Operating System (23CY403)

II B.Tech II Semester (NR23)

Prepared By
Mrs. V V LAKSHMI SWATHI,
Assistant Professor,
Department of Cyber Security

UNIT-1

Introduction of Operating System

Operating system

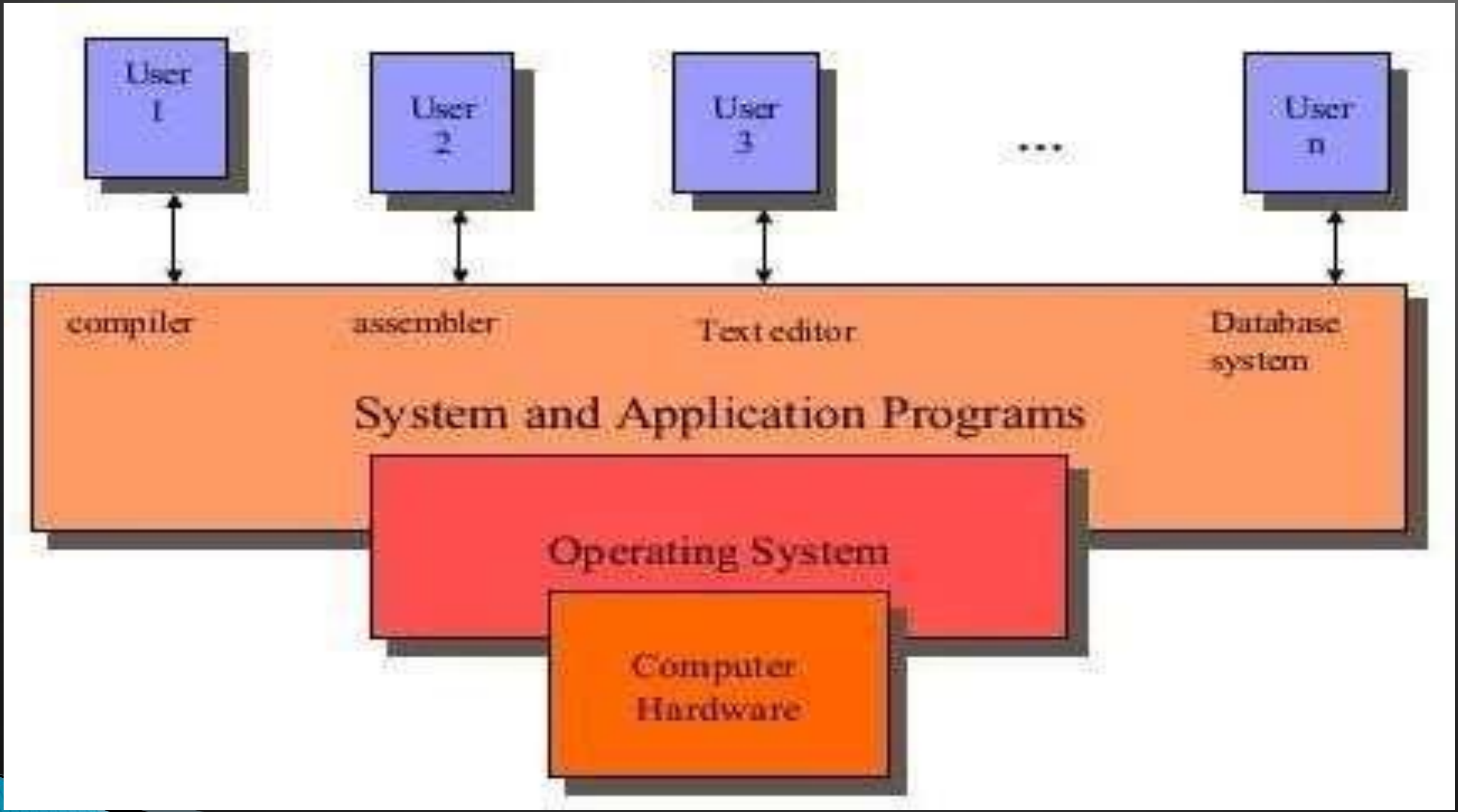
- An OS act as an interface between user and system hardware.

- Computer consists of the hardware, Operating System, system programs, application programs.

- The hardware consists of memory, CPU, ALU, I/O device, storage device and peripheral device.

- System program consists of compilers, loaders, editors, OS etc.

Operating System Structure

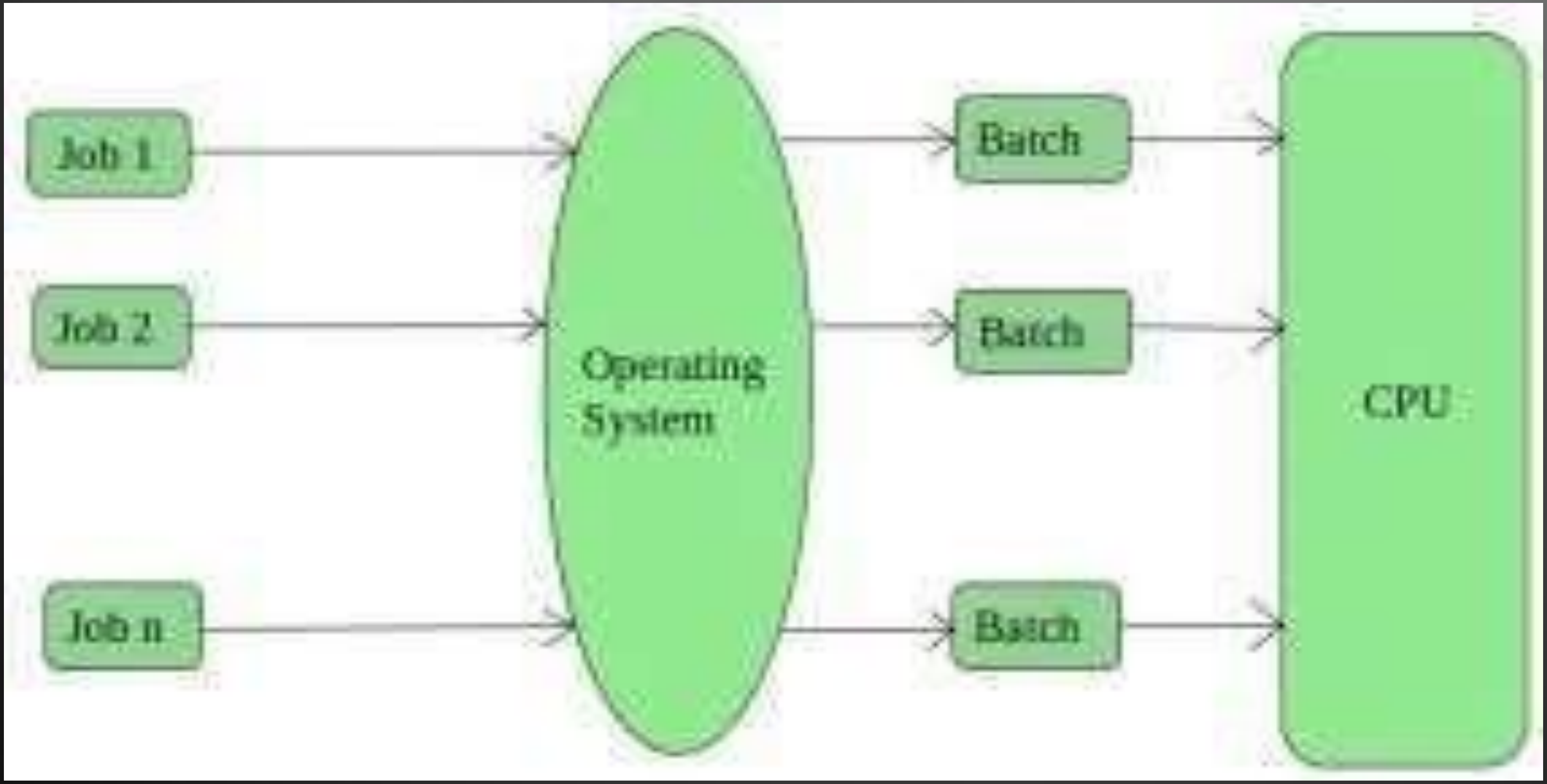


Batch Operating System

- This type of operating system does not interact with the computer directly.

- There is an operator which takes similar jobs having the same requirement and groups them into batches.
- It is the responsibility of the operator to sort jobs with similar needs.

Batch Operating System

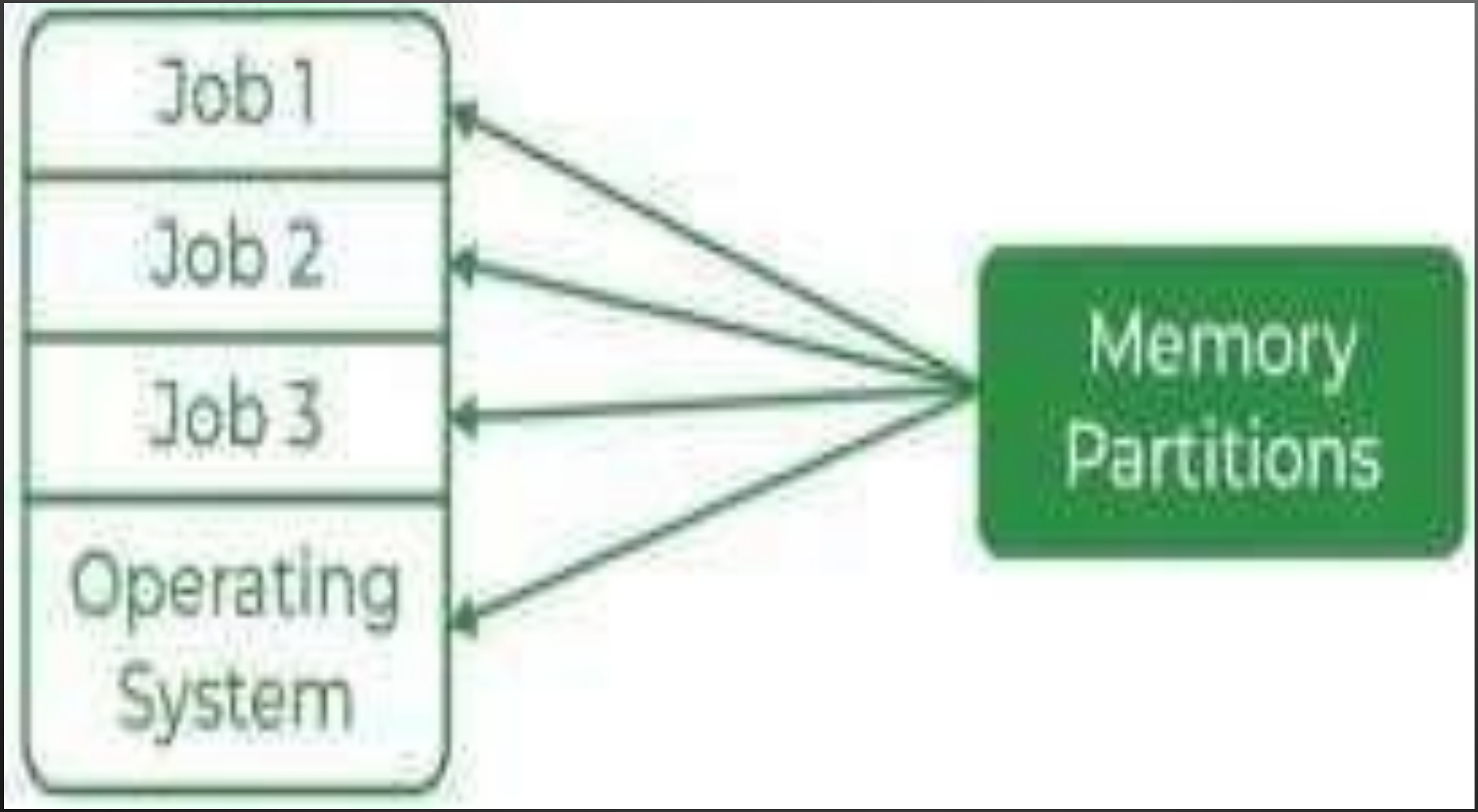


Multi–Programming Operating System

- Multiprogramming Operating Systems can be simply illustrated as more than one program is present in the main memory and any one of them can be kept in execution.

- This is basically used for better execution of resources.

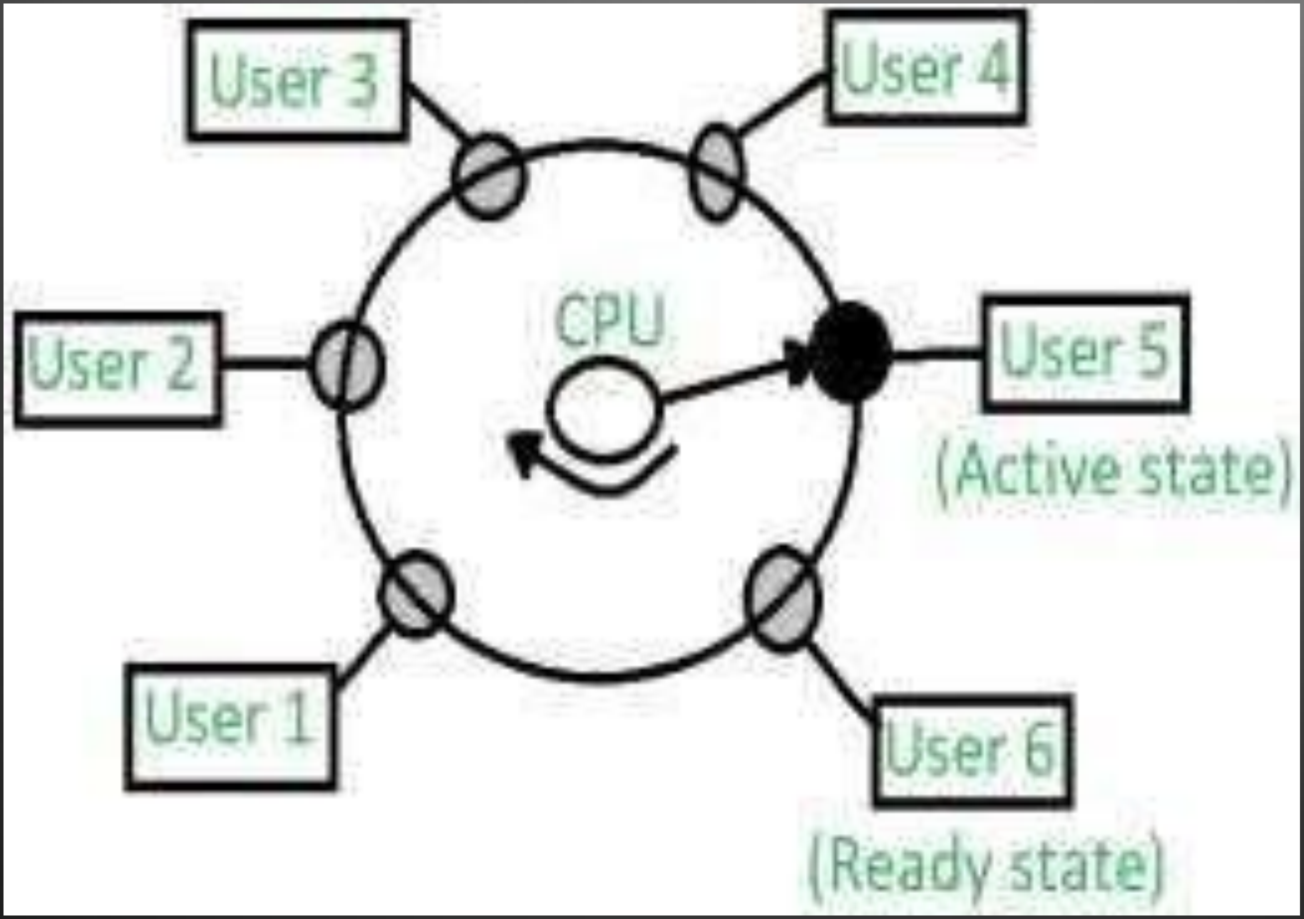
Multi-Programming Operating System



Time-Sharing Operating Systems

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

Time-Sharing Operating Systems



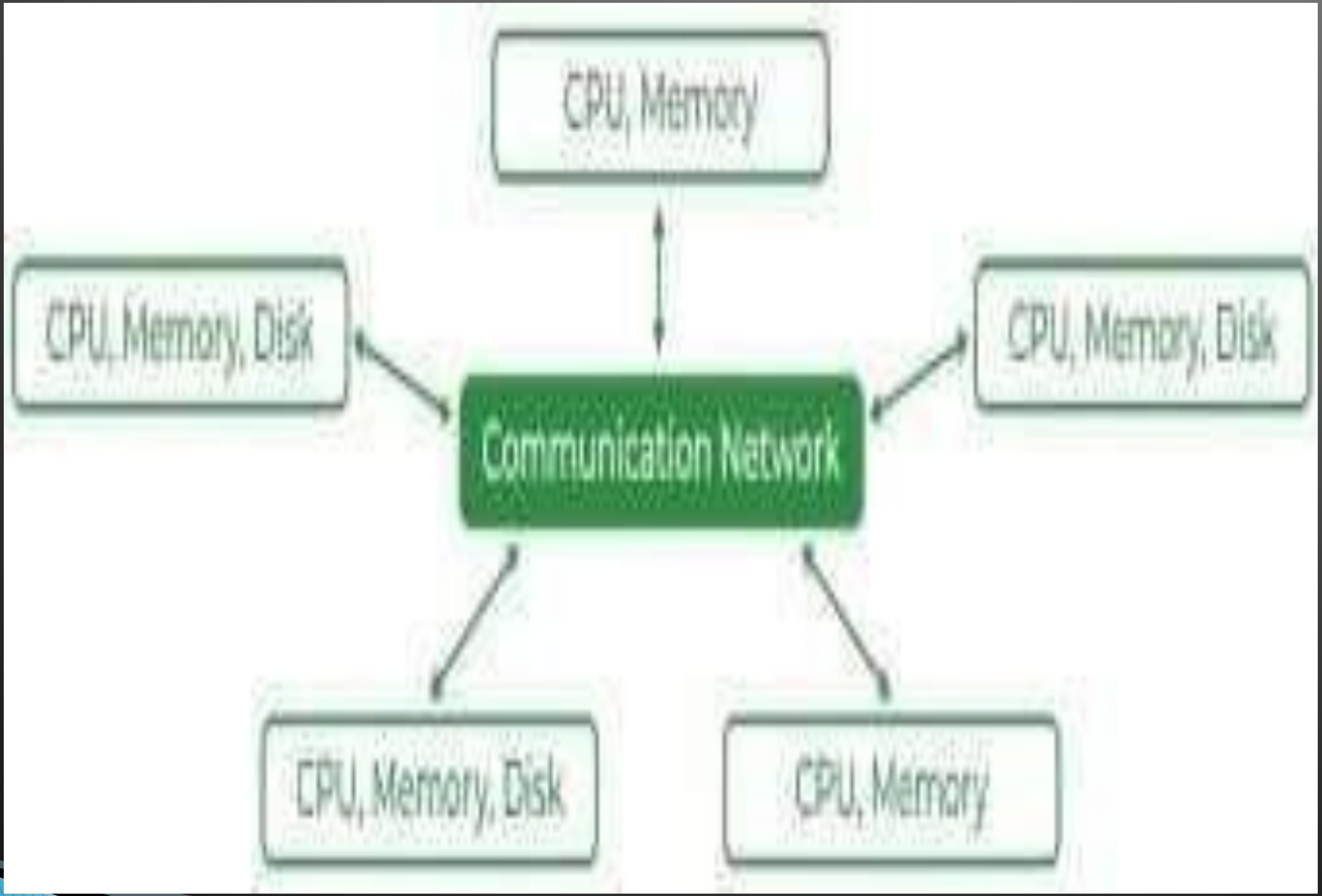
Personal Computer

- A personal computer (PC) is a microcomputer designed for use by one person at a time.
- Prior to the PC, computers were designed for -- and only affordable for -- companies that attached terminals for multiple users to a single large mainframe computer whose resources were shared among all users. By the
- word processing
- spreadsheets
- email
- instant messaging
- accounting

Distributed Operating System

- These types of operating systems are a recent advancement in the world of computer technology and are being widely accepted all over the world and, that too, at a great pace.
- Various autonomous interconnected computers communicate with each other using a shared communication network.
- Independent systems possess their own memory unit and CPU.

Distributed Operating System



Real-Time Operating System

- These types of OSs serve 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 that are very strict like missile systems, air traffic control systems, robots, etc.
- There are 2 types real time os.

