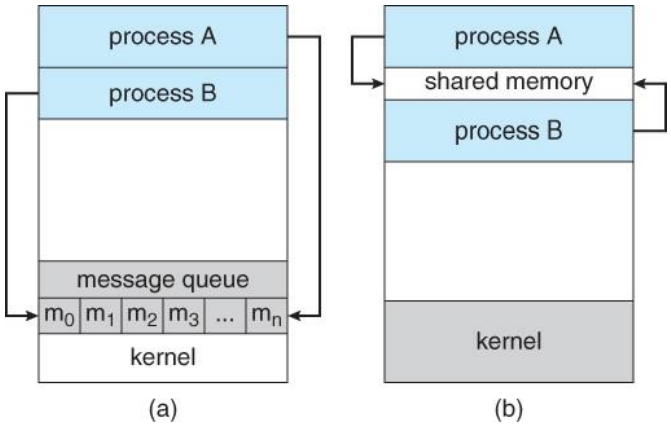
There are several reasons why cooperating processes are allowed:

- **Information Sharing** - There may be several processes which need access to the same file for example. (e.g. pipelines.)
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
- **Modularity** - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
- **Convenience** - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Cooperating processes require some type of inter-process communication, which is most commonly one of two types:

Shared Memory systems and Message Passing systems.

Figure illustrates the difference between the two systems:



- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.

- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

➢ 2.3.1 **Shared-Memory Systems**

In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.

Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.

Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)

The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.

```
#define BUFFER_SIZE 10

typedefstruct {
```

```
    . . .
} item;


item buffer[ BUFFER_SIZE ];
int in = 0;
int out = 0;
```

➢ 2.3.2 Message-Passing Systems

Message passing systems must support at a minimum system calls for "send message" and "receive message". A communication link must be established between the cooperating processes before messages can be sent.

There are three key issues to be resolved in message passing systems as further explored in the next three subsections:

- Direct or indirect communication (naming)
- Synchronous or asynchronous communication
- Automatic or explicit buffering.

2.3.2.1Direct or indirect communication (Naming):

- With direct communication the sender must know the name of the receiver to which it wishes to send a message.
- There is a one-to-one link between every sender-receiver pair.
- For symmetric communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
- For asymmetric communications, this is not necessary.

2.3.2.2 Indirect communication: it uses shared mailboxes, or ports.

- Multiple processes can share the same mailbox or boxes.
- Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
- The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

2.3.2.3 Synchronization:

Either the sending or receiving of messages (or neither or both) may be either blocking or non-blocking.

Blocking send: the sending process is blocked until the message is received by the receiving process or the mail box.

Non blocking send: the sending process sends the message and resumes operation.

Blocking receive: the receiver blocks until a message is available.

Non blocking receiver: the receiver receives either a valid message or a null

2.3.2.4 Buffering: whether communication is direct or indirect message exchange by communicating processes resides in temporary queue., which may have one of three capacity configurations:

- Zero capacity - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
- Bounded capacity- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
- Unbounded capacity - The queue has a theoretical infinite capacity, so senders are never forced to block.

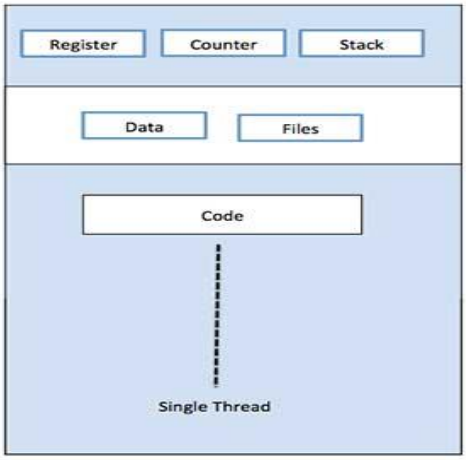
2.4 Multi thread programming model:

What is Thread?
A thread is a flow of execution through the process code, with its own program counter that keeps rack of which instruction to execute next, system registers which hold its current working variables and stack which contains the execution history.

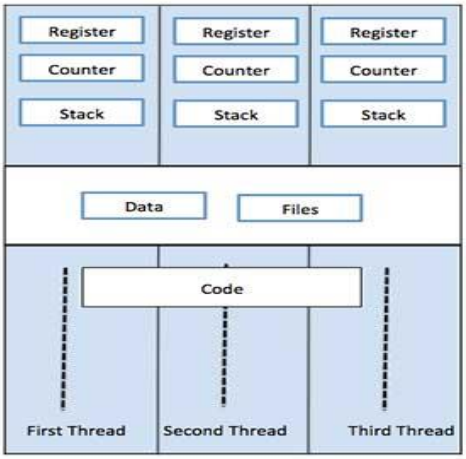
A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads has been successfully used in implementing network servers and web server. They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.



Single Process P with single thread



Single Process P with three threads

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

Advantages of Thread:

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

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**

Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

2. **Resource sharing**

By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **Economy**

Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads.

In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

4. **Scalability:**

The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only run on one CPU, no matter how many are available.

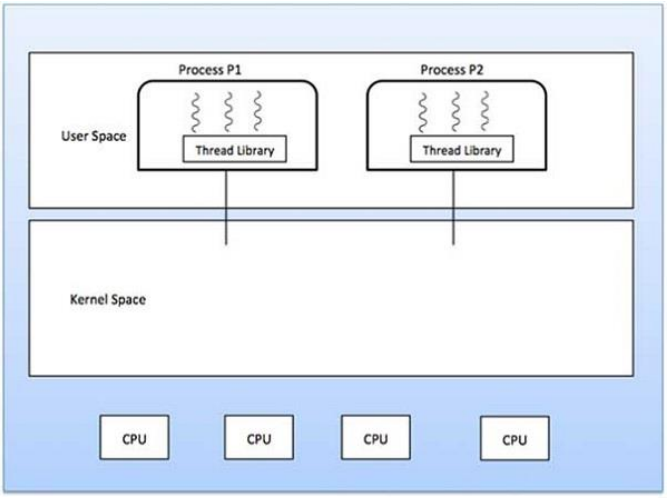
Types of Thread

Threads are implemented in following two ways −

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

User Level Threads

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



Advantages

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

Disadvantages

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

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads.

Advantages

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

Disadvantages

- Kernel threads are generally slower to create and manage than the user threads.

- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

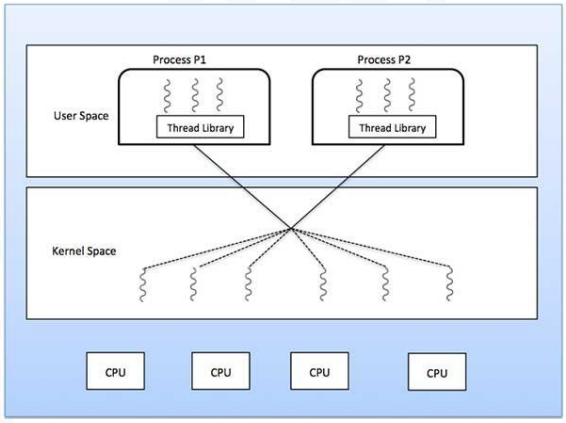
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the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

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Models:

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

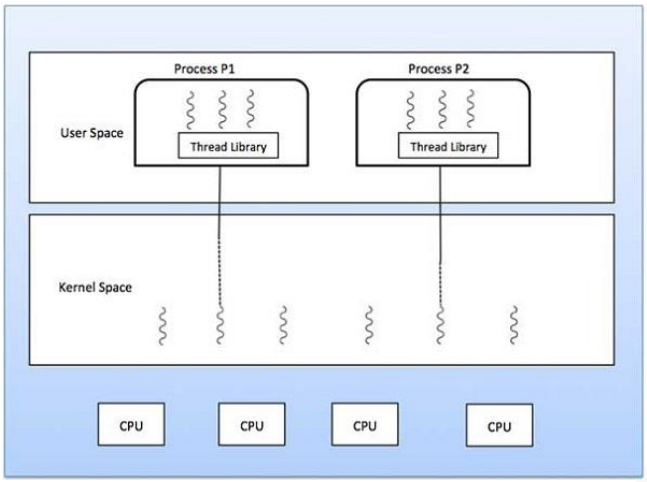
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model:

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

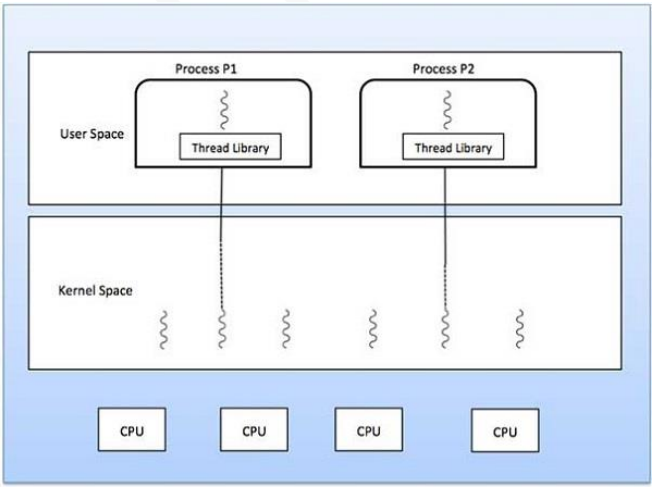
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model:

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread

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

2.5 CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Scheduling criteria:

There are many different criteria’s to check when considering the "best" scheduling algorithm :

- **CPU utilization**

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

- **Throughput**

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

- **Turnaround time**

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

- **Waiting time**

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

- **Response time**

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

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

Scheduling algorithms:

We'll discuss four major scheduling algorithms here which are following :

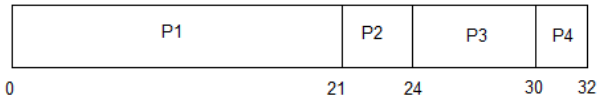
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

First Come First Serve (FCFS) Scheduling:

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

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



This is the GANTT chart for the above processes

Shortest-Job-First (SJF) Scheduling:

Shortest job next (**SJN**), also known as **shortest job first** (**SJF**) is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

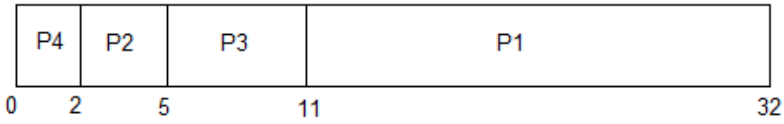
- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.

- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

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

Hence the GANTT chart will be following :



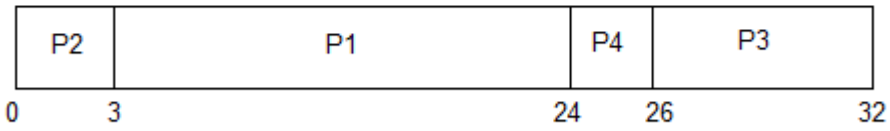
Now, the average waiting time will be = (0 + 2 + 5 + 11)/4 = 4.5 ms

Priority Scheduling:

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

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

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



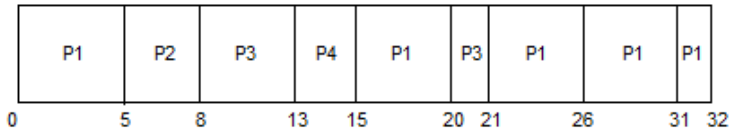
The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Round Robin(RR) Scheduling:

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

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Multilevel Queue Scheduling:

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc.

One general classification of the processes is foreground processes and background processes. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into.

Each queue will be assigned a priority and will have its own scheduling algorithm like Round-robin schedulingor FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues

- Multiple queues are maintained for processes.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

Multilevel feedback queue:

Unlike multilevel queue scheduling algorithm where processes are permanently assigned to a queue, multilevel feedback queue scheduling allows a process to move between queues. This movement is facilitated by the characteristic of the CPU burst of the process. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher priority queue. This form of aging also helps to prevent starvation of certain lower priority processes.

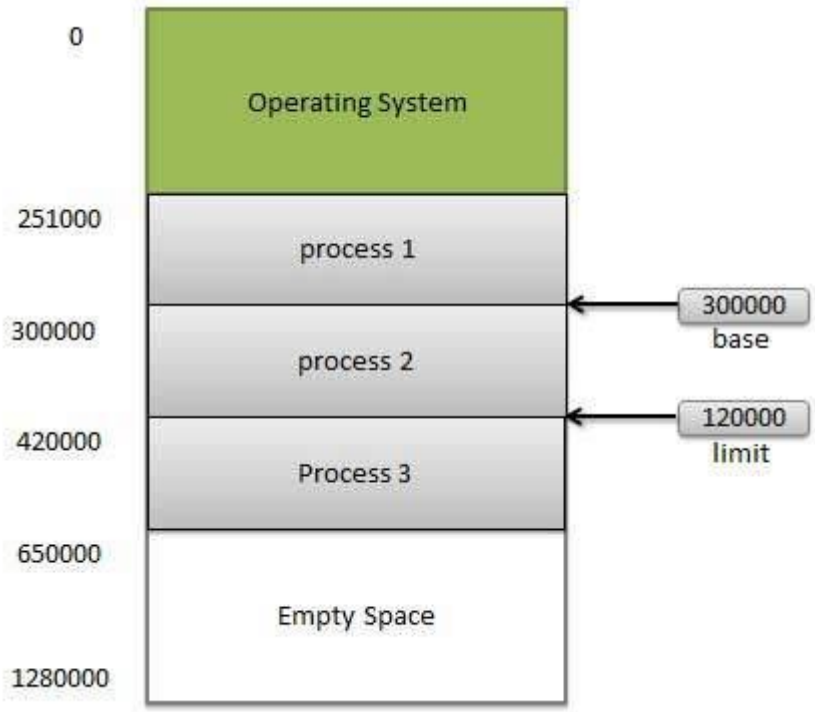
Unit -3

Memory Management

Introduction to Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Memory management provides protection by using two registers, a base register and a limit register. The base register holds the smallest legal physical memory address and the limit register specifies the size of the range. For example, if the base register holds 300000 and the limit register is 120000, then the program can legally access all addresses from 300000 through 419999.



Instructions and data to memory addresses can be done in following ways

- **Compile time** -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.
- **Load time** -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.
- **Execution time** -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

Dynamic Loading

In dynamic loading, a routine of a program is not loaded until it is called by the program. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed. Other routines methods or modules are loaded on request. Dynamic loading makes better memory space utilization and unused routines are never loaded.

Dynamic Linking

Linking is the process of collecting and combining various modules of code and data into a executable file that can be loaded into memory and executed. Operating system can link system level libraries to a program. When it combines the libraries at load time, the linking is called static linking and when this linking is done at the time of execution, it is called as dynamic linking.

In static linking, libraries linked at compile time, so program code size becomes bigger whereas in dynamic linking libraries linked at execution time so program code size remains smaller.

Logical versus Physical Address Space

An address generated by the CPU is a logical address whereas address actually available on memory unit is a physical address. Logical address is also known a Virtual address.

Virtual and physical addresses are the same in compile-time and load-time address-binding schemes. Virtual and physical addresses differ in execution-time address-binding scheme.