Real-Time Operating System

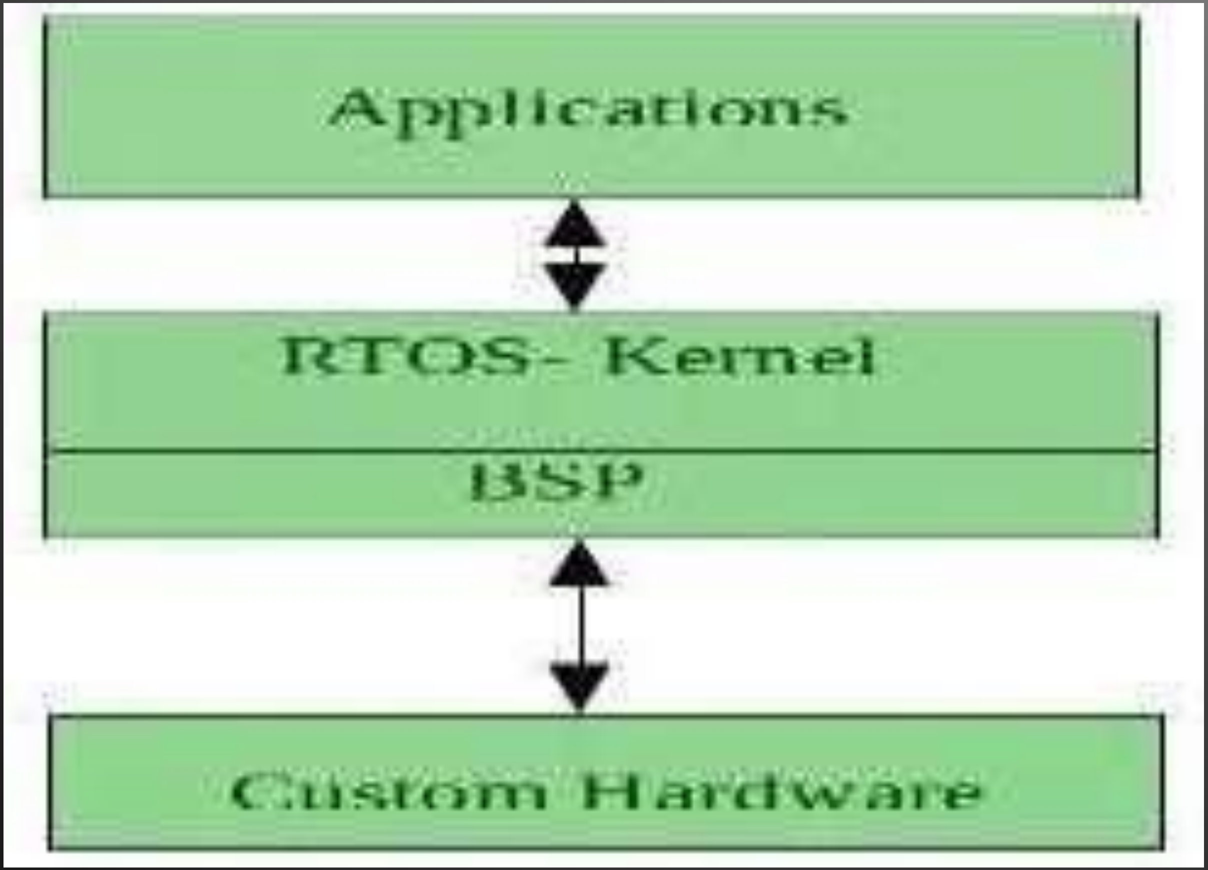
Hard Real-Time Systems

- ➢ Hard Real-Time OSs are meant for applications where time constraints are very strict and even the shortest possible delay is not acceptable.

Soft Real-Time Systems

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

Real-Time Operating System



Operating System Services

- **User Interface -** User interface is essential and all operating systems provide it.
- Users either interface with the operating system through command-line interface (CUI) or graphical user interface (GUI). Command interpreter executes next user-specified command.
- A GUI offers the user a mouse-based window and menu system as an interface.

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

- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device.

- **File-system manipulation** - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

- **Communications** – Processes may exchange information, on the same computer or between computers over a network. Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors may occur in the CPU and memory hardware, in I/O devices, in user program. For each type of error, OS should take the appropriate action to ensure correct and consistent computing. Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system.

System Calls

Types of System Calls

- There are commonly five types of system calls. These are as follows:
- **Process Control**
- **File Management**
- **Device Management**
- **Information Maintenance**
- **Communication Process Control**
- Process control is the system call that is used to direct the processes. Some process control examples include creating, load, abort, end, execute, process, terminate the process, etc.

- **File Management**
- File management is a system call that is used to handle the files. Some file management examples include creating files, delete files, open, close, read, write, etc.

- **Device Management**
- Device management is a system call that is used to deal with devices. Some examples of device management include read, device, write, get device attributes, release device, etc.
-
-

- **Information Maintenance**

- Information maintenance is a system call that is used to maintain information.
- There are some examples of information maintenance, including getting system data, set time or date, get time or date, set system data, etc.

- **Communication**

- Communication is a system call that is used for communication.
- There are some examples of communication, including create, delete communication connections, send, receive messages, etc.

System Components

- ProcessManagement
- FileManagement
- NetworkManagement
- MainMemoryManagement
- SecondaryStorageManagement
- I/ODeviceManagement
- SecurityManagement
- CommandInterpreterSystem

PROCESS

- A process can be thought of as a program in execution.
- A process is the unit of work in most systems.
- A process will need certain resources—such as CPU time, memory, files, and I/O devices to accomplish its task.

Process States

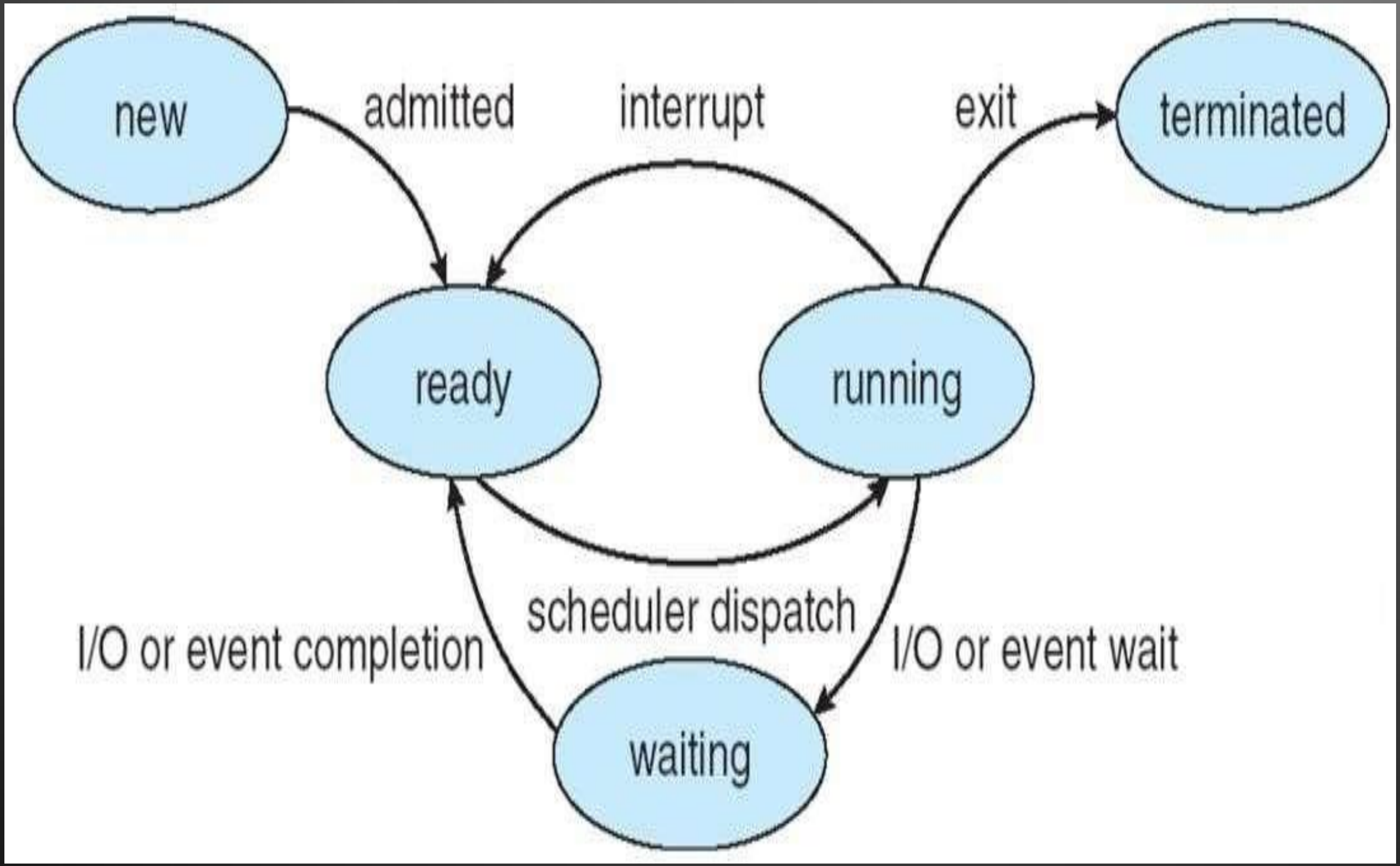
- As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process.
- A process may be in one of the following states:
    - **New:** The process is being created.
    - **Running:** Instructions are being executed.

- **Waiting**: The process is waiting for some event to occur (such as an 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.

Process state diagram



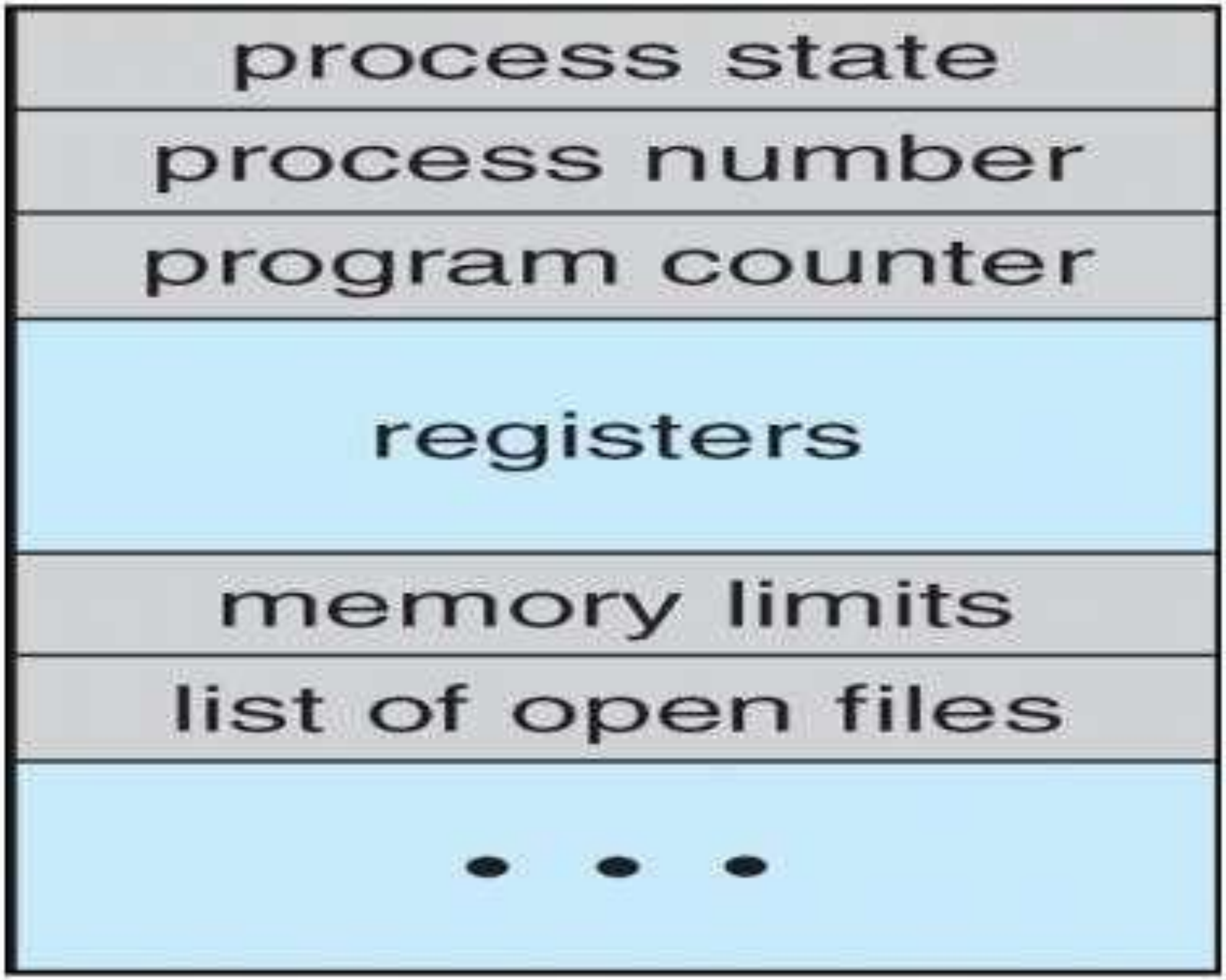
Process Control Block

- Each process is represented in the operating system by a **Process Control Block (PCB)** or **Task Control Block.**
- It contains many pieces of information associated with a specific process, including these:
- **Process state:** The state may be new, ready, running, and waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

- **CPU registers**. The registers vary in number ar type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general- purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

- **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
- **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.


process state
process number
program counter
registers
memory limits
list of open files
• • •

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

Operation On Process

- The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

- **Process Creation:** During the course of execution, a process may create several new processes.

- The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

- **fork():**

  - Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique **process identifier** (or **pid**), which is typically an integer number.

  - A new process is created by the fork () system call. The new process consists of a copy of the address space of the original process.

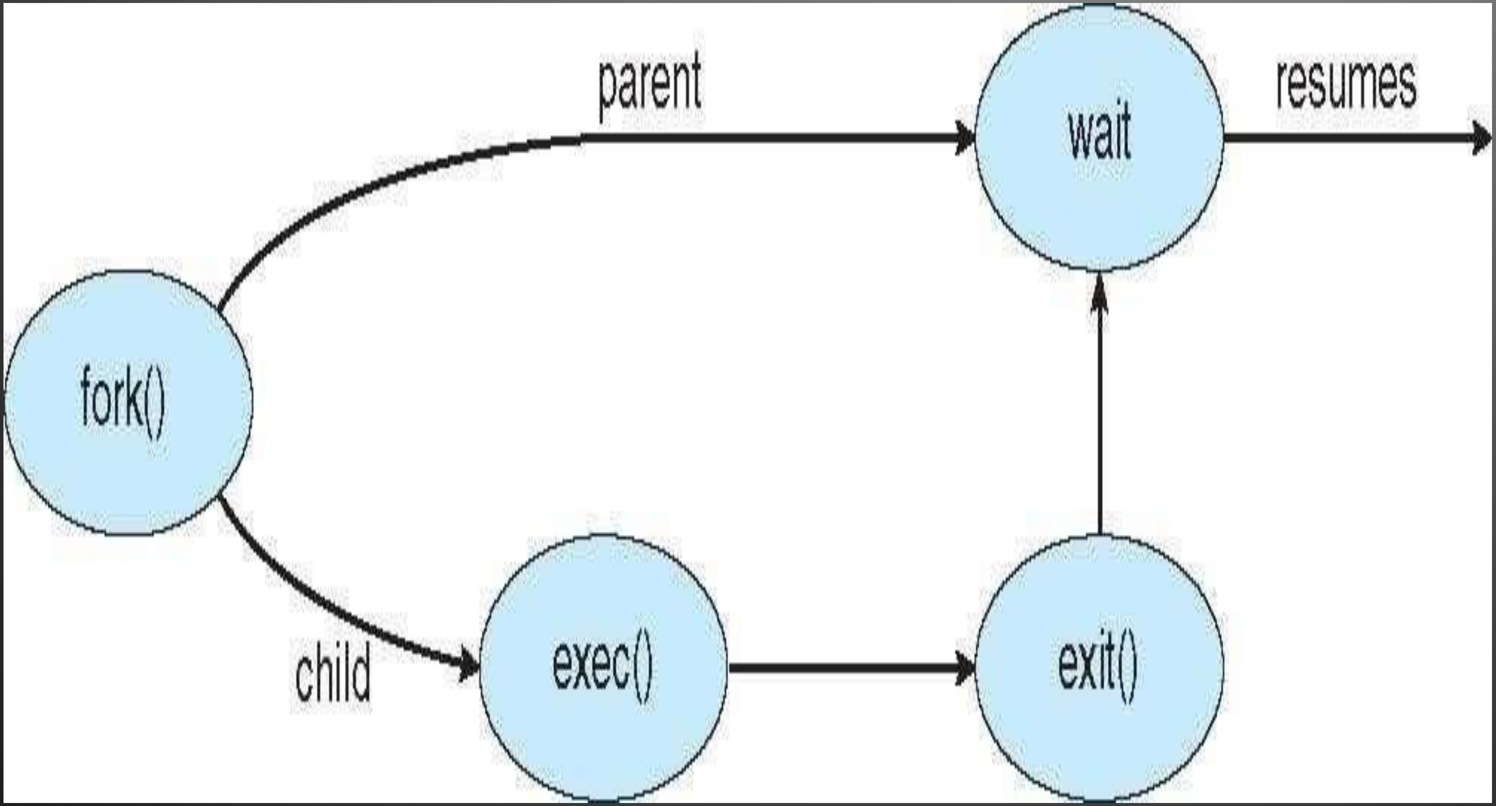
- **exec()**

  - After a fork () system call, one of the two processes typically uses the exec () system call to replace the process's memory space with a new program.

  - The exec () system call loads a binary file into memory and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.

- **wait()**
  - The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child.
  - Because the call to exec () overlays the process's address space with a new program, the call to exec () does not return control unless an error occurs.

Process Creation



Process Termination

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

- At that point, the process may return a status value (typically an integer) to its parent process (via the wait () system call).

- All the resources of the process—including physical and virtual memory, open files and I/O buffers—are de allocated by the operating system.
- Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, Terminate Process () in Windows).

Co-Operating Process

- Shared Memory
- Message passing

➢ The following figure shows a basic structure of communication between processes via the shared memory method and via the message passing method.

Shared Memory

➢ Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it.

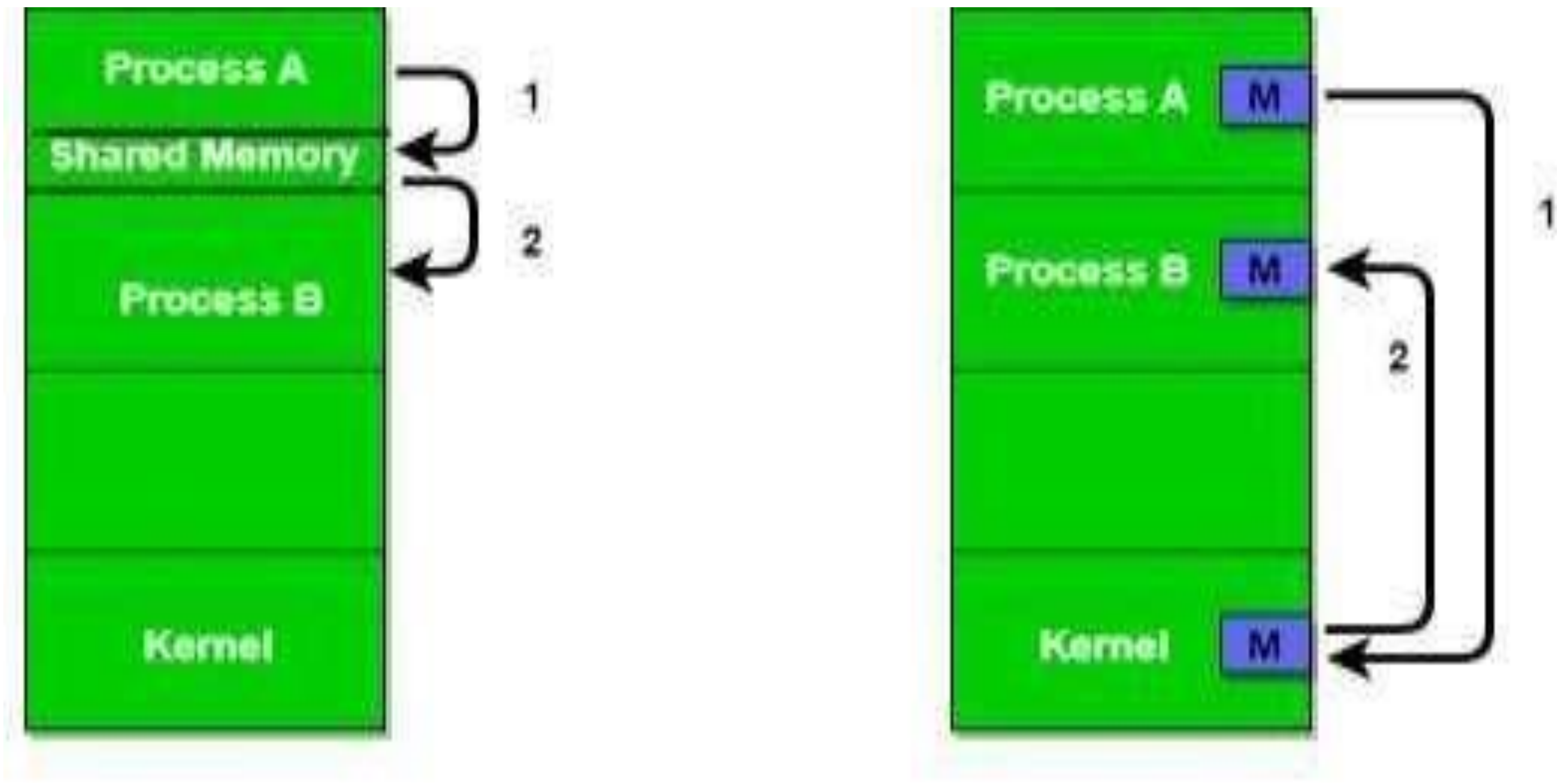
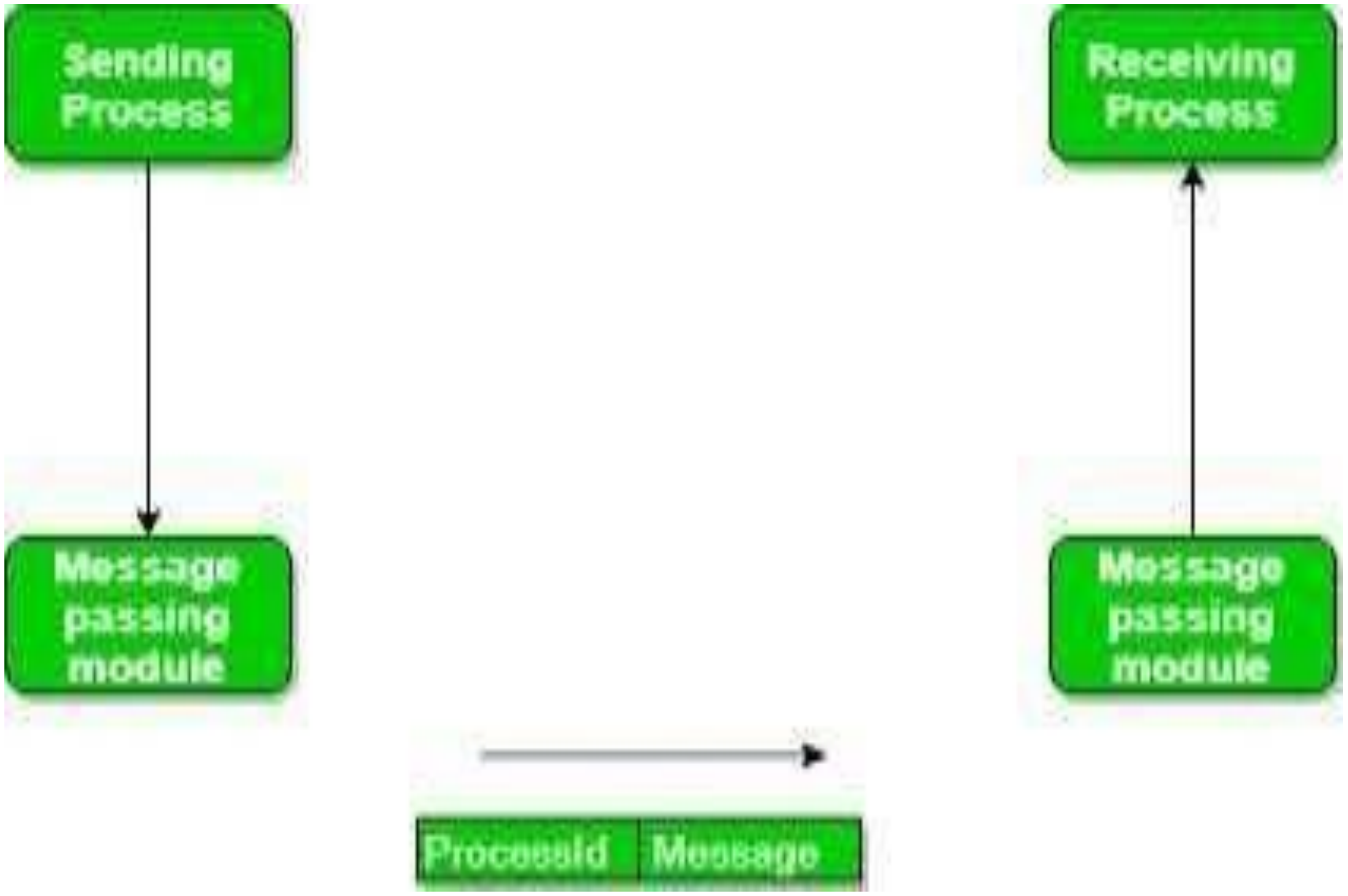


Figure 1 - Shared Memory and Message Passing

Messaging Passing Method

- In this method, processes communicate with each other without using any kind of shared memory.

- If two processes want to communicate with each other, they proceed as follows


Sending Process
Receiving Process
Message passing module
Message passing module
ProcessId
Message

Thread

- A thread is a Light weight process .Thread is a flow of control execution of the program.

- A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Single Thread

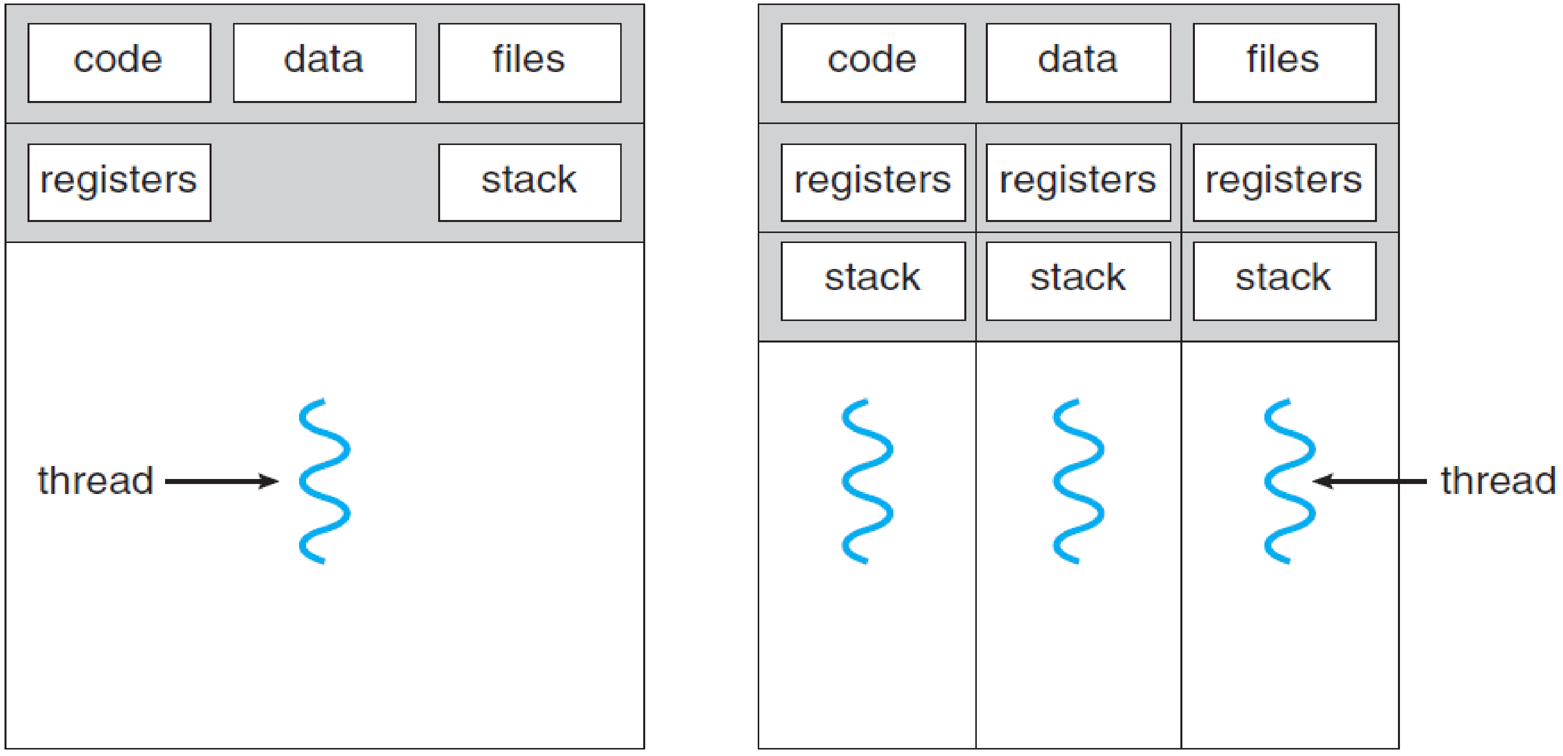
- A process is a program that performs a single **thread** of execution.

- For example, when a process is running a word-processor program, a single thread of instructions is being executed.

Multithreading Models

- Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.

- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

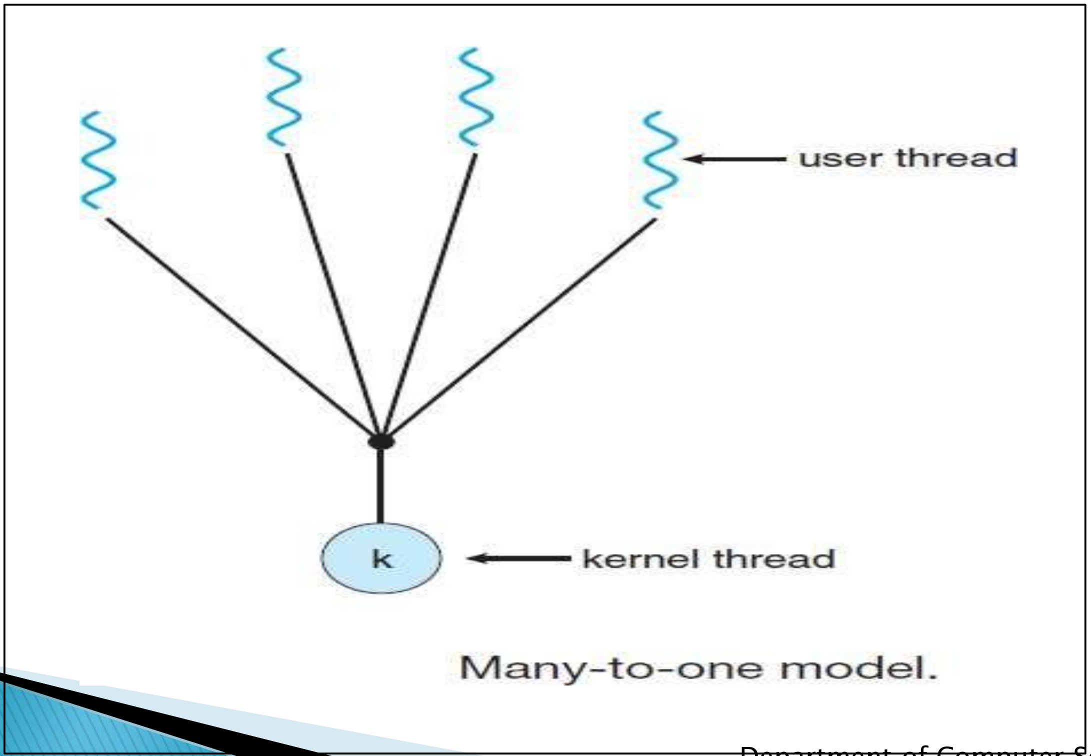


single-threaded process multithreaded process

Thread relationship

Many-to-One Model

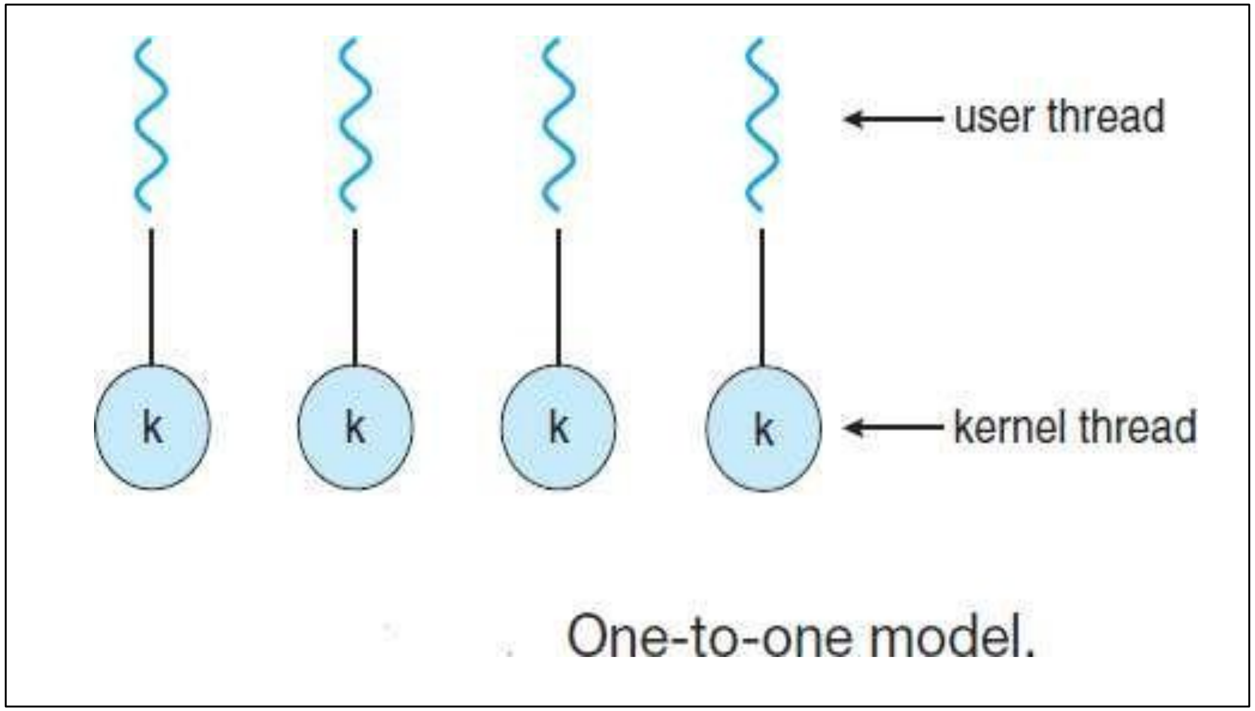
- The many-to-one model maps many user-level threads to one kernel thread.



Many-to-one model.

One-to-One Model

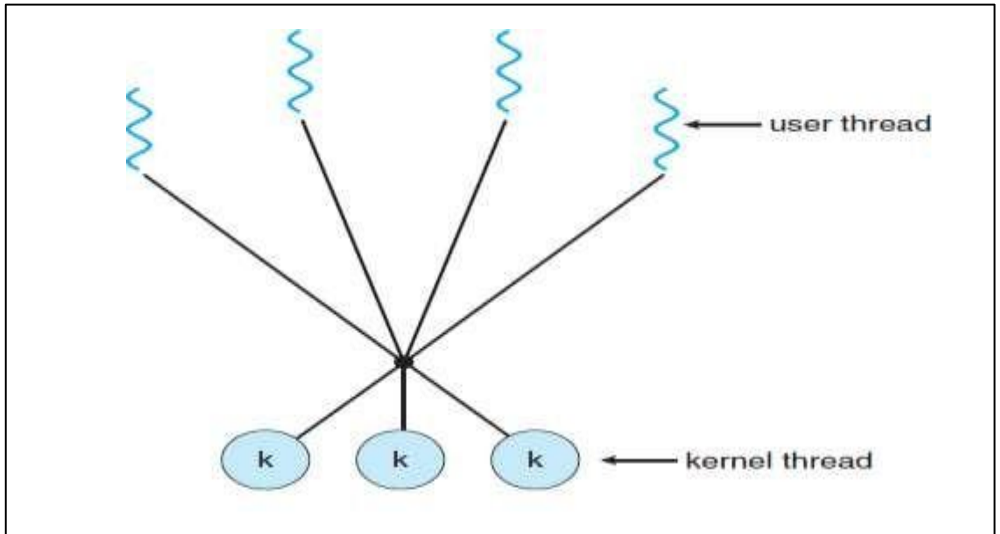
- The one-to-one model maps each user thread to a kernel thread.



One-to-one model.

Many-to-Many Model

- It multiplexes many user-level threads to a smaller or equal number of kernel threads.



Many-to-many model.

UNIT-2

UNIT-2 CPU scheduling

- **CPU scheduling** is the process of deciding which process will own the CPU to use while another process is suspended.

- The main function of the CPU scheduling is to ensure that whenever the CPU remains idle, the OS has at least selected one of the processes available in the ready-to-use line.

THE SCHEDULING CRITERIA

- **CPU utilization:**
- The main purpose of any CPU algorithm is to keep the CPU as busy as possible. Theoretically, CPU usage can range from 0 to 100 but in a real-time system, it varies from 40 to 90 percent depending on the system load.
- **Throughput:**
- The average CPU performance is the number of processes performed and completed during each unit. This is called throughput. The output may vary depending on the length or duration of the processes.
-

- **Turn round Time:**
- For a particular process, the important conditions are how long it takes to perform that process. The time elapsed from the time of process delivery to the time of completion is known as the conversion time. Conversion time is the amount of time spent waiting for memory access, waiting in line, using CPU, and waiting for I / O.
- **Waiting Time:**
- The Scheduling algorithm does not affect the time required to complete the process once it has started performing. It only affects the waiting time of the process i.e. the time spent in the waiting process in the ready queue.

Turn round Time:

- For a particular process, the important conditions are how long it takes to perform that process.
- The time elapsed from the time of process delivery to the time of completion is known as the conversion time.
- Conversion time is the amount of time spent waiting for memory access, waiting in line, using CPU, and waiting for I / O.

Waiting Time:

- The Scheduling algorithm does not affect the time required to complete the process once it has started performing.
- It only affects the waiting time of the process i.e. the time spent in the waiting process in the ready queue.

Response Time:

- In a collaborative system, turn around time is not the best option.

- The process may produce something early and continue to computing the new results while the previous results are released to the user.

- Therefore another method is the time taken in the submission of the application process until the first response is issued. This measure is called response time.

Types of CPU Scheduling Algorithms

- There are mainly two types of scheduling methods:

Preemptive Scheduling:

- Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to the ready state.

Non-Preemptive Scheduling:

- Non-Preemptive scheduling is used when a process terminates , or when a process switches from running state to waiting state.

First Come First Serve Scheduling:

- **FCFS** considered to be the simplest of all operating system scheduling algorithms.

- First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Shortest Job First (SJF) Scheduling:

- **Shortest job first (SJF)** is a scheduling process that selects the waiting process with the smallest execution time to execute next.

- This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.

Longest Job First(LJF) Scheduling:

- This is just opposite of shortest job first (SJF), as the name suggests this algorithm is based upon the fact that the process with the largest burst time is processed first.
- Longest Job First is non-preemptive in nature.

Priority Scheduling:

- **Preemptive Priority CPU Scheduling Algorithm** is a pre-emptive method of CPU scheduling algorithm that works **based on the priority** of a process.
- In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first.
- In the case of any conflict, that is, where there are more than one processor with equal value, then the most important CPU planning algorithm works on the basis of the FCFS **Characteristics**

Round Robin Scheduling:

- **Round Robin** is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot.

- It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Deadlock

- *A* ***deadlock*** is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

- Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other.

System model

- A system consists of a finite number of resources to be distributed among anumber of competing processes.

- The resources are partitioned into several types, each consisting of some number of identical instances

- **REQUEST**: The process requests the resource. If the request cannot be granted immediately(if the resource is being used by another process),then the requesting process must wait until it can acquire the resource.
- **USE**: The process can operate on the resource.
- If the resource is a printer, the process can print on the printer.
- **RELEASE**: The process releases the resource.

NECESSARY CONDITIONS FOR DEADLOCK

Mutual Exclusion

- Two or more resources are non-shareable (Only one process can use at a time)

Hold and Wait

- A process is holding at least one resource and waiting for resources.

No Pre-emption

- A resource cannot be taken from a process unless the process releases the resource.

Circular Wait

- A set of processes waiting for each other in circular form.

METHODS FOR HANDLING DEADLOCK

PREVENTION

- The idea is to not let the system into a deadlock state. This system will make sure that above mentioned four conditions will not arise.
- These techniques are very costly so we use this in cases where our priority is making a system deadlock-free.
- One can zoom into each category individually, Prevention is done by negating one of the four necessary conditions for deadlock.

Hold and Wait

- Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization.
- for example, if a process requires a printer at a later time and we have allocated a printer before the start of its execution printer will remain blocked till it has completed its execution.
- The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

AVOIDANCE

- Avoidance is kind of futuristic. By using the strategy of “Avoidance”, we have to make an assumption.

- We need to ensure that all information about resources that the process will need is known to us before the execution of the process.

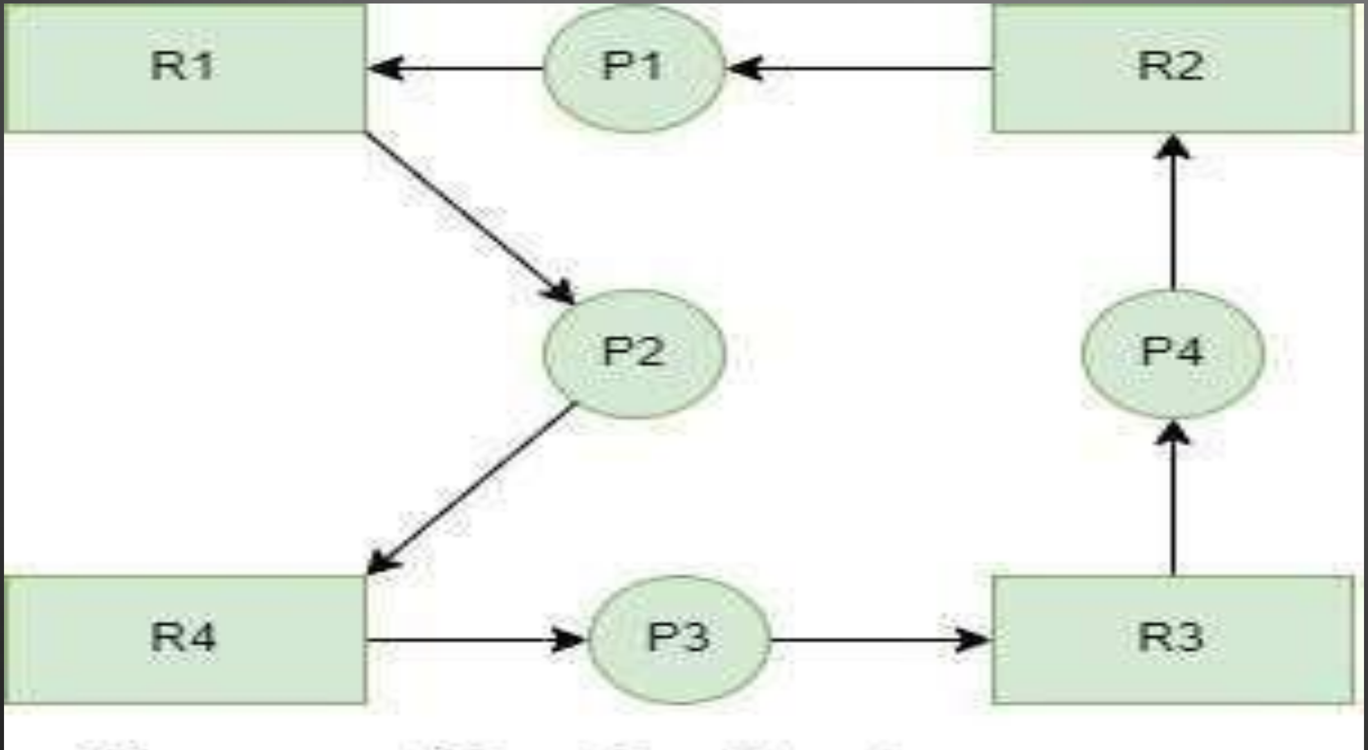
Resource Allocation Graph

- The resource allocation graph (RAG) is used to visualize the system‟s current state as a graph.

- The Graph includes all processes, the resources that are assigned to them, as well as the resources that each Process requests.