The set of all logical addresses generated by a program is referred to as a logical address space. The set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

The run-time mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device. MMU uses following mechanism to convert virtual address to physical address.

- The value in the base register is added to every address generated by a user process which is treated as offset at the time it is sent to memory. For example, if the base register value is 10000, then an attempt by the user to use address location 100 will be dynamically reallocated to location 10100.
- The user program deals with virtual addresses; it never sees the real physical addresses.

Swapping

Swapping is a mechanism in which a process can be swapped temporarily out of main memory to a backing store, and then brought back into memory for continued execution.

Backing store is a usually a hard disk drive or any other secondary storage which fast in access and large enough to accommodate copies of all memory images for all users. It must be capable of providing direct access to these memory images.

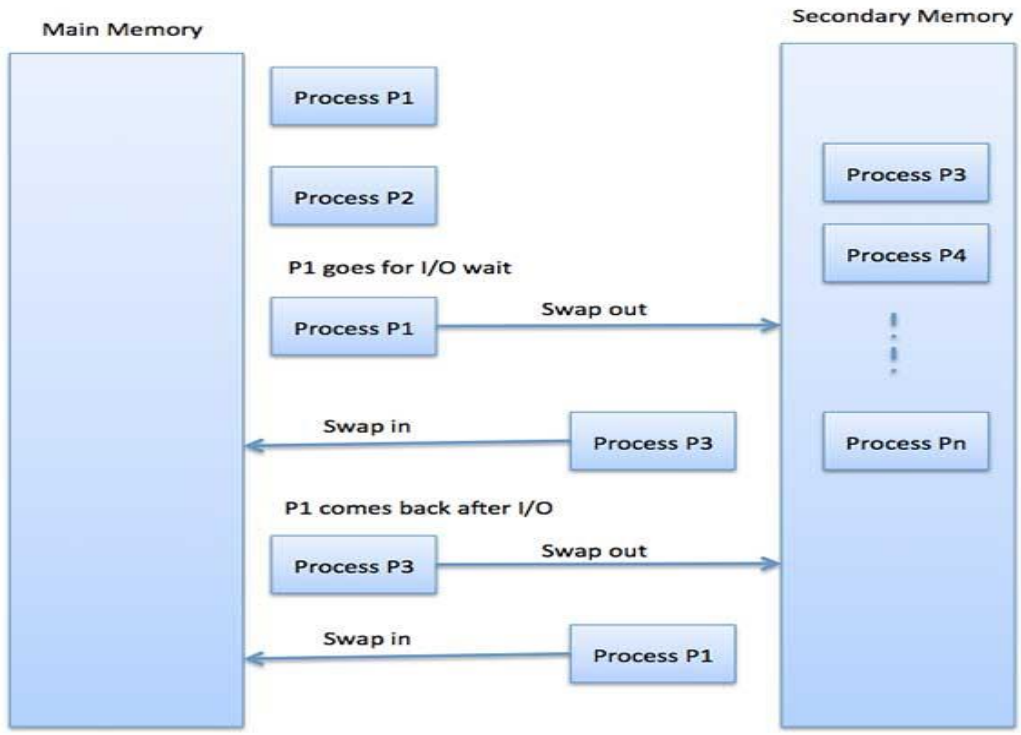
Major time consuming part of swapping is transfer time. Total transfer time is directly proportional to the amount of memory swapped. Let us assume that the user process is of size

100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second. The actual transfer of the 100K process to or from memory will take

100KB / 1000KB per second

= 1/10 second

= 100 milliseconds



4.3.1.1.Lecture-2

Contiguous Allocation

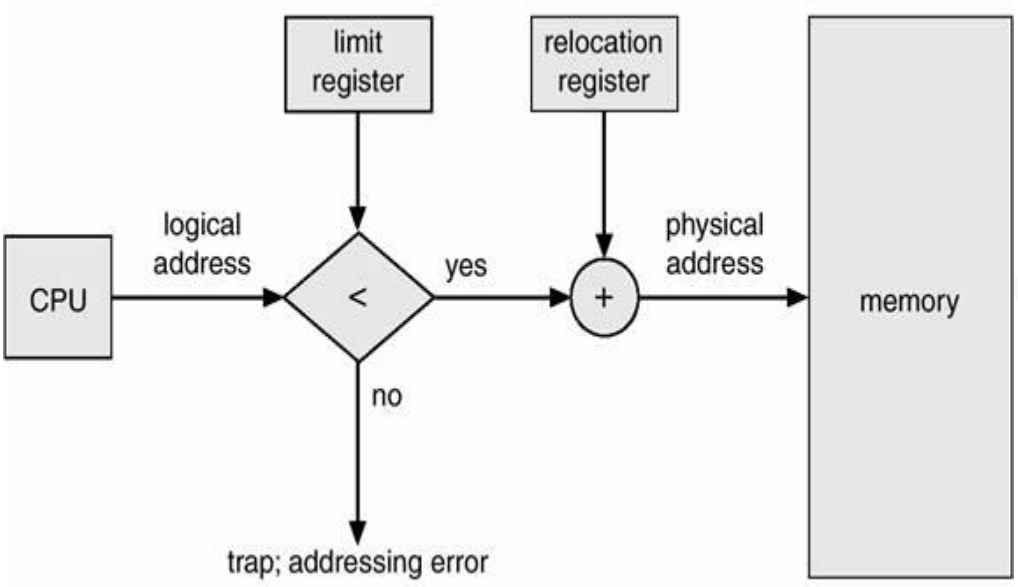
Main memory usually into two partitions:

✦ User processes then held in high memory. Single-partition allocation

✦ Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.

✦ Relocation register contains value of smallest physical address; limit registercontains range of logical addresses – each logical address must be less than the limit register.

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

✦*Hole* – block of available memory; holes of various size are scattered throughout memory.

✦ When a process arrives, it is allocated memory from a hole large enough to accommodate it.

✦ Operating system maintains information about:

a) Allocated partitions b) free partitions (hole)

First-fit: Allocate the *first* hole that is big enough.

Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size.Produces the smallest leftover hole.

Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

Fragmentation

External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous.

Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Reduce external fragmentation by compaction

- ✦ Shuffle memory contents to place all free memory together in one large block.
- ✦ Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- ✦ I/O problem

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames.
- To run a program of size *n* pages, need to find *n* free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

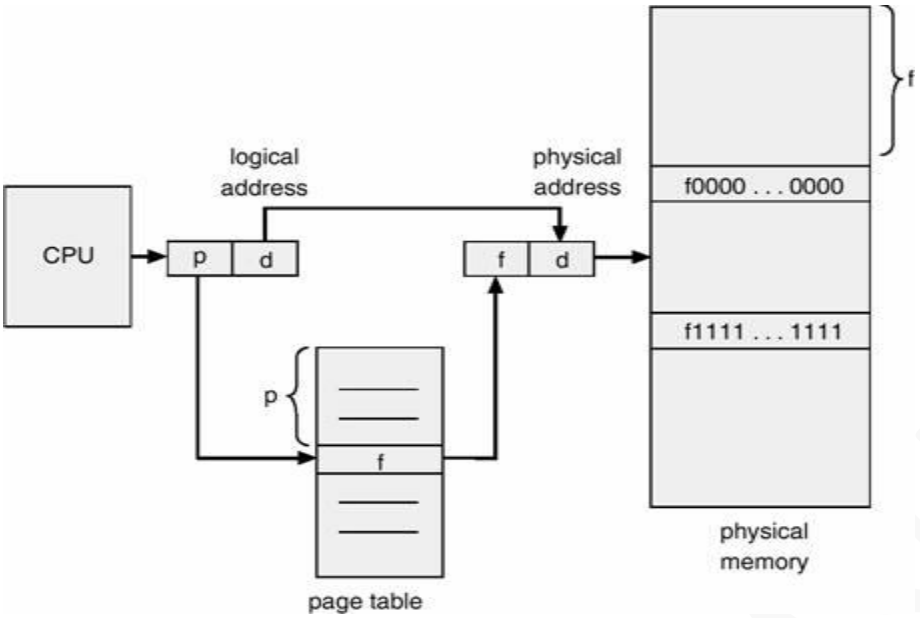
Address Translation Scheme

Address generated by CPU is divided into:

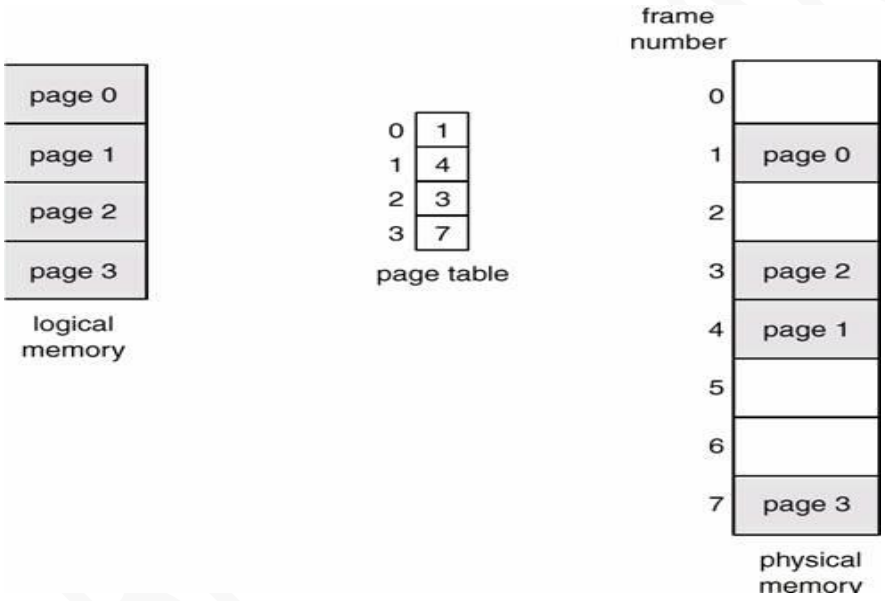
1. *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.

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

Address Translation Architecture

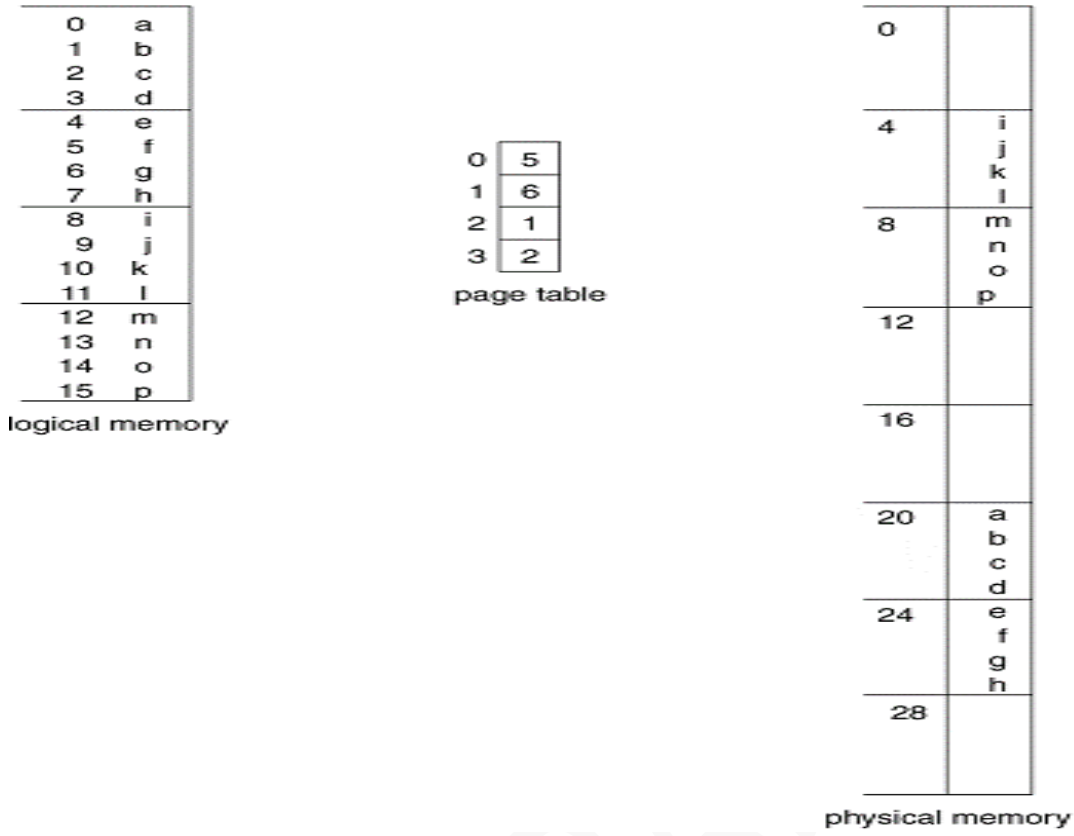


Paging Example

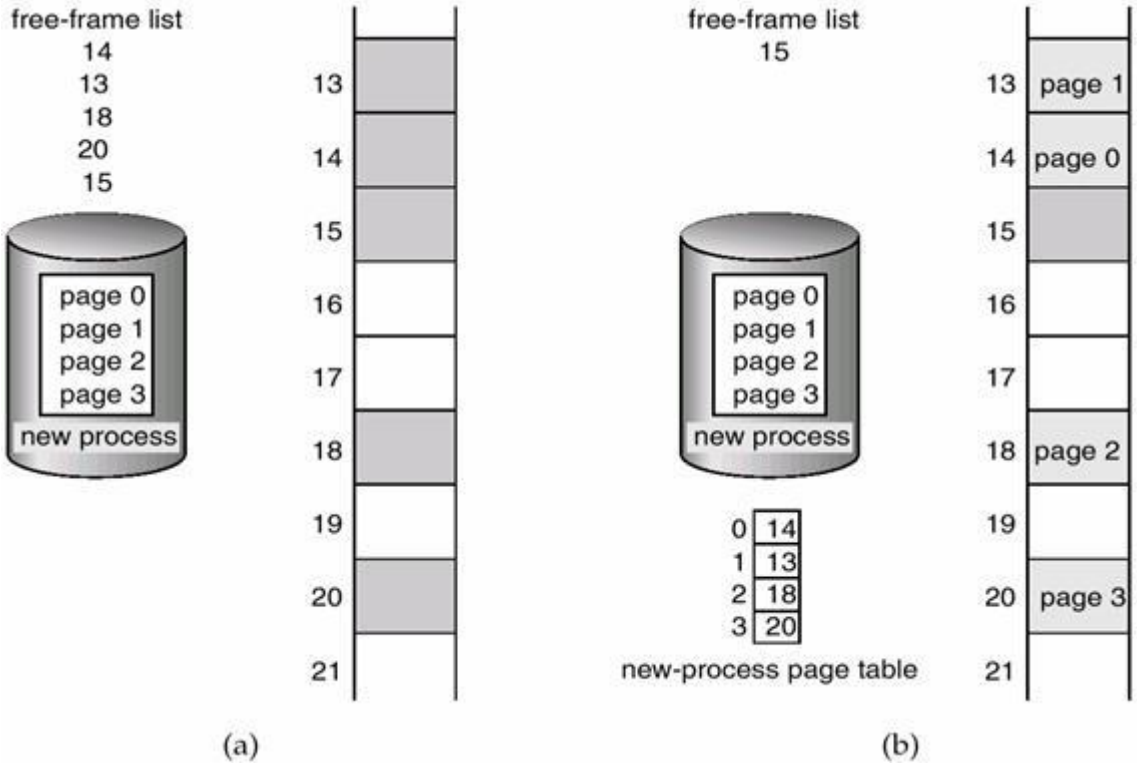


4.3.1.1.Lecture-3

Paging Example



Free Frames



Implementation of Page Table

- Page table is kept in main memory.
- *Page-table base register (*PTBR) points to the page table.
- *Page-table length register* (PRLR) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special

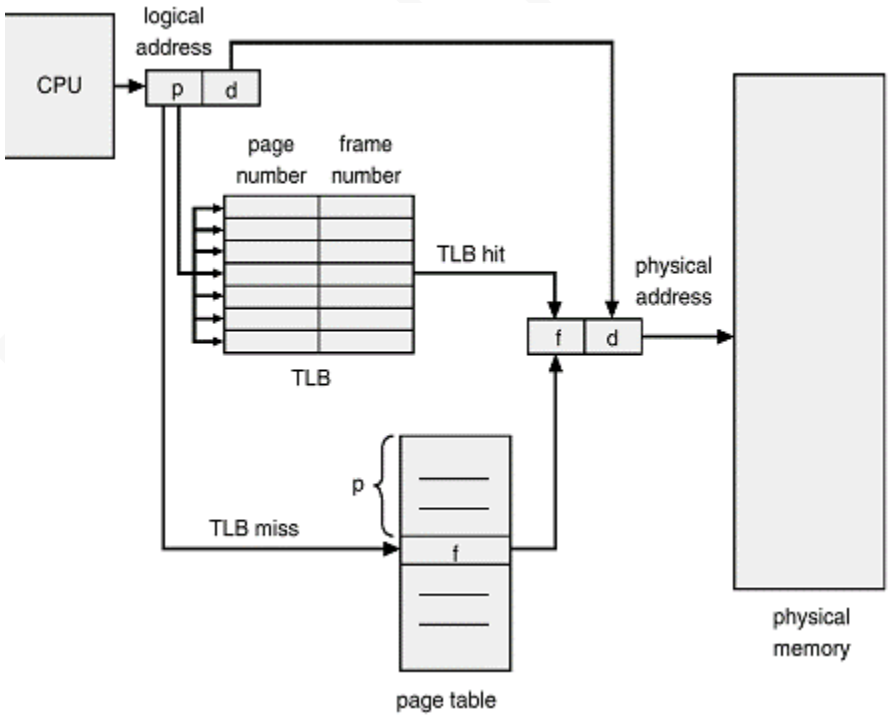
Associative Memory

Associative memory – parallel search Address translation (A´, A´´)

1. If A´ is in associative register, get frame # out.

2. Otherwise get frame # from page table in memory

Paging Hardware with TLB



Effective Access Time

Associative Lookup = å time unit

Assume memory cycle time is 1 microsecond

Hit ratio – percentage of times that a page number is found in the associative registers; ration related to

number of associative registers.

Hit ratio = a

Effective Access Time (EAT)

EAT = (1 + å) a + (2 + å)(1 – a)

= 2 + å – a

Memory Protection

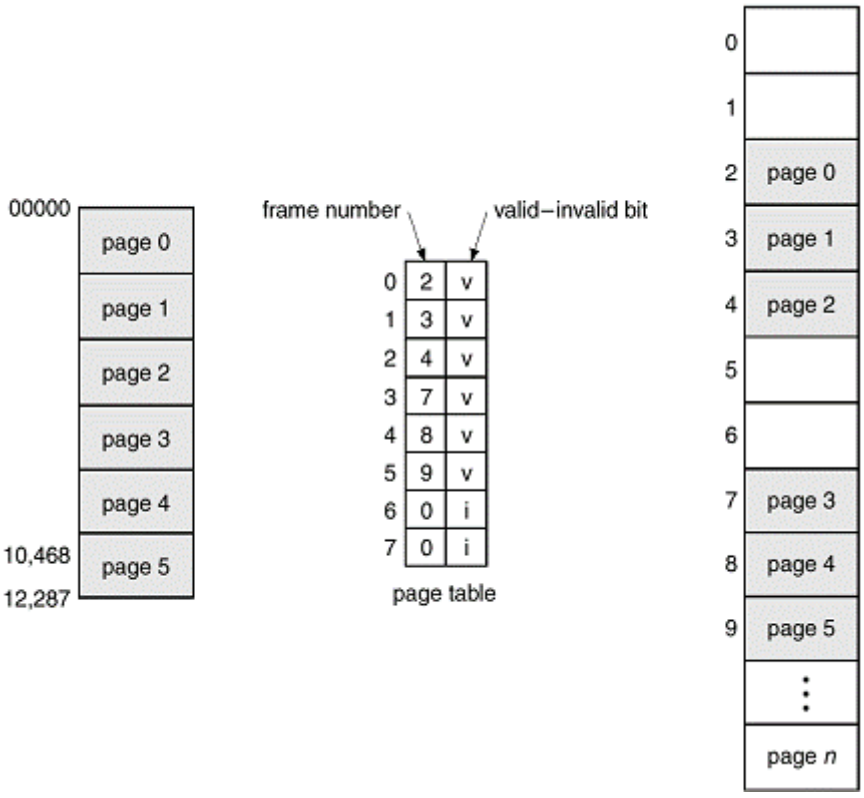
Memory protection implemented by associating protection bit with each frame.

***Valid-invalid* bit attached to each entry in the page table:**

1. "Valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.

2. "invalid" indicates that the page is not in the process' logical address space.

Valid (v) or Invalid (i) Bit In A Page Table



Page Table Structure

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

Hierarchical Page Tables

Break up the logical address space into multiple page tables.

A simple technique is a two-level page table.

Two-Level Paging Example

A logical address (on 32-bit machine with 4K page size) is divided into:

1. a page number consisting of 20 bits.
2. a page offset consisting of 12 bits.

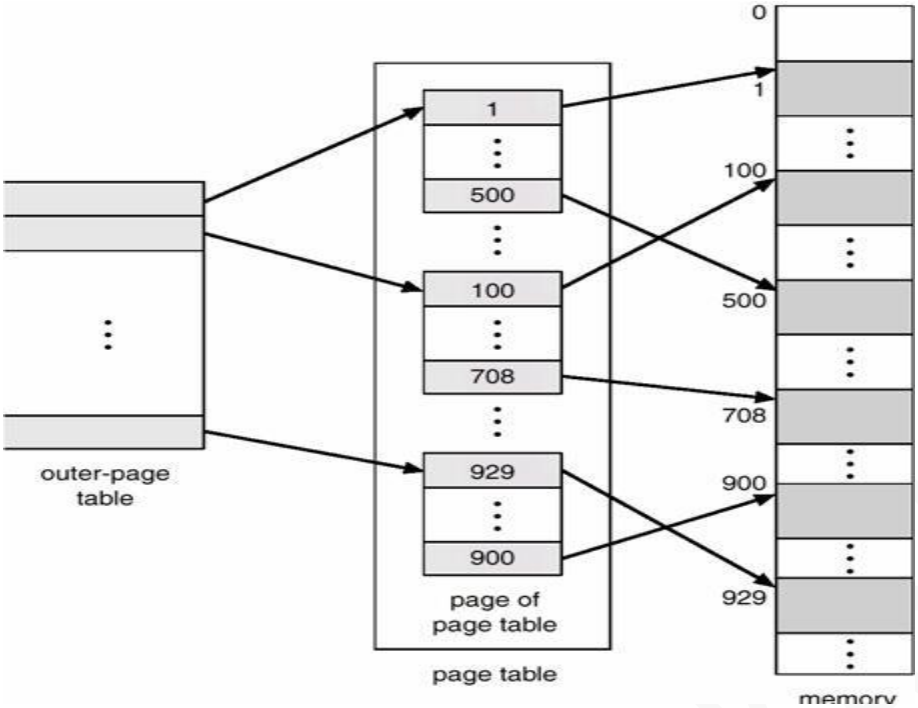
Since the page table is paged, the page number is further divided into:

3. a 10-bit page number.
4. a 10-bit page offset.
5. Thus, a logical address is as follows:

page number		page offset
*p*i	*p*2	*d*
10	10	12

where*pi* is an index into the outer page table, and *p2* is the displacement within the page of the outer page table.