- Sometimes, if there are fewer processes, we can quickly spot a deadlock in the system by looking at the graph rather than the tables we use in Banker‟s algorithm.

Resource Allocation Graph



Resource Allocation Graph

Banker’s Algorithm

- Bankers‟s Algorithm is a resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources.
- It checks for the safe state, and after granting a request system remains in the safe state it allows the request, and if there is no safe state it doesn‟t allow the request made by the process.

- When working with a banker's algorithm, it requests to know about three things:

- How much each process can request for each resource in the system. It is denoted by the [**MAX**] request.

- How much each process is currently holding each resource in a system. It is denoted by the [**ALLOCATED**] resource.

- It represents the number of each resource currently available in the system. It is denoted by the [**AVAILABLE**] resource.

- Following are the important data structures terms applied in the banker's algorithm as follows:

- Suppose n is the number of processes, and m is the number of each type of resource used in a computer system.

- **Available**: It is an array of length 'm' that defines each type of resource available in the system. When Available[j] = K, means that 'K' instances of Resources type R[j] are available in the system.

- **Max:** It is a [n x m] matrix that indicates each process P[i] can store the maximum number of resources R[j] (each type) in a system.

- **Allocation:** It is a matrix of m x n orders that indicates the type of resources currently allocated to each process in the system. When Allocation [i, j] = K, it means that process P[i] is currently allocated K instances of Resources type R[j] in the system.

- **Need:** It is an M x N matrix sequence representing the number of remaining resources for each process.

- When the Need[i] [j] = k, then process P[i] may require K more instances of resources type Rj to complete the assigned work.

- Need[i][j] = Max[i][j] - Allocation[i][j].
- **Finish**: It is the vector of the order **m**. It includes a Boolean value (true/false) indicating whether the process has been allocated to the requested resources, and all resources have been released after finishing its task.

Deadlock detection and recovery

- If Deadlock prevention or avoidance is not applied to the software then we can handle this by deadlock detection and recovery, which consist of two phases.
- In the first phase, we examine the state of the process and check whether there is a deadlock or not in the system.
- If found deadlock in the first phase then we apply the algorithm for recovery of the deadlock.

Deadlock ignorance

- If a deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take. We use the ostrich algorithm for deadlock ignorance.

- In Deadlock, ignorance performance is better than the above two methods but not the correctness of data.

Safety Algorithm

- It is a safety algorithm used to check whether or not a system is in a safe state or follows the safe sequence in a banker's algorithm:

- **Step1:** There are two vectors **Wok** and **Finish** of length m and n in a safety algorithm. Initialize: Work = Available
- Finish[i] = false; for I = 0, 1, 2, 3, 4… n – 1.

Step2:

Check the availability status for each type of resources [i], such as: Need[i] <= Work

Finish[i] == false

If the i does not exist, go to step 4.

Step3: Work = Work +Allocation(i)

Step4: Finish[i] = true

Go to step2 to check the status of resource availability for the next process. If Finish[i] == true; it means that the system is safe for all processes.

Deadlock detection

- A deadlock detection algorithm is a technique used by an operating system to identify deadlocks in the system.

- This algorithm checks the status of processes and resources to determine whether any deadlock has occurred and takes appropriate actions to recover from the deadlock.
- The algorithm employs several times varying data structures:

- **Available –** A vector of length m indicates the number of available resources of each type.

- **Allocation** – An n*m matrix defines the number of resources of each type currently allocated to a process. The column represents resource and rows represent a process.

- **Request** – An n*m matrix indicates the current request of each process. If request[i][j] equals k then process P_i is requesting k more instances of resource type R_j.

RECOVERY FROM DEADLOCK

- The OS will use various recovery techniques to restore the system if it encounters any deadlocks.

- When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock.

Process Termination

- To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

Abort all the Deadlocked Processes:

- Aborting all the processes will certainly break the deadlock but at a great expense.
- The deadlocked processes may have been computed for a long time, and the result of those partial computations must be discarded and there is a probability of recalculating them later.

Abort one process at a time until the deadlock is eliminated:

- Abort one deadlocked process at a time, until the deadlock cycle is eliminated from the system.
- Due to this method, there may be considerable overhead, because, after aborting each process, we have to run a deadlock detection algorithm to check whether any processes are still deadlocked.

Resource Preemption

- To eliminate deadlocks using resource preemption, we preempt some resources from processes and give those resources to other processes.
- This method will raise three issues

Selecting a victim:

- We must determine which resources and which processes are to be preempted and also in order to minimize the cost.

Rollback:

- We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback.
- That means aborting the process and restarting it.

Starvation:

- In a system, it may happen that the same process is always picked as a victim.
- As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided.
- One solution is that a process must be picked as a victim only a finite number of times.

UNIT-3

UNIT-3 Synchronization

- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner.

- It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.

Types of Synchronization

- **Cooperative Process**: A process that can affect or be affected by other processes executing in the system

- **Independent Process**: The execution of one process does not affect the execution of other processes.

CRITICAL SECTION PROBLEM

- A critical section is a code segment that can be accessed by only one process at a time.

- The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables.

- So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

- In the entry section, the process requests for entry in the **Critical Section.**

- Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section.

- Then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can"t be postponed indefinitely.

Bounded Waiting:

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

PETERSON'S SOLUTION

- Peterson‟s Solution is a classical software-based solution to the critical section problem.
- In Peterson‟s solution, we have two shared variables:
- **boolean flag[i]**: Initialized to FALSE, initially no one is interested in entering the critical section
- **int turn**: The process whose turn is to enter the critical section.

```
// code for producer i
do
{
        flag[i] = true; turn = j;
        while (flag[j] == true && turn == j);
```

➢ **critical section**

```
        flag[i] = false;
        reminder section
                }while(TRUE);

                // code for consumer j
                do
                {
                flag[j] = true; turn = i;
                while (flag[i] == true && turn == i);
        critical section
        flag[i] = false;
        reminder section
        }while(TRUE);
```

SEMAPHOR ES

- Semaphore is a Hardware Solution. This Hardware solution is written or given to critical section problem. The Semaphore is just a normal integer.
- The Semaphore cannot be negative. The least value for a Semaphore is zero (0). The Maximum value of a Semaphore can be anything.
-
- The Semaphores usually have two operations. The two operations have the capability to decide the values of the semaphores.

- The two Semaphore Operations are:

1.Wait ( )

- The Wait operation works on the basis of Semaphore or Mutex Value.

- If the Semaphore value is greater than zero, then the Process can enter the Critical Section Area.

➢ Definition of wait()

```
wait(Semaphore S)
{
while (S<=0) ;      //no operation S--;
          }
```

2. Signal ( )

- The most important part is that this Signal Operation or V Function is executed only when the process comes out of the critical section.

- The value of semaphore cannot be incremented before the exit of process from the critical section.

Definition of signal()

```
signal(S)
{
        S++;
}
```

Two types of semaphores

Binary Semaphores:

- They can only be either 0 or 1. They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1.
- Then, a process has to wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section. When it completes its critical section, it can reset the value of the mutex semaphore to 0 and some other process can enter its critical section.

Counting Semaphores

- They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses.

- The semaphore can be initialized to the number of instances of the resource. Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available.

- Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1.

- After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

CLASSICAL PROBLEMS OF SYNCHRONIZATION

- The following problems of synchronization are considered as classical problems:

1. Bounded-buffer (or Producer-Consumer) Problem,
2. Dining-Philosophers Problem,
3. Readers and Writers Problem,

- Bounded-buffer (or Producer-Consumer) Problem
-
- Bounded Buffer problem is also called **producer consumer problem** and it is one of the classic problems of synchronization. This problem is generalized in terms of the producer consumer problems.

- Solution to this problem is, creating two counting semaphores “full” and “empty” to keep track of the current number of full and empty buffers respectively.

- Producers produce a product and consumers consume the product, but both use of one of the containers each time.

The Producer Operation

The Producer Operation

```
        do
    {
   wait(empty);              // wait until empty > 0 and
   then decrement 'empty' wait(mutex);    // acquire
   lock
   /* perform the insert operation in a slot */
   signal(mutex);   // release lock
   signal(full);       // increment 'full'


        } while(TRUE);
```

The Consumer Operation

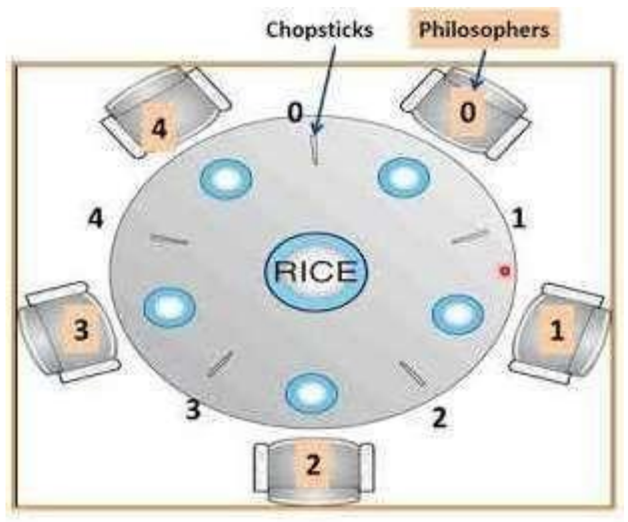
```
do
{
wait(full);                // wait until full > 0 and then decrement
'full' wait(mutex);  // acquire the lock

        /* perform the remove operation in a slot */

signal(mutex);        // release the lock signal(empty);     //
increment 'empty'

        } while(TRUE);
```

Dining-Philosophers problem

- The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers.

- There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him.

- One chopstick may be picked up by any one of its adjacent followers but not both.
- This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.



The Reader Process

- Reader requests the entry to critical section.
- If allowed:
- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the **wrt** semaphore to restrict the entry of writers if any reader is inside.

- do
- {
- wait(mutex); // Reader wants to enter the critical section readcnt++;
  // The number of readers has now increased by 1
-
- if (readcnt==1) // there is atleast one reader in the critical section
  wait(wrt); // no writer can enter if there is even one reader
-
- signal(mutex);// other readers can enter where otherer is inside
-
- ..... perform READING
-
- wait(mutex); // a reader wants to leave readcnt--;
-
- if (readcnt == 0) // no reader is left in the critical section,
  signal(wrt); // writers can enter

- signal(mutex);// reader leaves
-
- } while(true);
-

Writer Process

- **Writer process**
  - Writer requests the entry to critical section.
  - If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
  - It exits the critical section.

do
- {
- wait(wrt);   // writer requests for critical section
- ...perform WRITING
- signal(wrt);   // leaves the critical section
- } while(true);

MONITOR

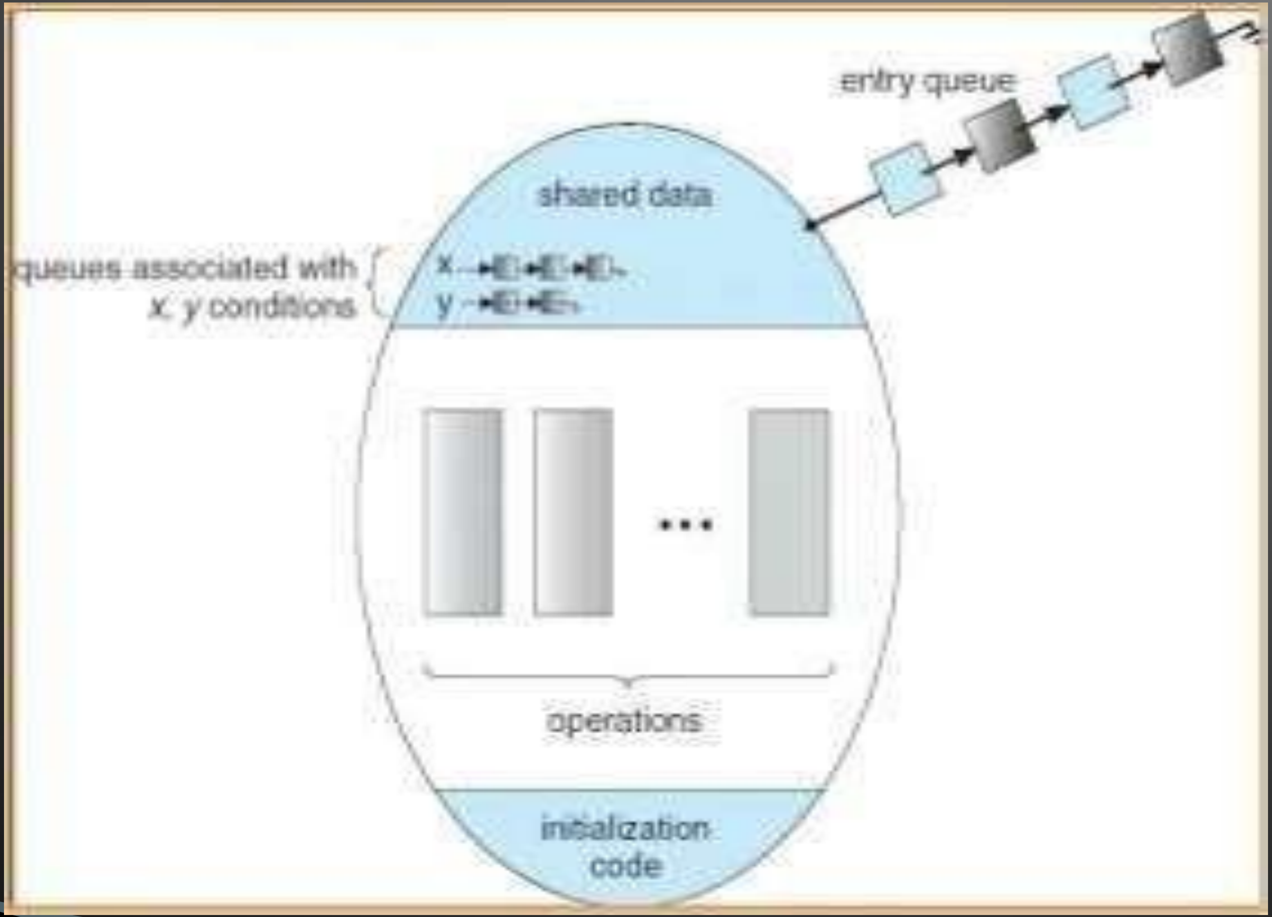
- It is a synchronization technique that enables threads to mutual exclusion and the **wait()** for a given condition to become true.

- It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable

- A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

- At any particular time, only one process may be active in a monitor. Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

Monitor


entry queue
shared data
queues associated with x, y conditions
x
y
...
operations
initialization code

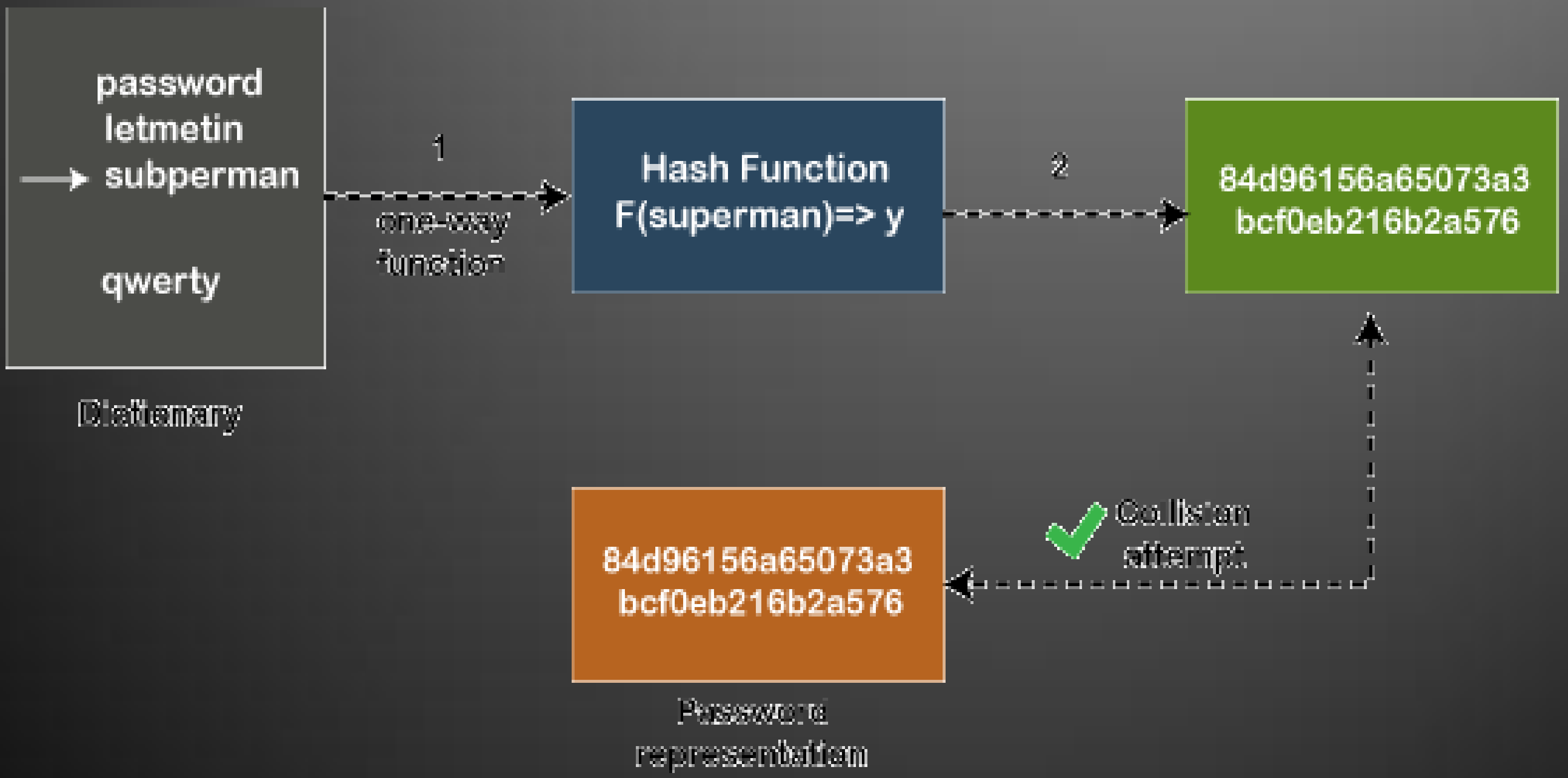
- **Syntax:**
- monitor
- {
- //shared variable declarations data variables;
- Procedure P1() { ... }
- Procedure P2() { ... }
- .
- .
- .
- Procedure Pn() { ... } Initialization Code() { ... }
- }
-

Inter Process Communication

- "Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)."

- To understand inter process communication, you can consider the following given diagram that illustrates the importance of inter-process communication

Role of Synchronization in Inter Process Communication



Synchronization methods

- These are the following methods that used to provide the synchronization:
- **Mutual Exclusion**
- **Semaphore**
- **Barrier**
- **Spinlock**
-

➢ **Mutual Exclusion:-**

➢ It is generally required that only one process thread can enter the critical section at a time. This also helps in synchronization and creates a stable state to avoid the race condition.

Semaphore:-

- Semaphore is a type of variable that usually controls the access to the shared resources by several processes.

- Semaphore is further divided into two types which are as follows:
- Binary Semaphore
- Counting Semaphore

Barrier:-

- A barrier typically not allows an individual process to proceed unless all the processes does not reach it. It is used by many parallel languages, and collective routines impose barriers.

Spinlock:-

- Spinlock is a type of lock as its name implies. The processes are trying to acquire the spinlock waits or stays in a loop while checking that the lock is available or not.

- It is known as busy waiting because even though the process active, the process does not perform any functional operation (or task).

Pipe

- The pipe is a type of data channel that is unidirectional in nature. It means that the data in this type of data channel can be moved in only a single direction at a time.

- Still, one can use two-channel of this type, so that he can able to send and receive data in two processes. Typically, it uses the standard methods for input and output.
- These pipes are used in all types of POSIX systems and in different versions of window operating systems as well.

Shared Memory

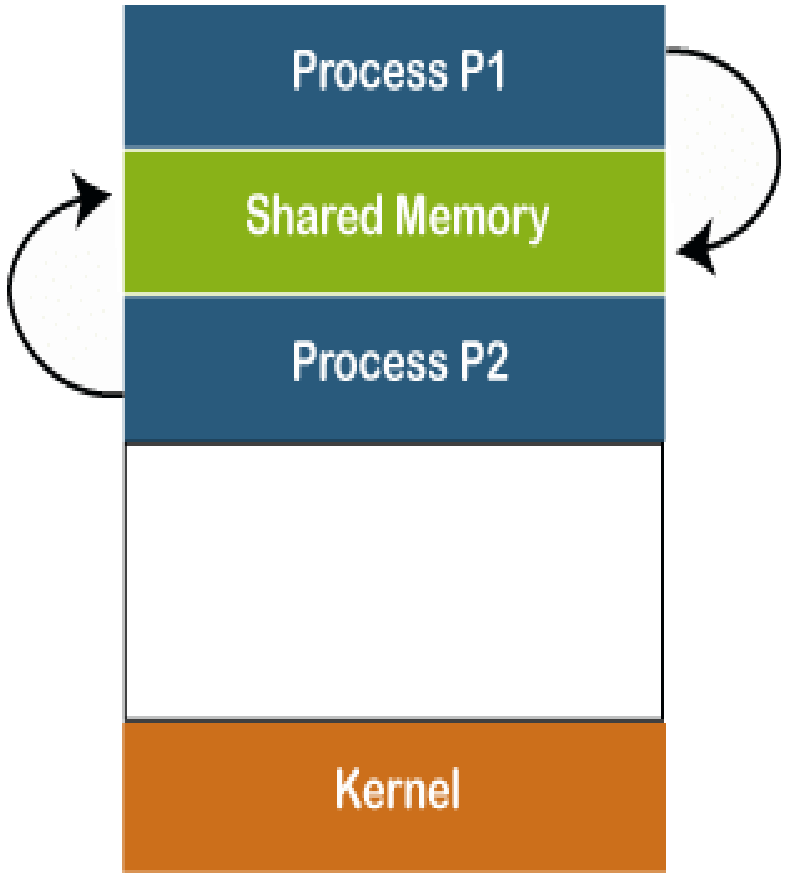
- It can be referred to as a type of memory that can be used or accessed by multiple processes simultaneously.