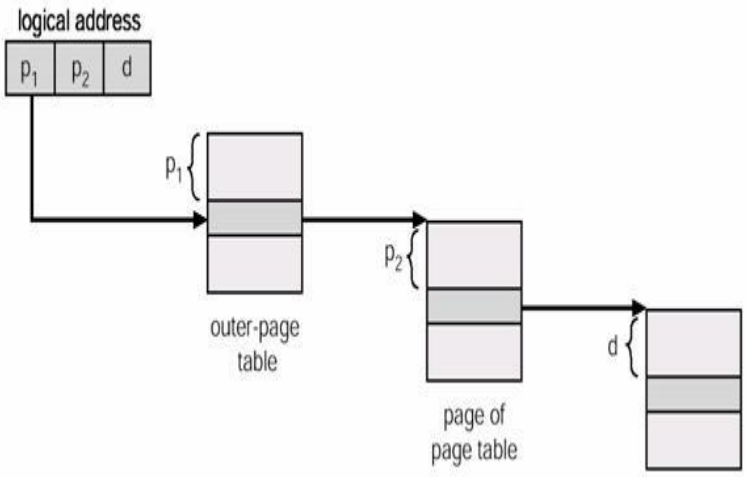
Two-Level Page-Table Scheme



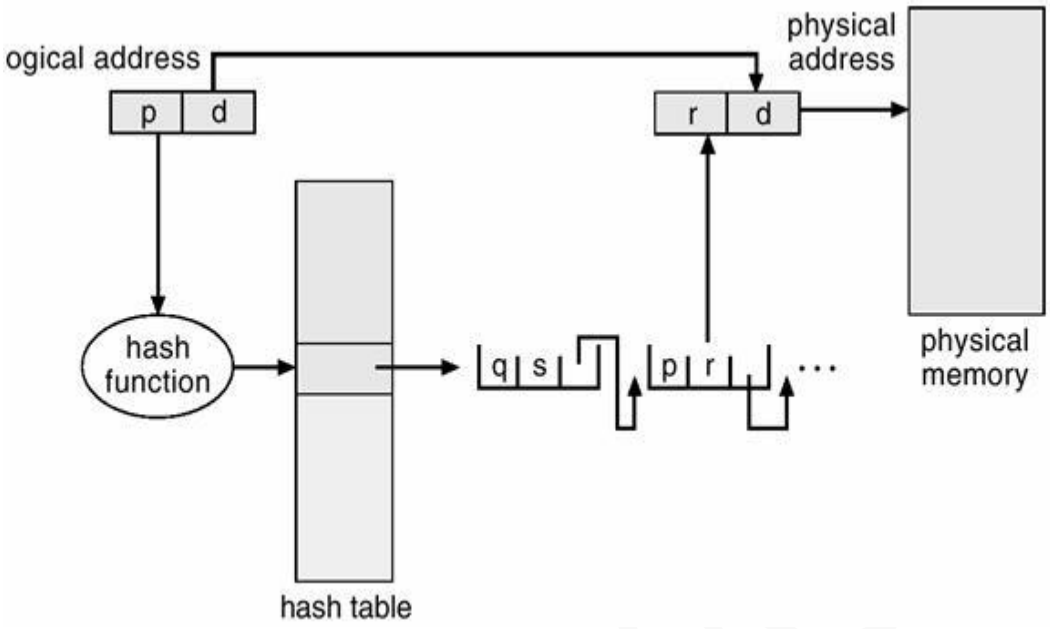
4.3.1.1.Lecture-4

Address-Translation Scheme

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



Hashed Page Table



Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Shared Pages

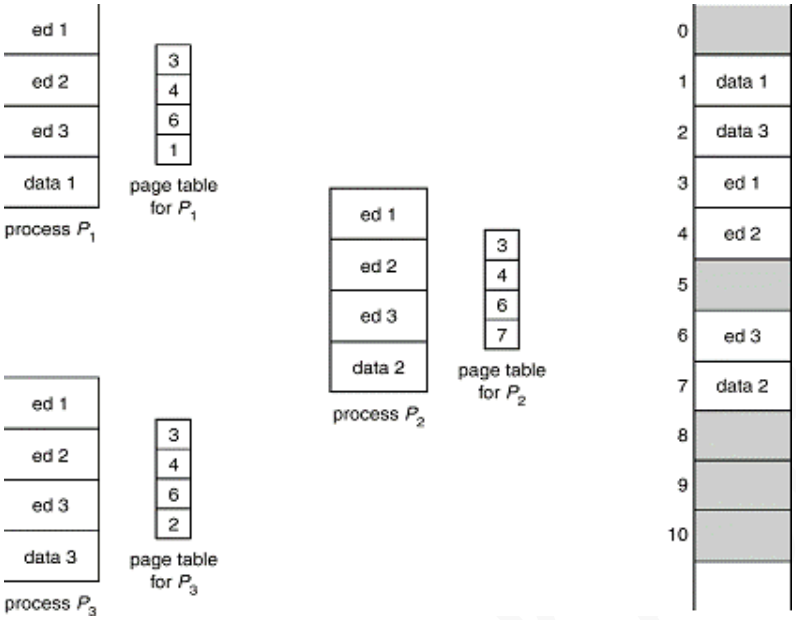
Shared code

1. One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

2. Shared code must appear in same location in the logical address space of all processes.

3. Each process keeps a separate copy of the code and data.

4.The pages for the private code and data can appear anywhere in the logical address space.

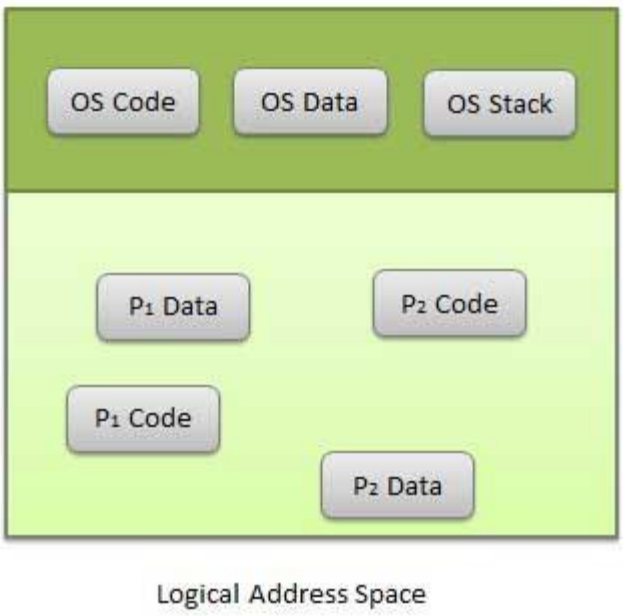


4.3.1.1.Lecture-5

Segmentation

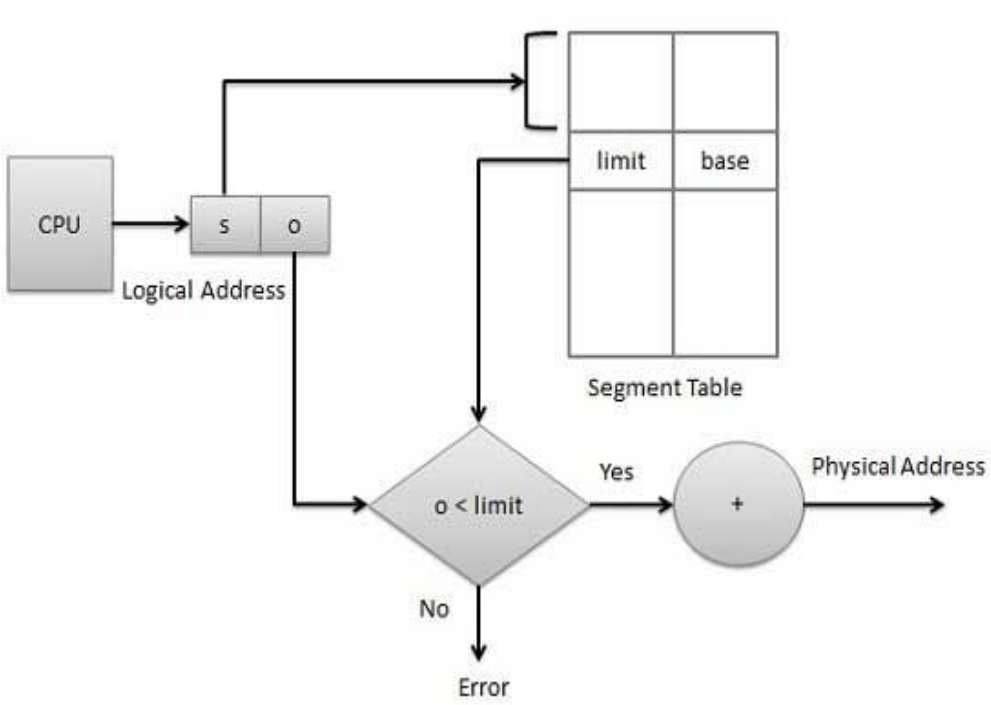
Segmentation is a technique to break memory into logical pieces where each piece represents a group of related information. For example ,data segments or code segment for each process, data segment for operating system and so on. Segmentation can be implemented using or without using paging.

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



Address generated by CPU is divided into

- **Segment number (s)** -- segment number is used as an index into a segment table which contains base address of each segment in physical memory and a limit of segment.
- **Segment offset (o)** -- segment offset is first checked against limit and then is combined with base address to define the physical memory address.



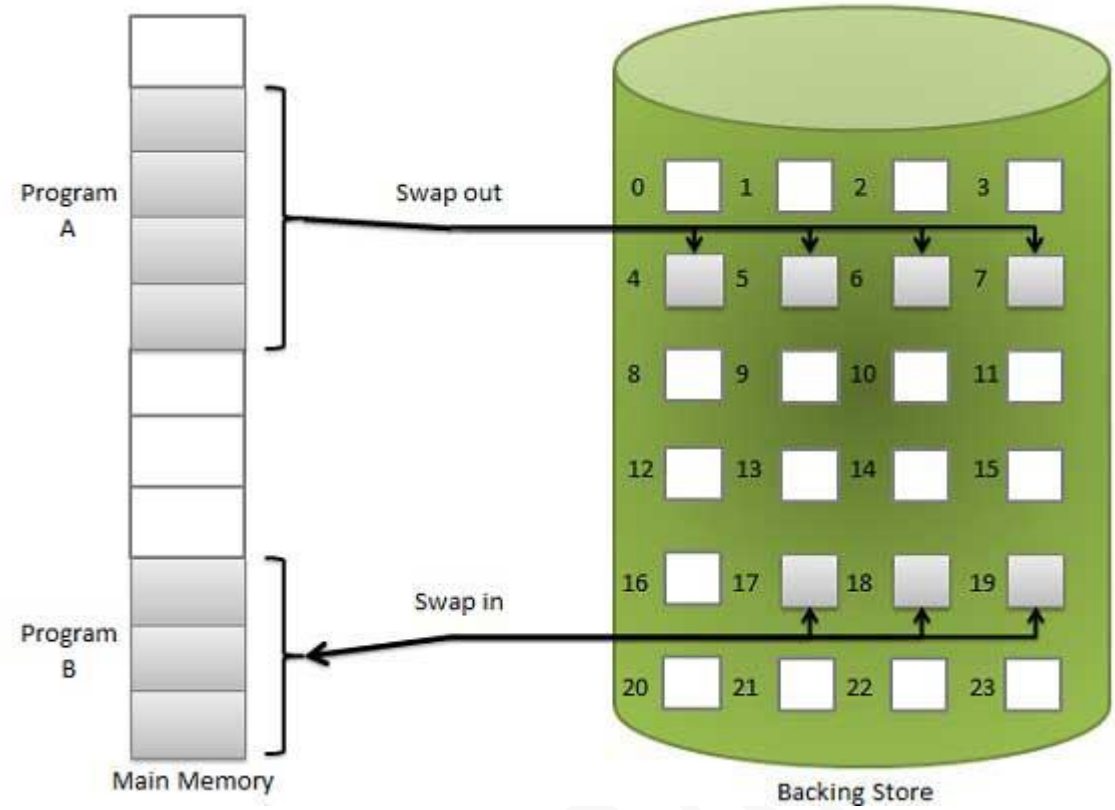
4.3.1.1.Lecture-6

Demand Paging

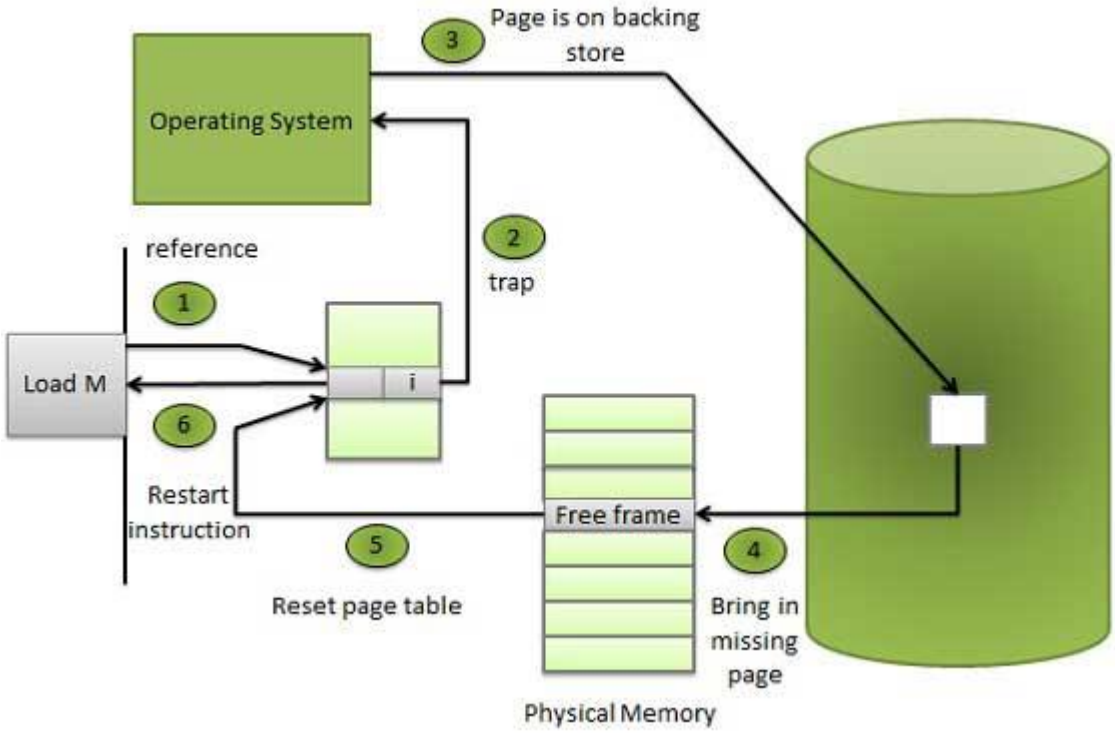
A demand paging system is quite similar to a paging system with swapping. When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper called pager.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used in anyway, decreasing the swap time and the amount of physical memory needed.