- It is primarily used so that the processes can communicate with each other. Therefore the shared memory is used by almost all POSIX and Windows operating systems as well.

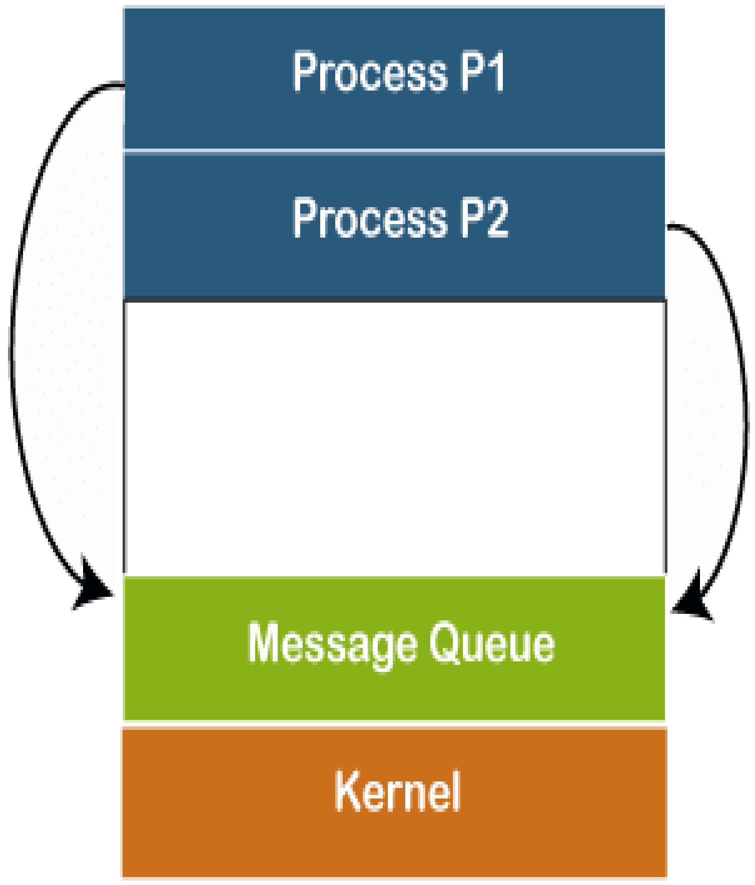
Message Queue

- In general, several different messages are allowed to read and write the data to the message queue.
- In the message queue, the messages are stored or stay in the queue unless their recipients retrieve them. In short, we can also say that the message queue is very helpful in inter-process communication and used by all operating systems.
- To understand the concept of Message queue and Shared memory in more detail.

Approaches to Interprocess Communication



Shared Memory



Message Queue

Message Passing

- It is a type of mechanism that allows processes to synchronize and communicate with each other.

- However, by using the message passing, the processes can communicate with each other without restoring the hared variables.

- Usually, the inter-process communication mechanism provides two operations that are as follows:

  send (message)
  received (message)

Direct Communication

- In this type of communication process, usually, a link is created or established between two communicating processes.

- However, in every pair of communicating processes, only one link can exist.

Indirect Communication

- Indirect communication can only exist or be established when processes share a common mailbox, and each pair of these processes shares multiple communication links

- . These shared links can be unidirectional or bi-directional.

FIFO

- It is a type of general communication between two unrelated processes. It can also be considered as full-duplex, which means that one process can communicate with another process and vice versa.

- Some other different approaches

  Socket

- It acts as a type of endpoint for receiving or sending the data in a network. It is correct for data sent between processes on the same computer or data sent between different computers on the same network. Hence, it used by several types of operating systems.

File

- A file is a type of data record or a document stored on the disk and can be acquired on demand by the file server. Another most important thing is that several processes can access that file as required or needed.

Signal

- As its name implies, they are a type of signal used in inter process communication in a minimal way. Typically, they are the massages of systems that are sent by one process to another.
- Therefore, they are not used for sending data but for remote commands between multiple processes.

UNIT-4

UNIT-4 Memory management

Memory Management

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

- A typical instruction-execution cycle, for example, first fetches an instruction from memory.

Logical Versus Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address or virtual address**.

- An 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**.

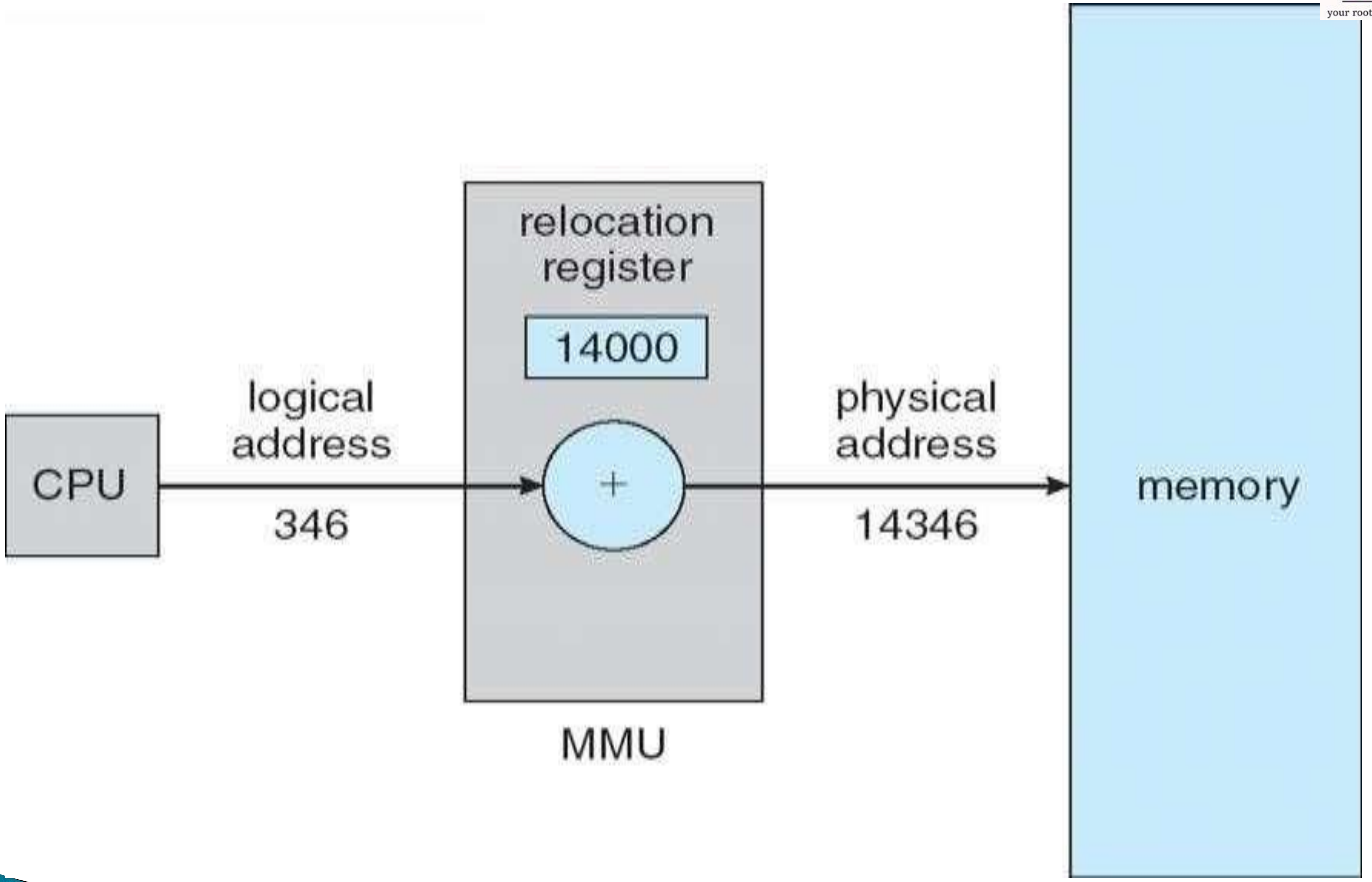
- The set of all logical addresses generated by a program is a **logical address space**.

- The set of all physical addresses corresponding to these logical addresses is a physical address space.

Memory-Management Unit (MMU)

- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.

- The base register is now called a **relocation register**. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.


CPU
logical address
346
relocation register
14000
+
MMU
physical address
14346
memory

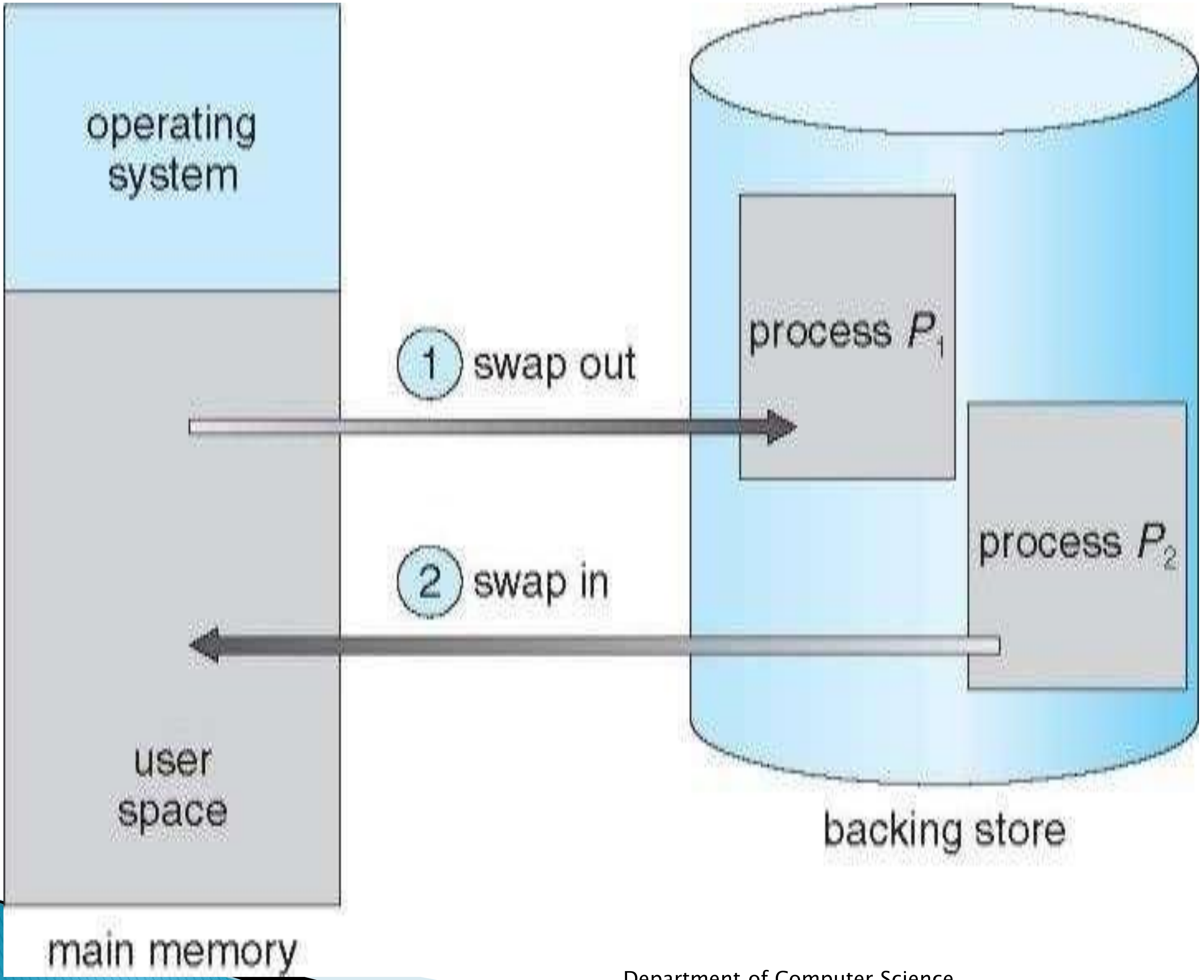
Swapping

- 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.
- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

Standard Swapping

- Standard swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk.

- It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.


operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store

Swapping on Mobile Systems

- Mobile systems typically do not support swapping in any form.

Reasons

- Mobile devices generally use flash memory rather than hard disks. The resulting space constraints avoid swapping.

- The limited number of writes that flash memory can tolerate before it becomes unreliable.

- The poor throughput between main memory ar flash memory in these devices.

Mechanisms instead of Swapping

- **Apple's iOS** *asks* applications to voluntarily relinquish allocated memory. Any applications that fail to free up sufficient memory may be terminated by the operating system.

- **Android** does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available.

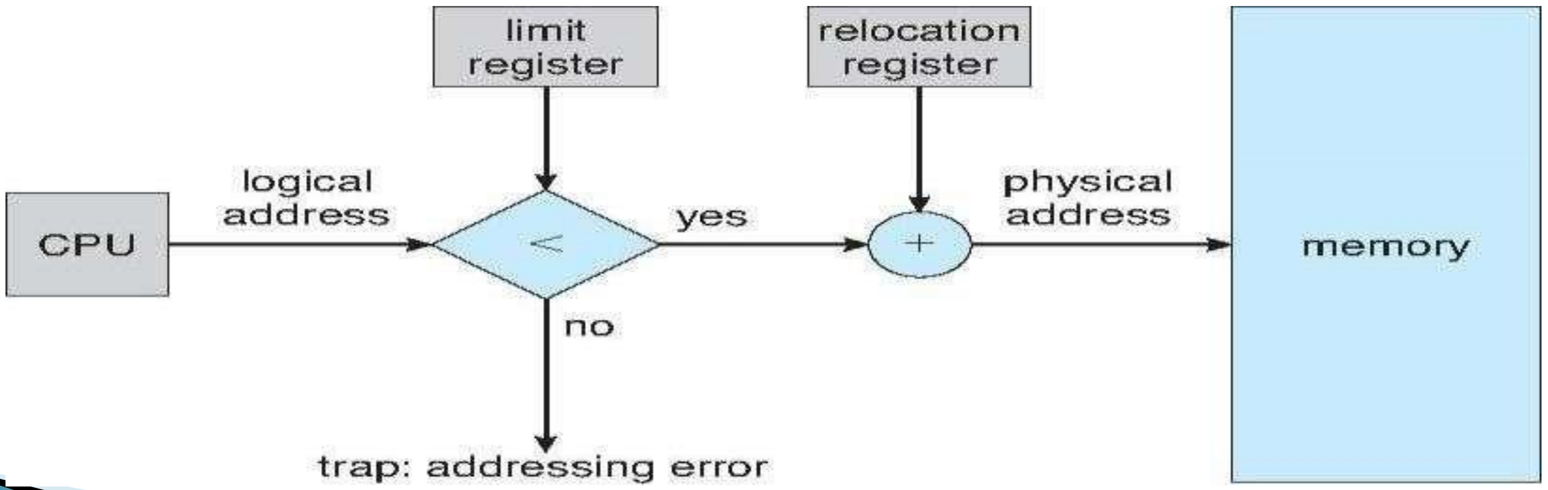
Contiguous Memory Allocation

- We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

- In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Memory Protection

- We can prevent a process from accessing memory it does not own by combining two ideas. If we have a system with a relocation register, together with a limit register, we accomplish our goal.



Memory allocation methods for memory allocation

Fixed-Sized Partitions

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized **partitions**.
- Each partition may contain exactly one process.
- In this **multiple partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.

Variable Sized -Partition

- In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.

- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

Dynamic Storage Allocation Problem (Memory Allocation Techniques)

- This concerns how to satisfy a request of size *n* from a list of free holes.

- There are many solutions to this problem.

- The **first-fit**, **best-fit**, and **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

First fit

- Allocate the first hole that is big enough.Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Best fit

- Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

Worst fit

- Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole.

- which may be more useful than the smaller leftover hole from a best-fit approach.

Fragmentation

- Memory fragmentation can be internal as well as external.
  - **Internal Fragmentation**
    - The overhead to keep track of this hole will be substantially larger than the hole itself.
    - The general approach to avoiding this problem is to break the physical memory into fixed-sized blocks and allocate memory in units based on block size.

- With this approach, the memory allocated to a process may be slightly larger than the requested memory.

External Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation**.

- As processes are loaded and removed from memory, the free memory space is broken into little pieces.

Segmentation

- Dealing with memory in terms of its physical properties is inconvenient to both the operating system and the programmer.
- What if the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory?
- The system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

Basic Method

- **Segmentation** is a memory-management scheme that supports the programmer view of memory.
- A logical address space is a collection of variable sized segments. Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities: a segment name and an offset.

Segmentation Hardware

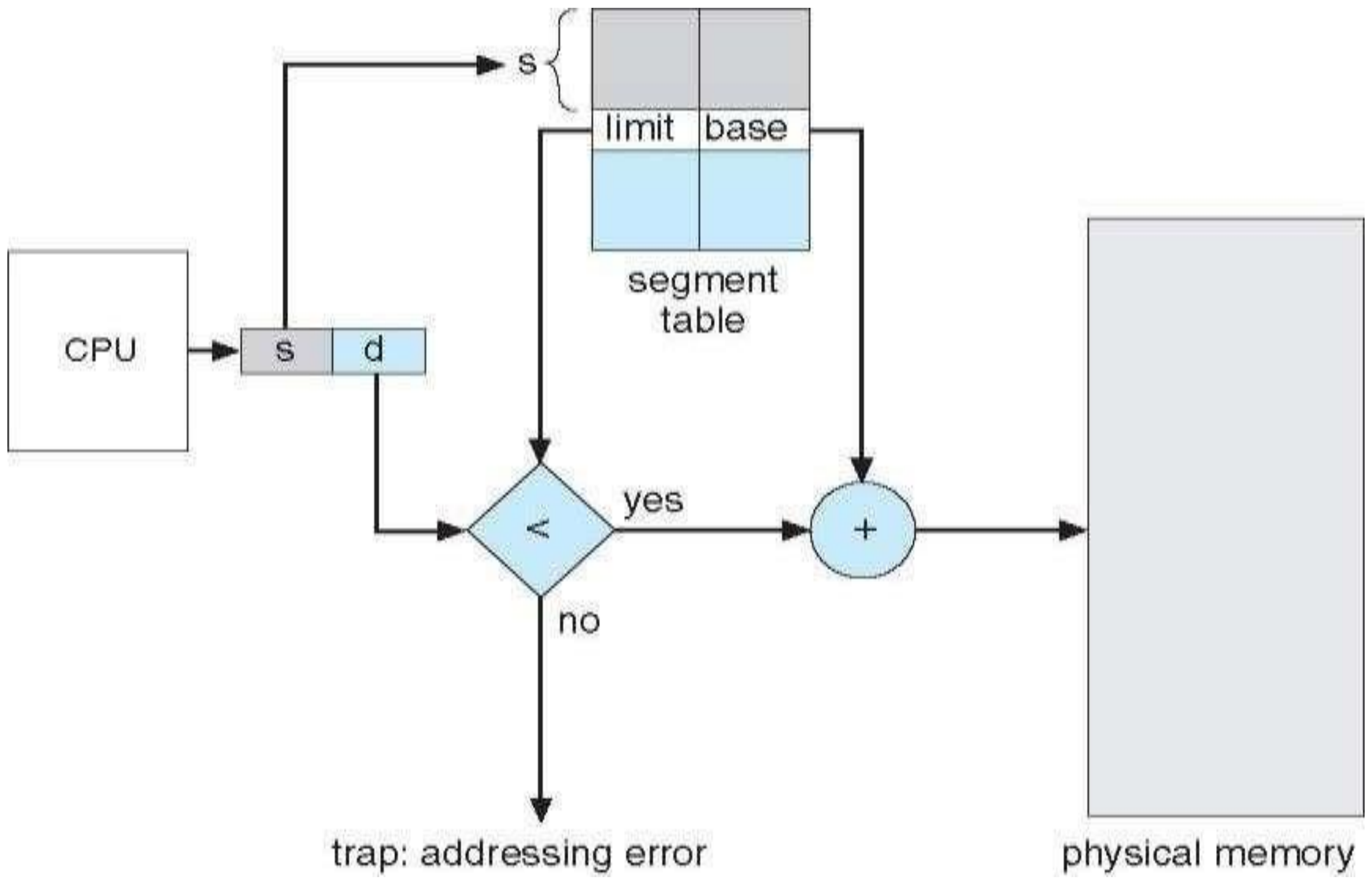
- Although the programmer can now refer to objects in the program by a two- dimensional address, the actual physical memory is still, of course, a one dimensional sequence of bytes.

- Thus, we must define an implementation to map two-dimensional user- defined addresses into one-dimensional physical addresses. This mapping is affected by a **segment table.**

- Each entry in the segment table has a **segment base** and a **segment limit**.

- **Segment base:** The segment base contains the starting physical address where the segment resides in memory.

- **Segment limit**: The segment limit specifies the length of the segment.


s
limit
base
segment table
CPU
s
d
<
yes
no
+
trap: addressing error
physical memory

Paging

- **Paging** is another memory-management scheme that offers physical address space of a process to be non-contiguous.

- Paging also avoids external fragmentation and the need for compaction, whereas segmentation does not. Because of its advantages, paging in its various forms is used in most operating systems, from mainframes to smart phones.

Basic Method of Paging

- **Frames**: Paging involves breaking physical memory into fixed-sized blocks called

frames

- **Pages:** Breaking logical memory into blocks of the same size called **pages**.

- When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).

Hardware Support for Paging

- Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**.

  Page Table

- The page number is used as an index into a **page table**.

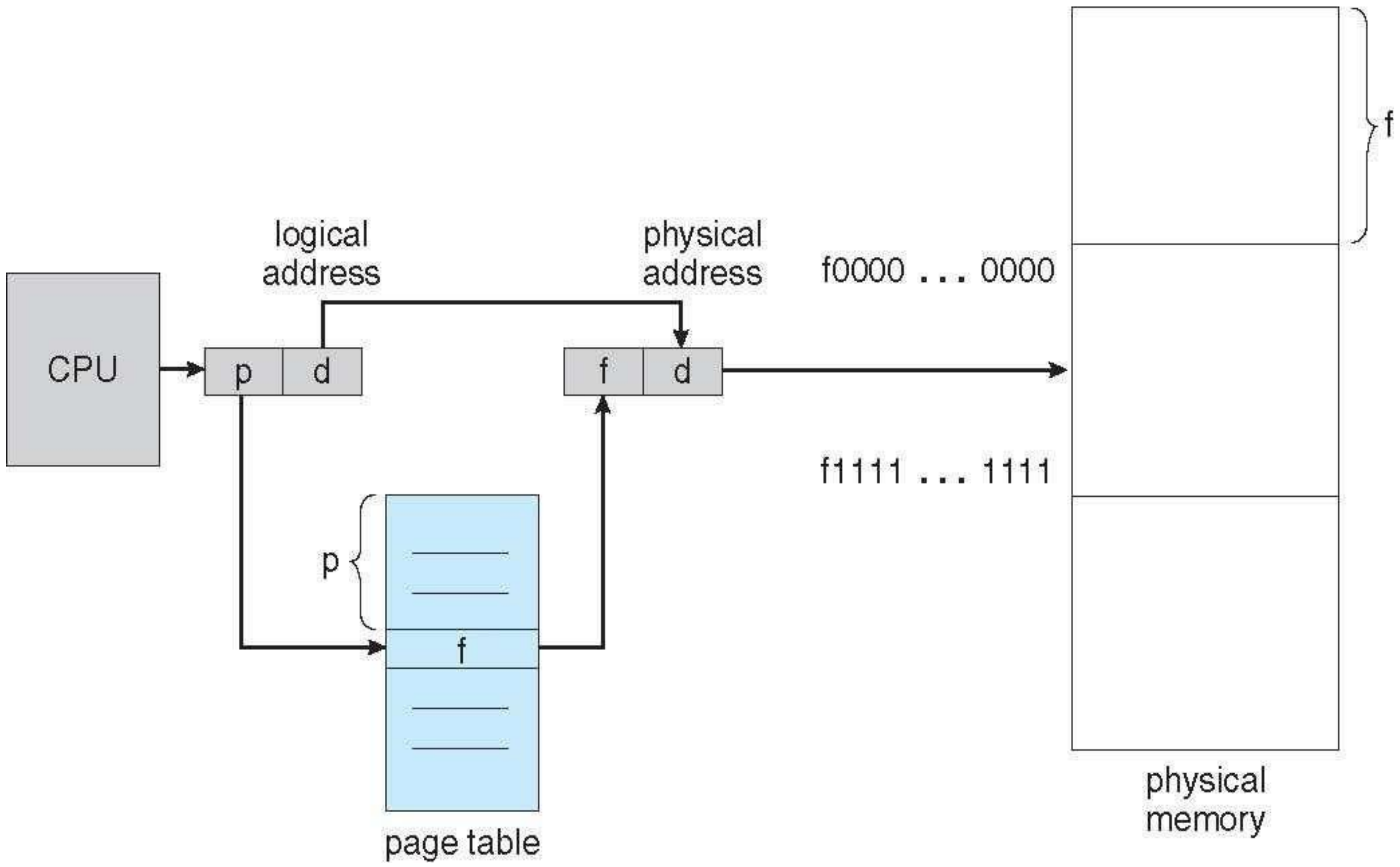
- The page table contains the base address of each page in physical memory.

- This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

Frame Table

- **Frame Table:** Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory.

- which frames are allocated, which frames are available, how many total frames there are, and so on? This information is generally kept in a data structure called a **frame table**.


logical address
physical address
CPU
p
d
f
d
f0000 ... 0000
f1111 ... 1111
f
p
f
page table
physical memory

Defining of Page Size

- The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page.

- Depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

- If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high- order $m- n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset.

- Thus, the logical address is as follows:
- page number-P
- page offset-d

- where *p* is an index into the page table and *d* is the displacement within the page.

Hardware Support

- **Methods for storing page table:** Each operating system has its own methods for storing page tables.
- Some allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block.
- When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

- Other operating systems provide one or at most a few page tables, which decreases the overhead involved when processes are context-switched.

- **Page-Table Base Register (PTBR)**

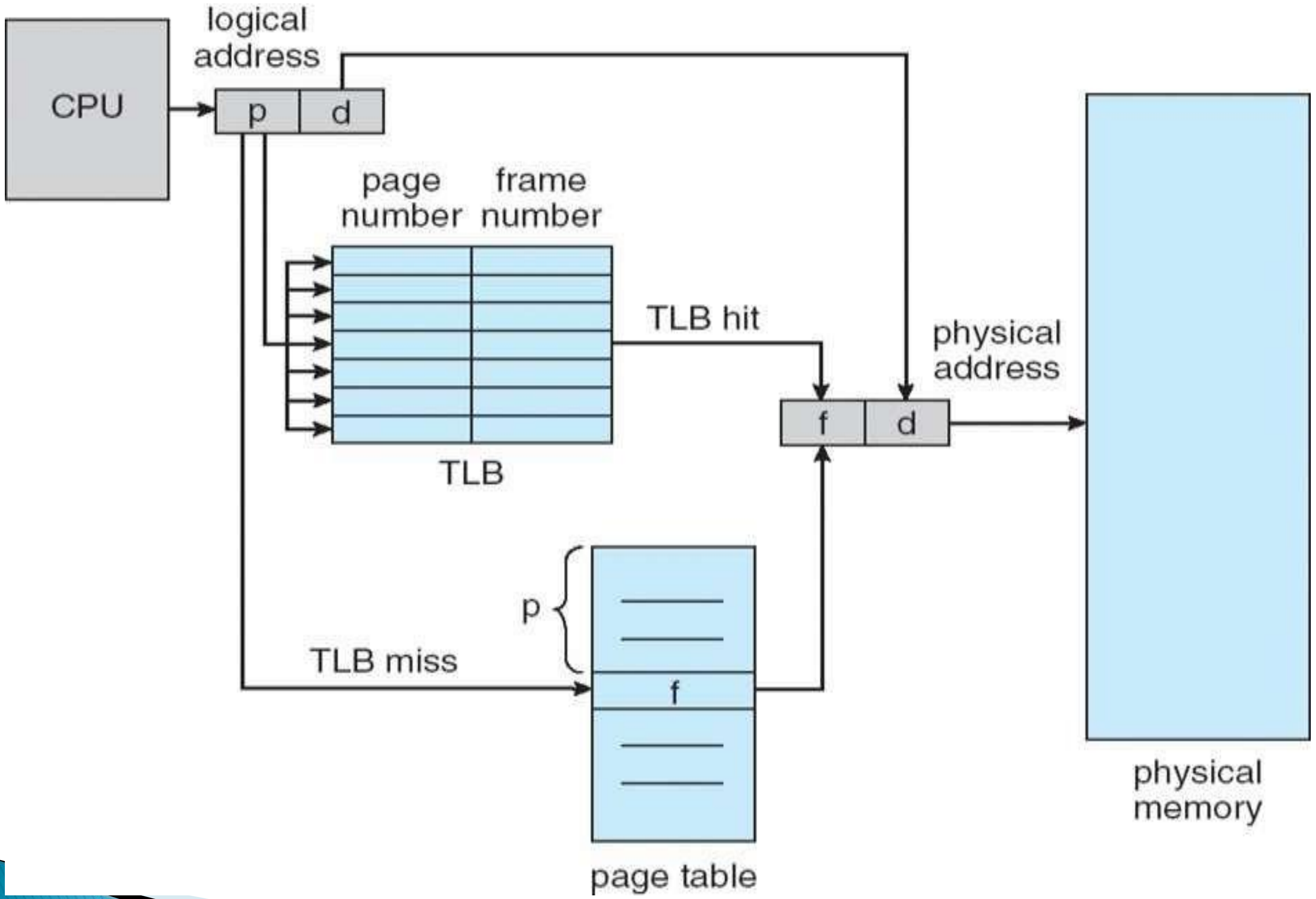
- Most contemporary computers, allow the page table to be very large (for example, 1 million entries). For these machines, the use of fast registers to implement the page table is not feasible

- Rather, the page table is kept in main memory, and a **page-table base register (PTBR)** points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time.

Translation Look-Aside Buffer (TLB).

- The standard solution to this problem is to use a special, small, fast lookup hardware cache called a **translation look-aside buffer (TLB)**.

- The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value.


CPU
logical address
p
d
page number
frame number
TLB hit
TLB
physical address
f
d
p
TLB miss
f
page table
physical memory

Protectio n

- Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table.

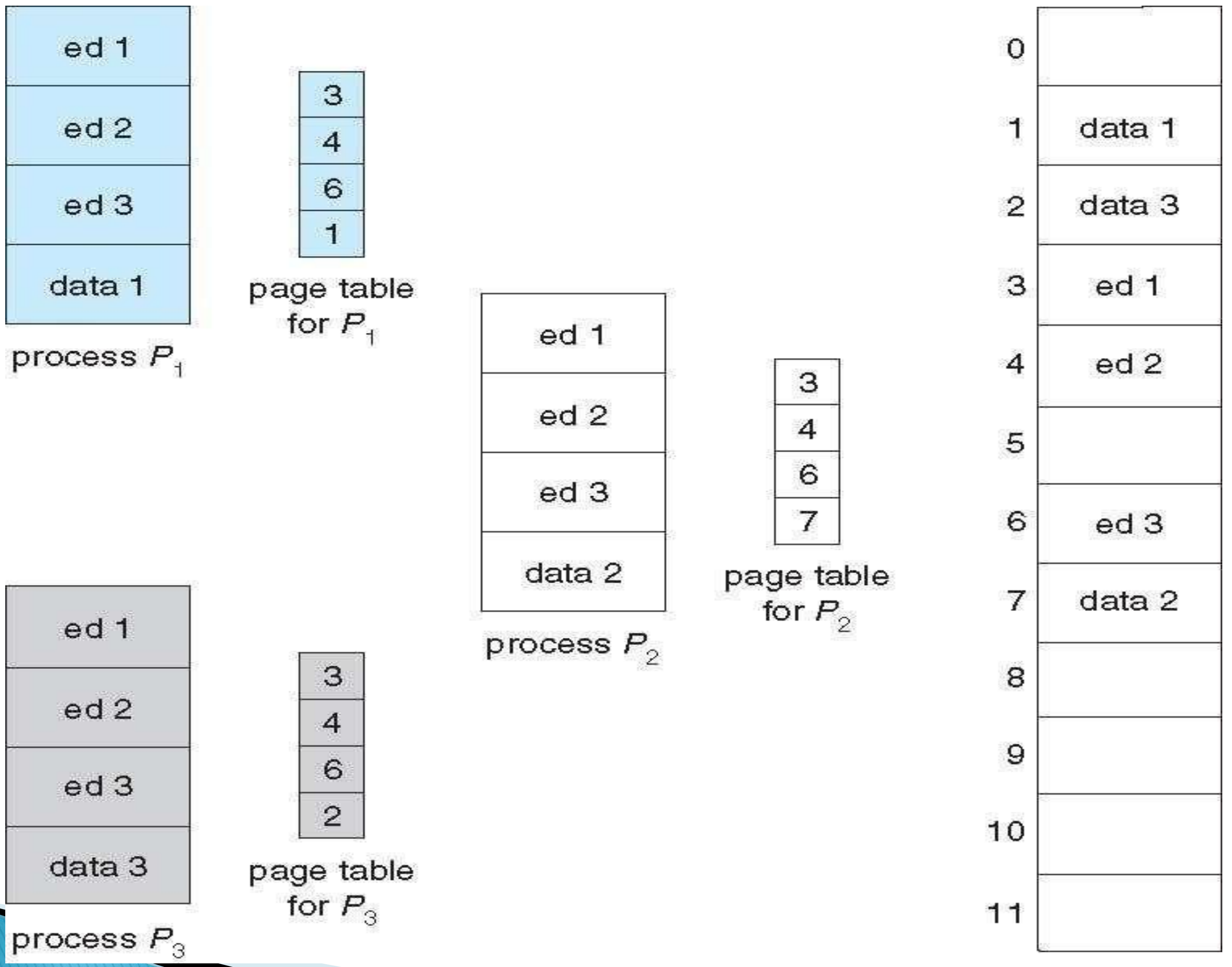
Read–Write or Read-Only Bit

- One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number.

- At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.

Shared Pages

- An advantage of paging is the possibility of ***sharing*** common code. This consideration is particularly important in a time-sharing environment.
- Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11

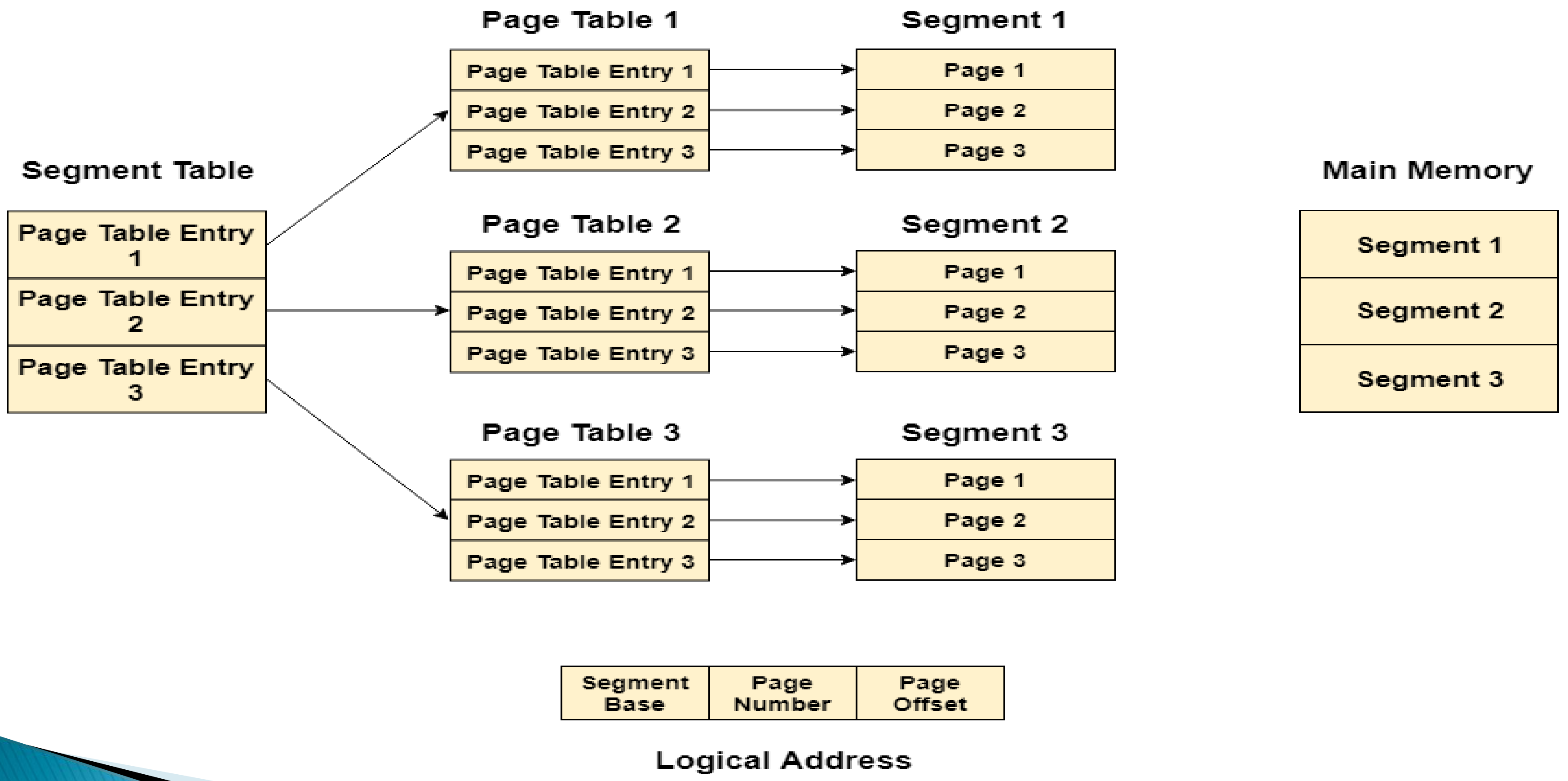
Segmentation with Paging

- Pure segmentation is not very popular and not being used in many of the operating systems.

- However, Segmentation can be combined with Paging to get the best features out of both the techniques.

- In Segmented Paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

- Pages are smaller than segments. Each Segment has a page table which means every program has multiple page tables.

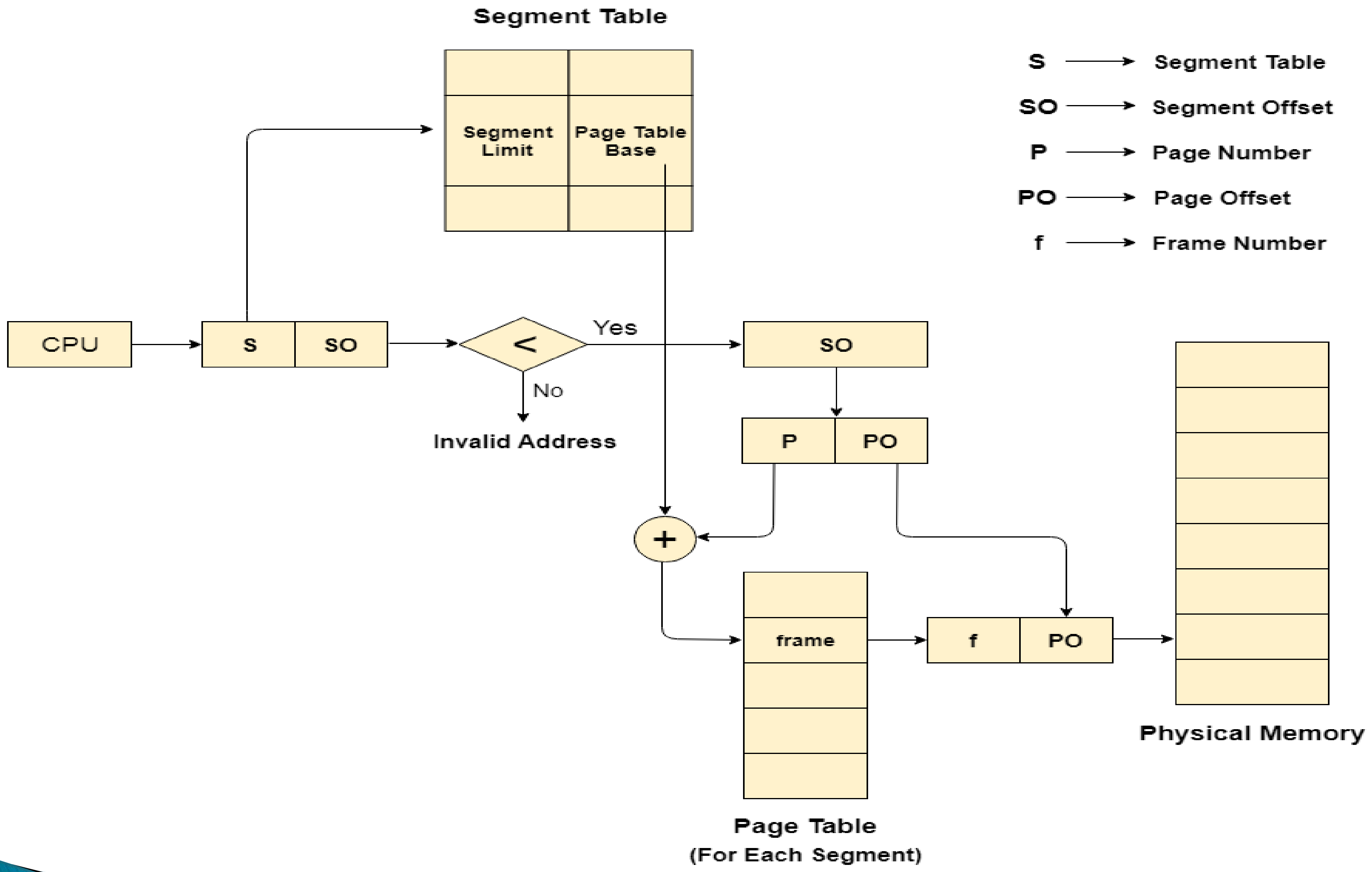


Translation of logical address to physical address

- The CPU generates a logical address which is divided into two parts: Segment Number and Segment Offset.

- The Segment Offset must be less than the segment limit. Offset is further divided into Page number and Page Offset.

- To map the exact page number in the page table, the page number is added into the page table base.