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



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



Step	Description

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

Advantages

Following are the advantages of Demand Paging

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

Disadvantages

Following are the disadvantages of Demand Paging

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

4.3.1.1Lecture-7

Page Replacement Algorithm

Page replacement algorithms are the techniques using which Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated. Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.

When the page that was selected for replacement and was paged out, is referenced again then it has to read in from disk, and this requires for I/O completion. This process determines the

quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm. A page replacement algorithm looks at the limited information about accessing the pages provided by hardware, and tries to select which pages should be replaced to minimize the total number of page misses, while balancing it with the costs of primary storage and processor time of the algorithm itself. There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults.

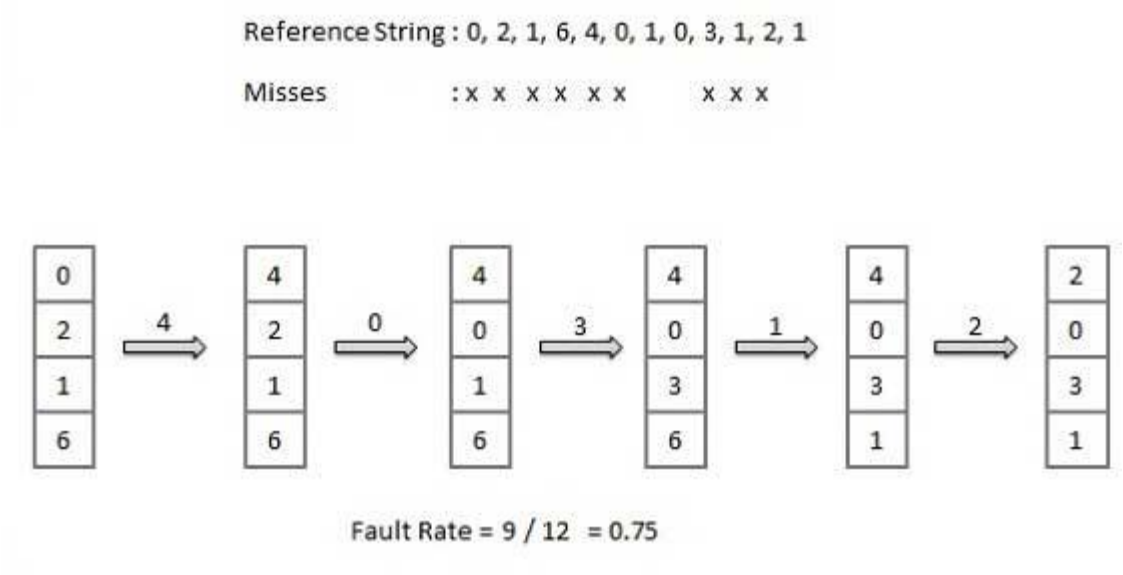
Reference String

The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data, where we note two things.

- For a given page size we need to consider only the page number, not the entire address.
- If we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.
- For example, consider the following sequence of addresses - 123,215,600,1234,76,96
- If page size is 100 then the reference string is 1,2,6,12,0,0

First in First Out (FIFO) algorithm

- Oldest page in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages from the tail and add new pages at the head.

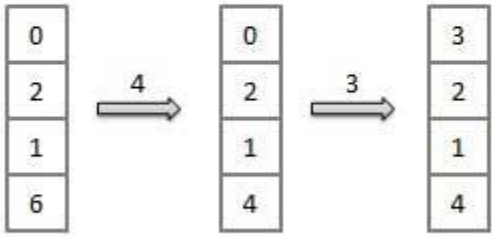


Optimal Page algorithm

- An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. An optimal page-replacement algorithm exists, and has been called OPT or MIN.
- Replace the page that will not be used for the longest period of time . Use the time when a page is to be used.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x



Fault Rate = 6 / 12 = 0.50

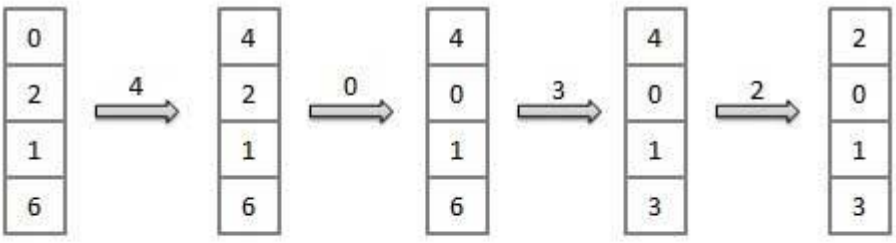
4.3.1.2 Lecture-8

Least Recently Used (LRU) algorithm

- Page which has not been used for the longest time in main memory is the one which will be selected for replacement.
- Easy to implement, keep a list, replace pages by looking back into time.

Reference String : 0, 2, 1, 6, 4, 0, 1, 0, 3, 1, 2, 1

Misses : x x x x x x x x



Fault Rate = 8 / 12 = 0.67

Page buffering algorithm

- To get process start quickly, keep a pool of free frames.
- On page fault, select a page to be replaced.
- Write new page in the frame of free pool, mark the page table and restart the process.

- Now write the dirty page out of disk and place the frame holding replaced page in free pool.

Least frequently Used (LFU) algorithm

- Page with the smallest count is the one which will be selected for replacement.
- This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

Most frequently Used (MFU) algorithm

- This algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

4.3.1.3 Lecture-9

Allocation of frames:

Priority Allocation:

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
- select for replacement one of its frames
- select for replacement a frame from a process with lower priority number

Global Replacement

Process selects a replacement frame from the set of all frames; one process can take a frame from another

Local Replacement

Each process selects from only its own set of allocated frames

Thrashing

If the process does not have number of frames it needs to support pages in active use, it will quickly page-fault. The high paging activity is called thrashing.

In other words we can say that when page fault ratio decreases below level, it is called thrashing.

4.3.1.4 Causes of Thrashing

It results in severe performance problems.

1) If CPU utilization is too low then we increase the degree of multiprogramming by introducing a new process to the system. A global page replacement algorithm is used. The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming.

2) CPU utilization is plotted against the degree of multiprogramming.

3) As the degree of multiprogramming increases, CPU utilization also increases.

4) If the degree of multiprogramming is increased further, thrashing sets in and CPU utilization drops sharply.

5) So, at this point, to increase CPU utilization and to stop thrashing, we must decrease the degree of multiprogramming.

To limit the effect of thrashing

To limit the effect of thrashing we can use **local replacement algorithm**. With Local replacement algorithm, if the process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well. The problem is not entirely solved.

Thus the effective access time will increase even for the process that is not thrashing