Segment Table
Segment Limit
Page Table Base
S → Segment Table
SO → Segment Offset
P → Page Number
PO → Page Offset
f → Frame Number
CPU
S
SO
<
Yes
No
Invalid Address
SO
P
PO
+
frame
f
PO
Physical Memory
Page Table
(For Each Segment)

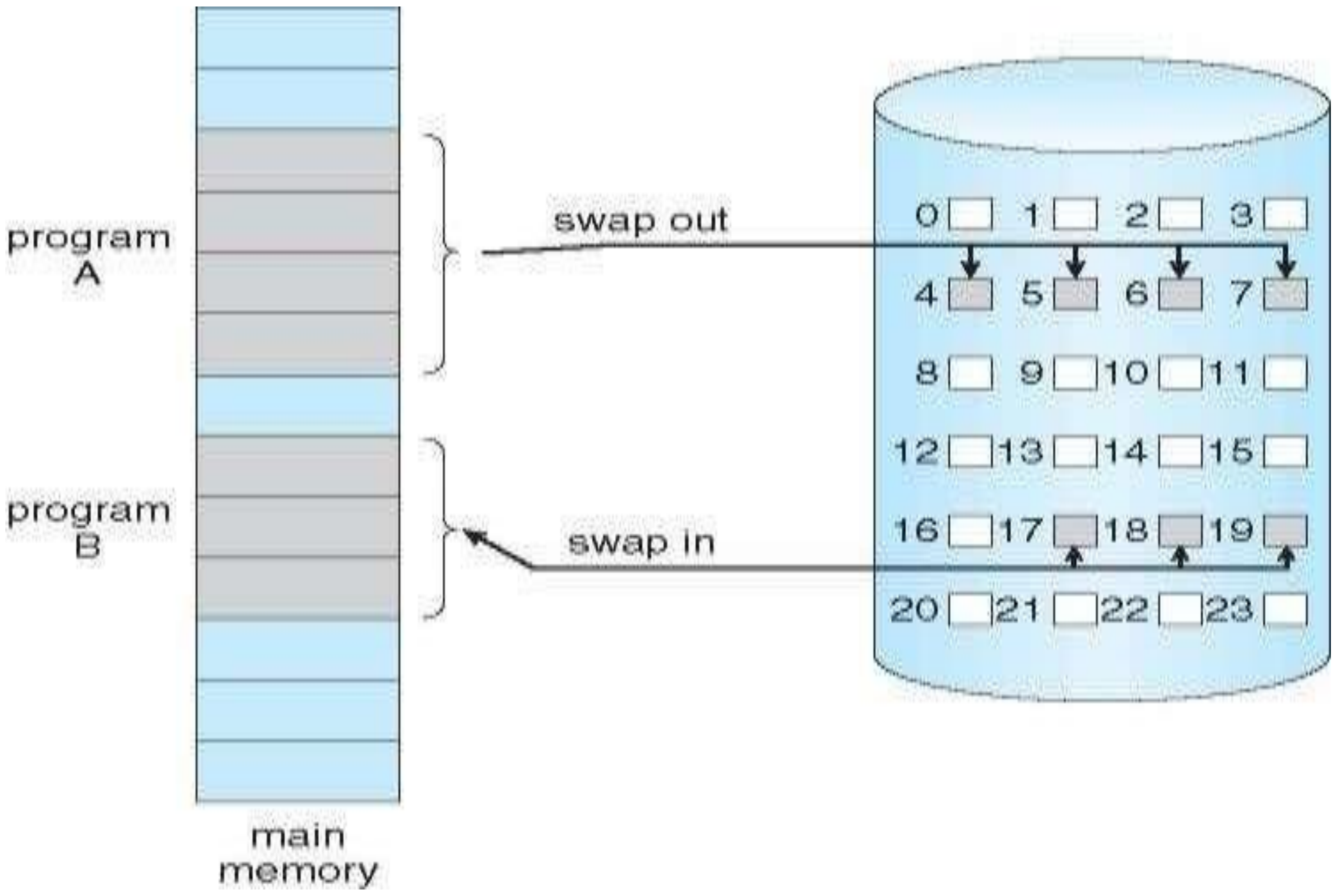
Demand Paging

- Loading the entire program into memory results in loading the executable code for ***all*** options, regardless of whether or not an option is ultimately selected by the user.

- An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.

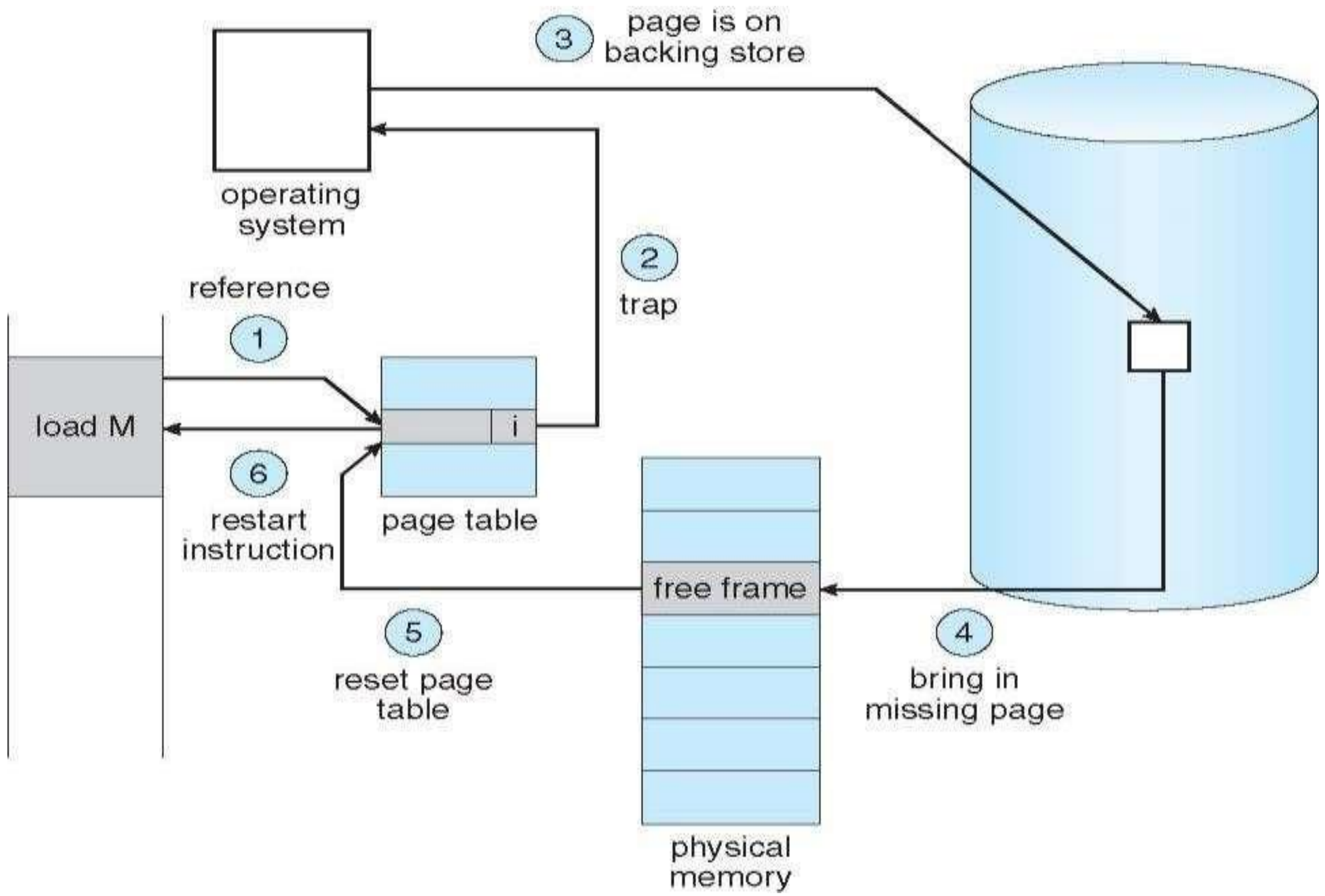
Lazy Swapper

- A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory (usually a disk).
- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, though, we use a **lazy swapper**.
- A lazy swapper never swaps a page into memory unless that page will be needed.


program A
program B
swap out
swap in
main memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

Page Fault

- Access to a page marked invalid causes a **page fault**.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system.
- This trap is the result of the operating system's failure to bring the desired page into memory.


3
page is on backing store
operating system
2
trap
reference
1
load M
i
6
restart instruction
page table
free frame
5
reset page table
4
bring in missing page
physical memory

Demand Paging

- In the extreme case, we can start executing a process with ***no*** pages in memory.

- When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.

- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.

- At that point, it can execute with no more faults. This scheme is **pure demand paging**: never bring a page into memory until it is required.
- **Hardware to Support Demand Paging**
- **Page table**. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.
- **Secondary memory**. This memory holds those pages that are not present in main memory.

- The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space**.

- A crucial requirement for demand paging is the ability to restart any instruction after a page fault.

Page Replacement

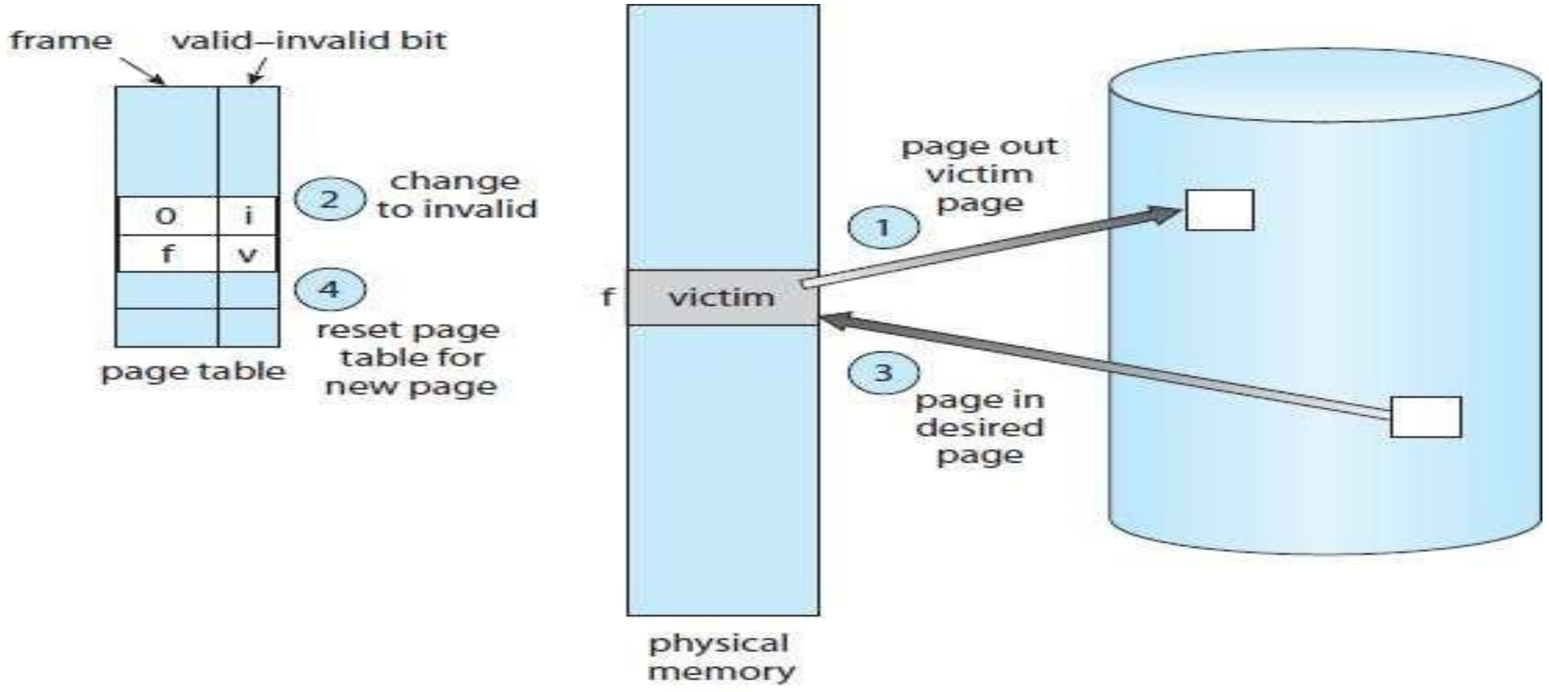
- Page replacement takes the following approach
  - Find the location of the desired page on the disk.
  - Find a free frame:
    - If there is a free frame, use it.
    - If there is no free frame, use a page-replacement algorithm to select a victim frame.

- Write the victim frame to the disk; change the page and frame tables accordingly.
  - Read the desired page into the newly freed frame; change the page and frame tables.
  - Continue the user process from where the page fault occurred.

Modify Bit (or **Dirty Bit**).

- If no frames are free, ***two*** page transfers (one out and one in) are required.

- This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a **modify bit** (or **dirty bit**).



Page replacement.

Page Replacement Algorithms

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.
- We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

- The FIFO page-replacement algorithm is easy to understand and program.

- However, its performance is not always good. a bad replacement choice increases the page-fault rate and slows process execution.

- If we place an active page, some other page should be replaced to bring it back.

Optimal Page Replacement

- It says that, Replace the page that will not be used for the longest period of time.
- It has the lowest page-fault rate of all algorithms and will never suffer from Belady’s anomaly.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

- LRU replacement associates with each page the time of that page’s last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.
- Like optimal replacement, LRU replacement does not suffer from Belady’s anomaly. Both belong to a class of page-replacement algorithms, called **stack algorithms**.

UNIT-5

UNIT-5 File System

- A file is a named collection of related information that is recorded on secondary storage.
- (or)A file is the smallest allotment of logical secondary storage.
- (or)A file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. Many different types of information may be stored in a file.

Access methods

- Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in the following ways,

  Sequential Access

- The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other. It is based on a tape model of a file and works as well on sequential-access devices.

Direct Access (or Relative Access)

- Another method is **direct access** (or **relative access**).
- Here, a file is made up of fixed- length **logical records** that allow programs to read and write records rapidly in no particular order.
- The direct-access method is based on a disk model of a file, since disks allow random access to any file block.

- For direct access, the file is viewed as a numbere sequence of blocks or records.

- Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct-access file.

Indexed Access

  - It involves the construction of an index for the file. The **index**, like an index in the back of a book, contains pointers to the various blocks.

  - To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

Directory Structure

- The most common schemes for defining the logical structure of a directory are
-

Single-Level Directory

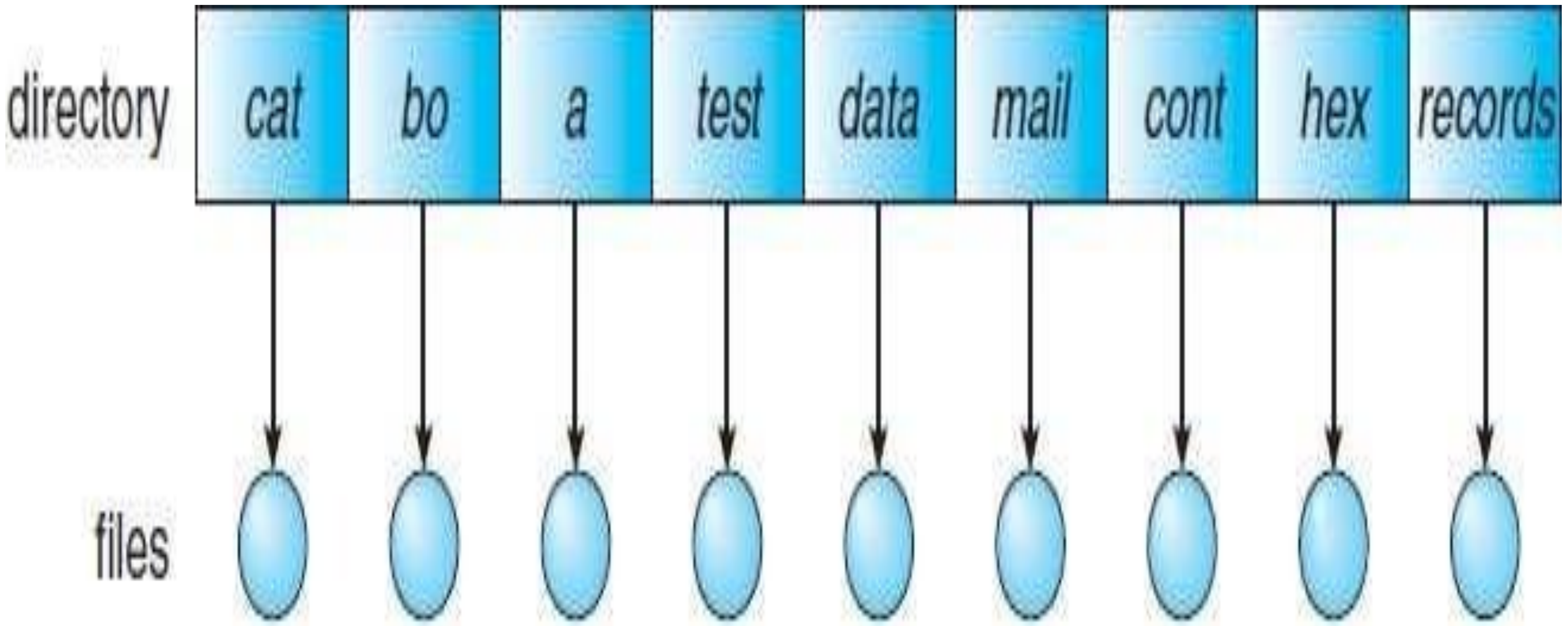
  – The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand.

  –

Limitations

- All files are in the same directory, they must have unique names. If two users call their data file test.txt, then the unique-name rule is violated.

- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. Keeping track of so many files is a problem.



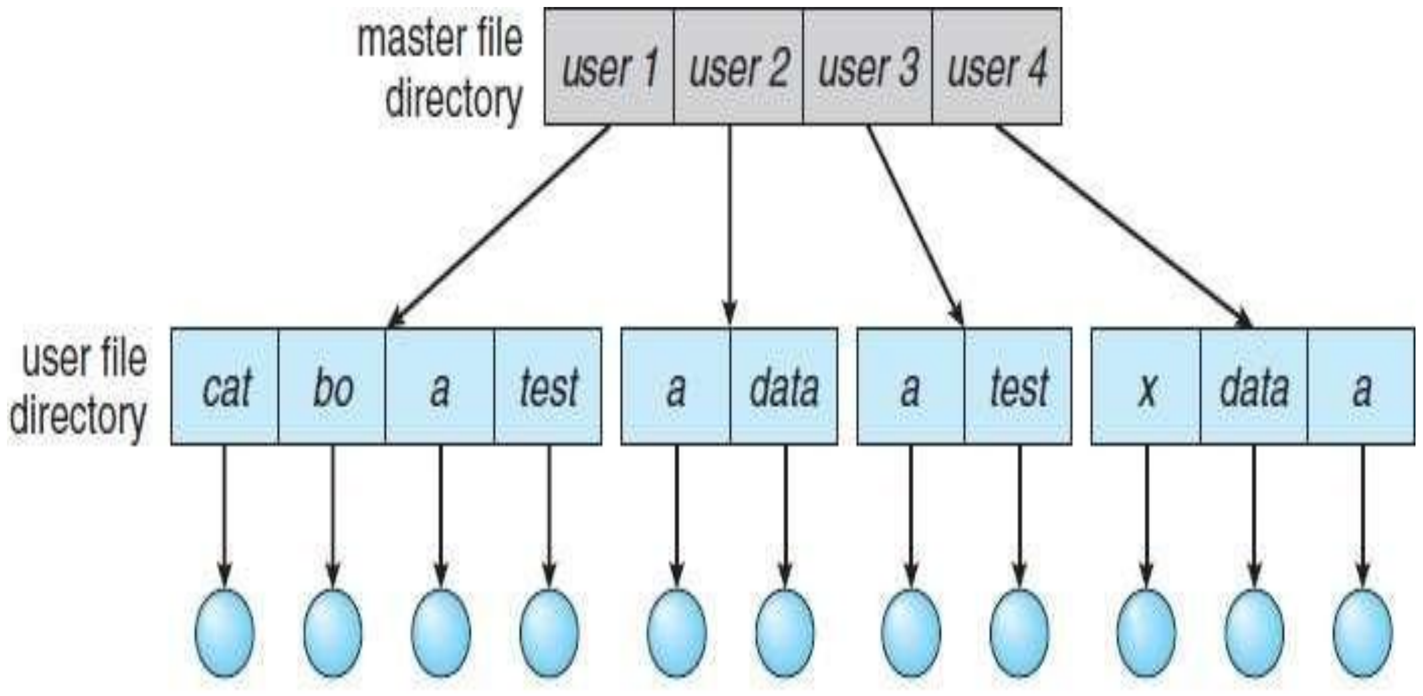
Single-level directory.

Two-Level Directory

- The standard solution to eliminate confusion of file names among different users is to create a separate directory for each user.
- So the two level directory structure contains 2 directories
  - Master File Directory (MFD) at the top level.
  - User File Directory (UFD) at the second level and
  - Actual files are at the third level.

- Each user has his own **user file directory (UFD).** When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user
- When a user refers to a particular file, only his own UFD is searched.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.

- To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.



Two-level directory structure.

Tree–Structured Directories

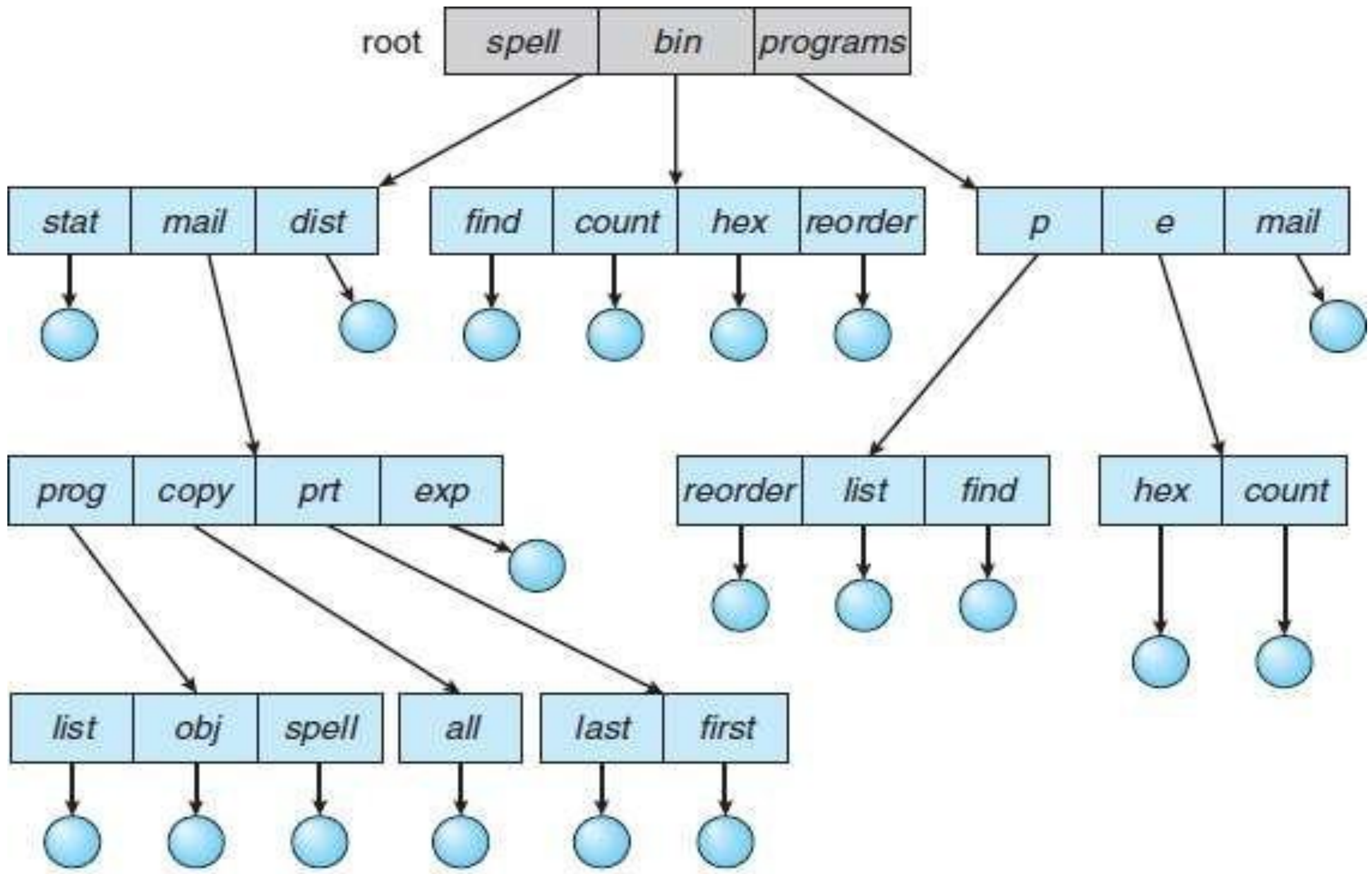
- A tree is the most common directory structure. The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- A directory is simply another file, but it is treated in a special way. All directories have the same internal format.
- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1). Special system calls are used to create and delete directories.

Current Directory

- Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.

- When reference is made to a file, the current directory is searched.

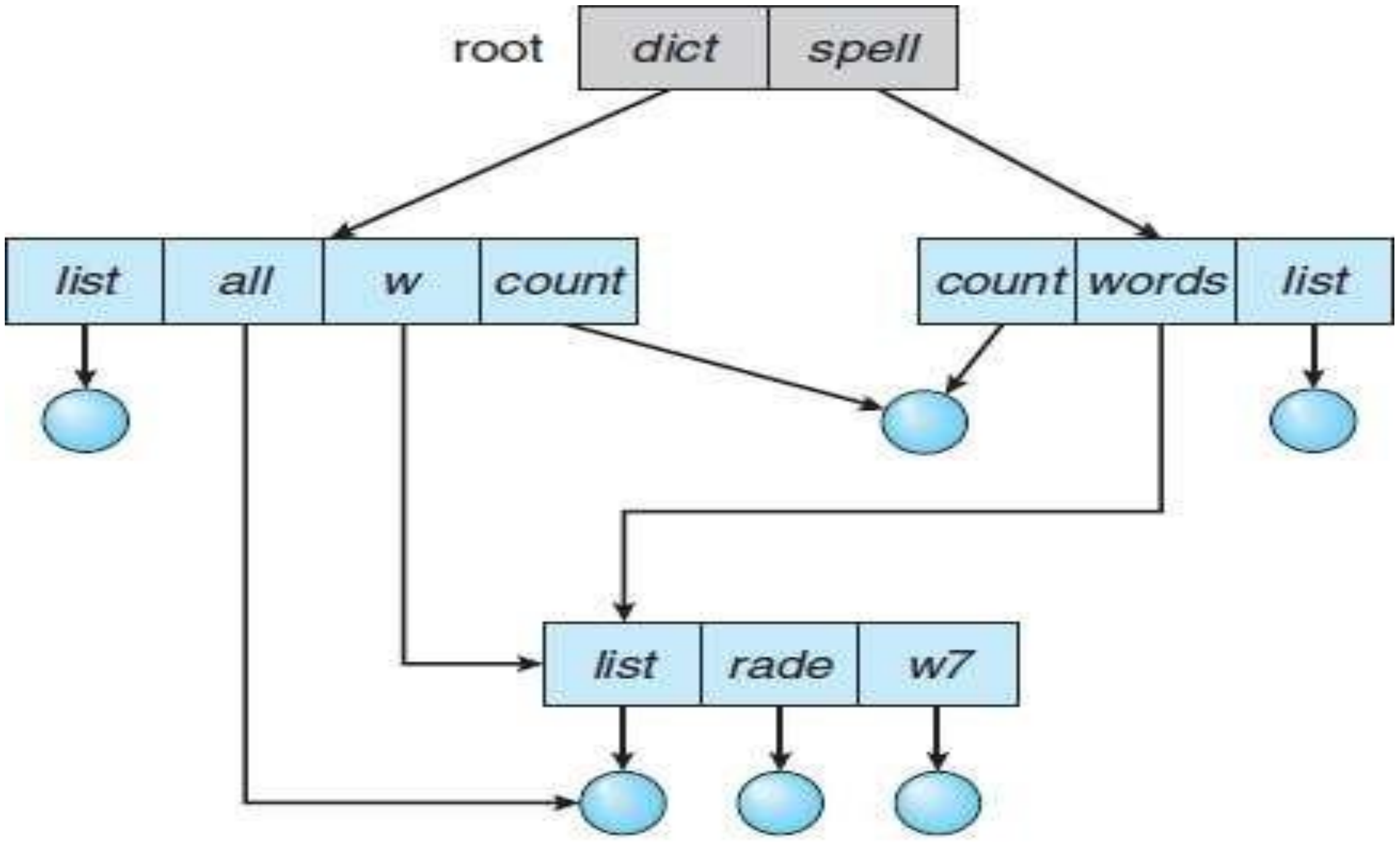
- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory (using change directory () system call) to be the directory holding that file.



Tree-structured directory structure.

Acyclic–Graph Directories

- A tree structure prohibits the sharing of files or directories.
- An **acyclic graph** i.e**.,** a graph with no cycles which allows directories to share subdirectories and files.
- The same file or subdirectory may be in two different directories.
- An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex.



Acyclic-graph directory structure.

Protectio n

- When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

- Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

Types of Access

- Protection mechanisms provide controlled access by limiting the types of file access that can be made.

- Access is permitted or denied depending on several factors, one of which is the type of access requested. Several different types of operations may be controlled:

  - **Read**. Read from the file.
  - **Write**. Write or rewrite the file.
  - **Execute**. Load the file into memory and execute it.

- **Append**. Write new information at the end of the file.
- **Delete**. Delete the file and free its space for possible reuse.
- **List**. List the name and attributes of the file.

- Other operations, such as renaming, copying, and editing the file, may also be controlled.

Access Control

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- Different users may need different types of access to a file or directory.
- The most general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.

Other Protection Approaches

- Another approach to the protection problem is to associate a password with each file. Just as access to the computer system is often controlled by a password, access to each file can be controlled in the same way.

- If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however.

File System Structure

- Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose are,
- A disk can be rewritten.
- A disk can access directly any block of information it contains.
- **File systems** provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily.

Logical File System

- The **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data.

- The logical file system manages the directory structure to provide the file-organization module with this information
- **Application Programs-**It contains user code that is making a request.

File-Organization Module

- The **file-organization module** knows about files and their logical blocks and physical blocks.

- By knowing the type of file allocation used and the location of the file, the file organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.

Basic File System

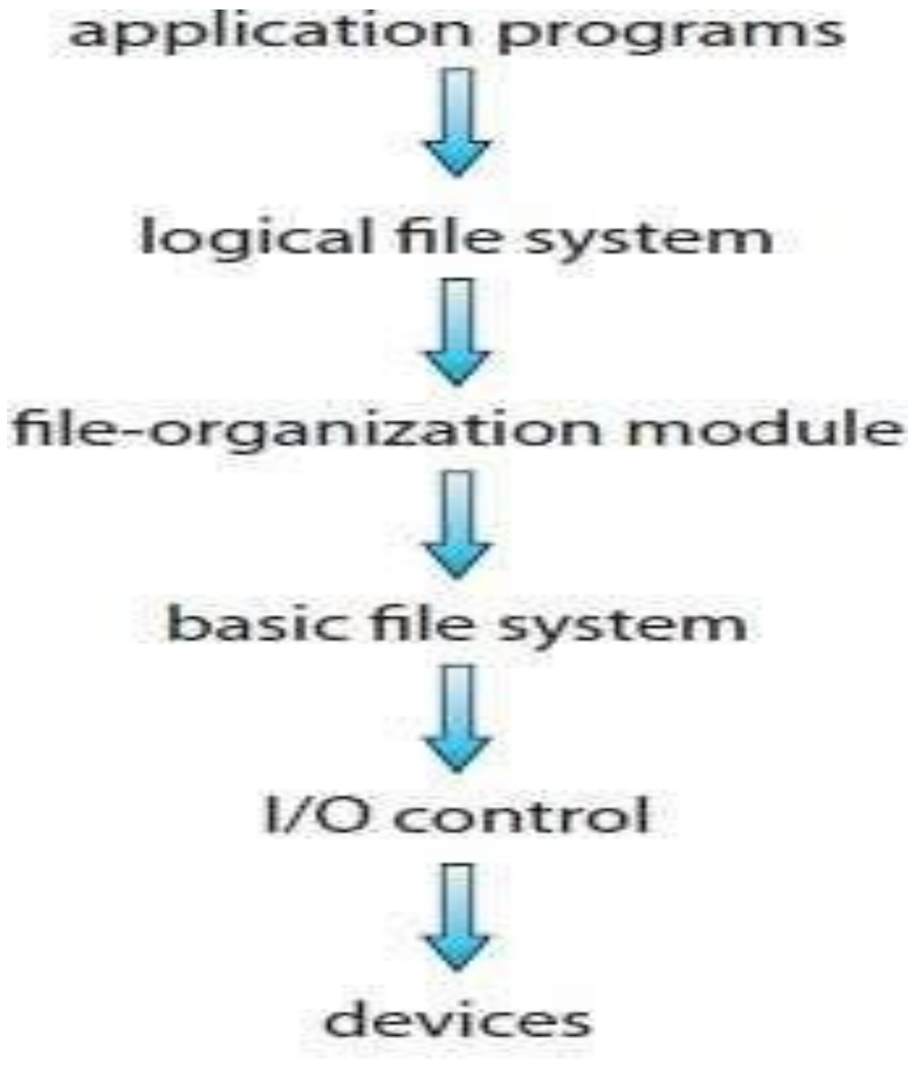
- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address.

I/O control

- The **I/O control** level consists of device drivers and interrupts handlers to transfer information between the main memory and the disk system.

- It acts like a translator, inputting high-level commands such as "retrieve block 123." And outputting low-level, hardware
- specific instructions that are used by the hardware controller

- **Devices-**These are the actual hardware devices like disk.



Layered file system

Allocation methods

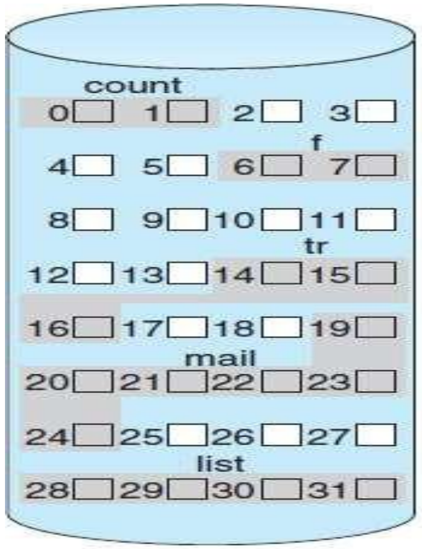
- Many files can be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.

- The following are the three major methods of allocating disk space that are in wide use:

Contiguous Allocation

  - **Contiguous allocation** requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk.

- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.
- If the file is *n* blocks long and starts at location *b,* then it occupies blocks *b, b* + 1, *b* + 2, *b* + *n* − 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation of disk space.

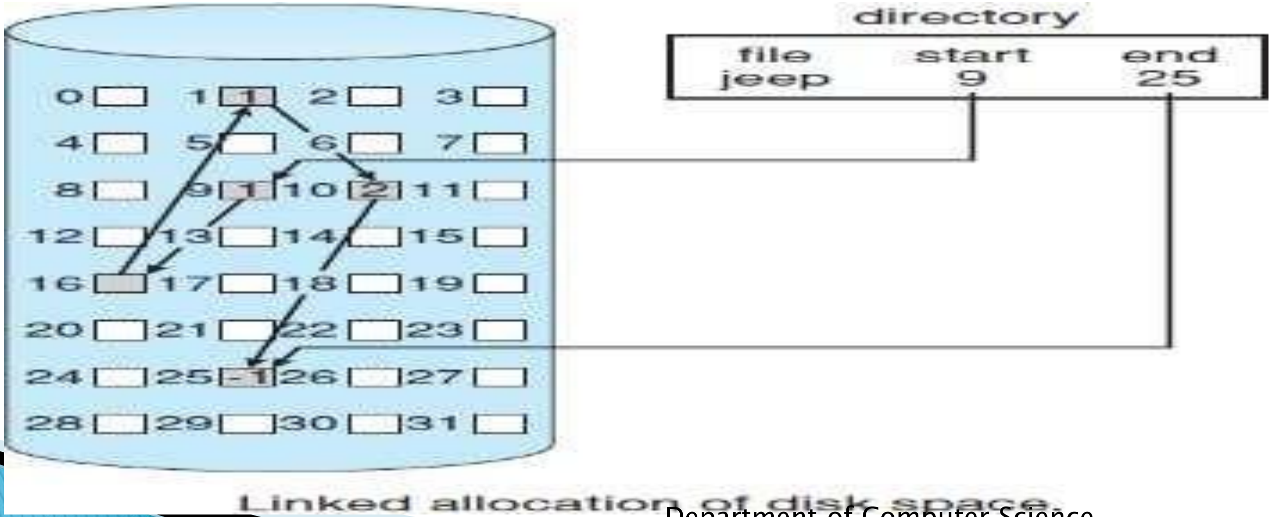
Linked Allocation

- **Linked allocation** solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks.

- The disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

- For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25.

- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to null (the end-of-list pointer value) to signify an empty file. The size field is also set to 0.

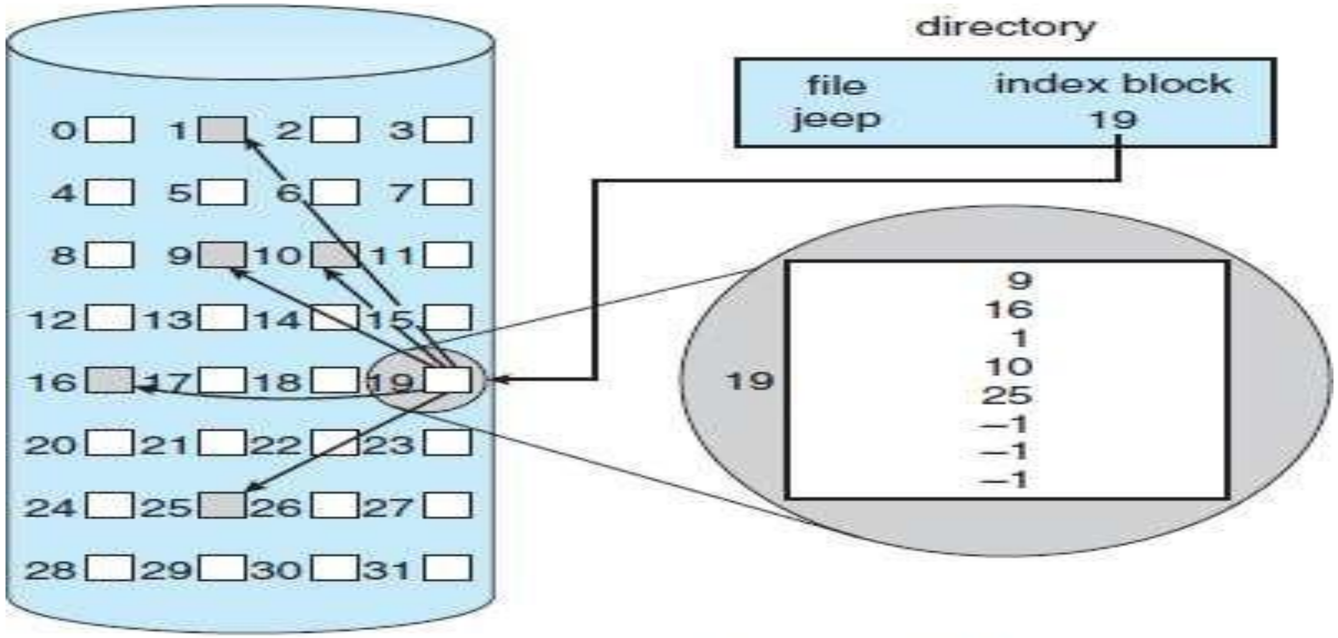


Linked allocation of disk space.

Indexed Allocation

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
- However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
- **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **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.



Indexed allocation of disk space.

Free-space Management

- To keep track of free disk space, the system maintains a **free-space list**.

- The free- space list records all free disk blocks—those not allocated to some file or directory.

- The following are implementations of free space list.

  Bit Vector

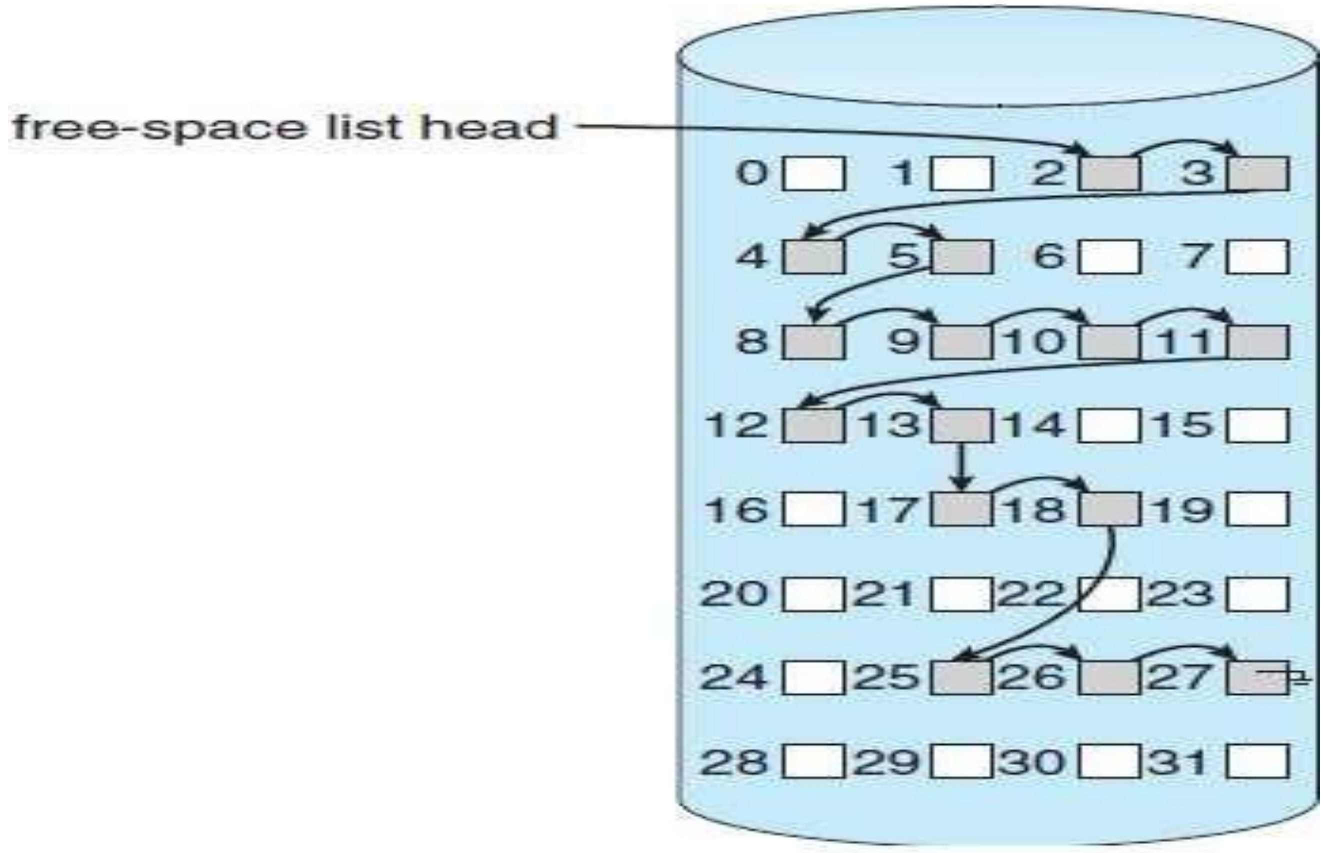
  - Free-space list is frequently implemented as a **bit map** or **bit vector.** Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000...

Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

- This first block contains a pointer to the next free disk block, and so on.



Linked free-space list on disk.

Grouping

- A modification of the free-list approach stores the addresses of *n* free blocks in the first free block.

- The first *n*−1 of these blocks are actually free. The last, block contains the addresses of another *n* free block, and so on.

- The addresses of a large number of free blocks can now be found quickly.

Countin g

- Several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering.
- Thus, rather than keeping a list of *n* free disk addresses, we can keep the address of the first free block and the number (*n*) of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.

Space Maps

- Oracle's **ZFS** file system was designed to encompass huge numbers of files, directories, and even file systems.

- In its management of free space, ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size.

- Each metaslab has an associated space map.

- The space map is a log of all block activity (allocating and freeing), in time order, in counting format.

- When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the login to that structure.

File Operations

create ()

- This is used to create a file. Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.

open ()

- Many systems require that an open () system call be made before a file is first used. When a file has been opened its entry is added in the open file table. It also contains open count associated with each file to indicate how many processes have the file open.

read ()

- To read from a file, we use a system call that specifies the name of the file and **read pointer** to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.

write ()

- To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location.
- The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

close ()

- This closes a file. Each close () decrements the open count and when the count reaches zero, the file is no longer in use so it can be closed.

delete ()

- To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.

truncate ()

- The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released.

seek ()

- It is also called as Reposition. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O.

unlink ()

- Deletes a name from the file system. If that name was the last link to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.

your roots to success...

THANK YOU