S Afroz

UNIT-3

~~Unit-3~~

DEADLOCKS

* In a multi-programming, several processes may compete for a number of resources.
* A process request resources, and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a 'DEADLOCK'.

System Model:-

* A system consists of a finite number of resources.
* These resources should be distributed among a number of co-operating processes.
* The resources are partitioned into several types.
* Each resource type consists of some number of instances.
* Printers, DVD drives, files, CPU cycles, memory space are examples of resource types.
* If a system has the resource type: Printer,

It may have 5 instances (five printers).

* A process may request as many resources as it requires to carry out its designated task.
* A process must request a resource before using it and must release the resource after using it.
* A process may utilize a resource in only the following sequence :

a) Request → A process should request first for any resource.

If the request cannot be granted immediately (Suppose, if the resource is being used by another process) then the requesting process must wait until it can get the resource.

b) Use → The process can operate (use) the resource. For example, if the resource is a printer, the process can print on the printer.

c) Release → The process releases the resource.

Consider a system with three DVD-drives. Suppose each of three processes holds one of these DVD-driver. If each process now requests another drive, the three processes will be in a deadlock state.

Consider a system with one printer and one DVD-drive. Suppose the process Pi is holding the DVD-drive and process Pj is holding the printer. If the Pi request the printer and Pj requests the DVD-drive, a deadlock occurs.

4m1
(10) SA

Deadlock Characterization → UM

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.

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

I.Necessary Conditions:

A deadlock situation can arise if the following conditions hold simultaneously in a system.

a.Mutual Exclusion: - At least one resource must be held in a non-sharable mode; if any other process requests this resource, then that process must wait for the resource to be released.

b.Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

c.No preemption - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.

d.Circular Wait - A set of waiting processes { P0, P1, P2, . . ., PN } must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , PN-1 is waiting for resource held by PN, and PN is waiting for a resource held by P0. A set of processes are waiting for each other in circular form.

We emphasizes that all four above conditions must hold for a deadlock to occur.

II. Resource-Allocation Graph:

Resource-Allocation Graph

* Deadlocks can be described more precisely in terms of a directed graph called a "System resource-allocation graph".
* This graph consists of a set of vertices 'V' and set of edges 'E'.
* The set of vertices 'V' is partitioned into two different types of nodes:

  P = {P1, P2, --- Pn}, the set consisting of all the active processes in the system, and

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

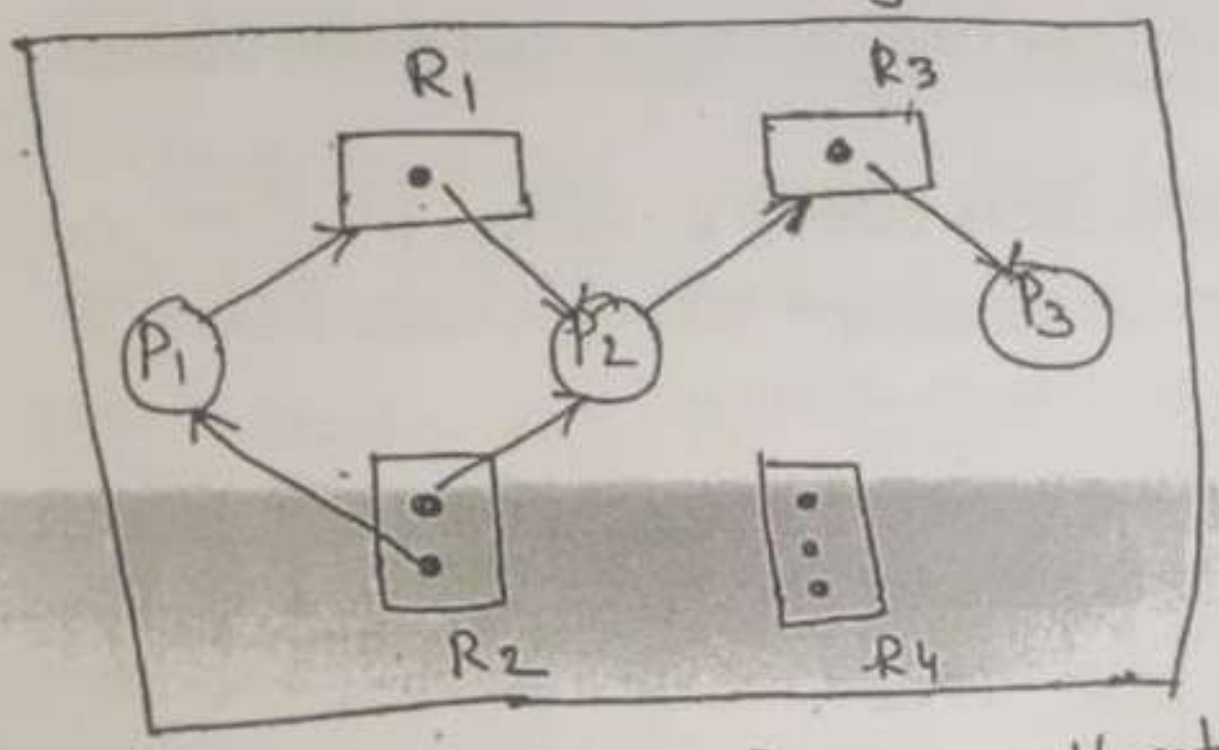


Fig: 7.2: Resource-allocation graph, but no deadlock

* A directed edge $P_i \rightarrow R_j$ is called a "Request Edge"
* A directed edge $R_j \rightarrow P_i$ is called an "Assignment Edge"
* A directed edge from Process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$; it signifies that Process P_i has requested an instance of resource type R_j and is currently waiting for that resource

* A directed edge from resource type R_J to Process P_i is denoted by $R_J \rightarrow P_i$; it signifies that an instance of resource type R_J has been allocated to Process P_i.

In the above R-A graph: (Fig 7.2):

→ we represent each Process P_i as a circle and
→ Each resource type R_j as a rectangle.
→ we represent each instance as a dot (•) within the rectangle
→ Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
→ Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
→ Process P_3 is holding an instance of R_3.

The above R-A graph has the following Situation:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$$

* Given the definition of resource-allocation graph, if the graph contains no cycles, then no process in the system is deadlocked.
* If the graph does contain a cycle, then a deadlock may exist

The above (Fig: 7-2) R-A graph does not contain a cycle and not involved in deadlock.

* Suppose that Process P_3 requests an instance of resource type R_2, a request edge $P_3 \rightarrow R_2$ is added to the graph (Fig: 7-3)
* At this point, two 'cycles' exist in the system

a) $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
b) $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

+Now, Processes P_1, P_2 and P_3 are deadlocked.

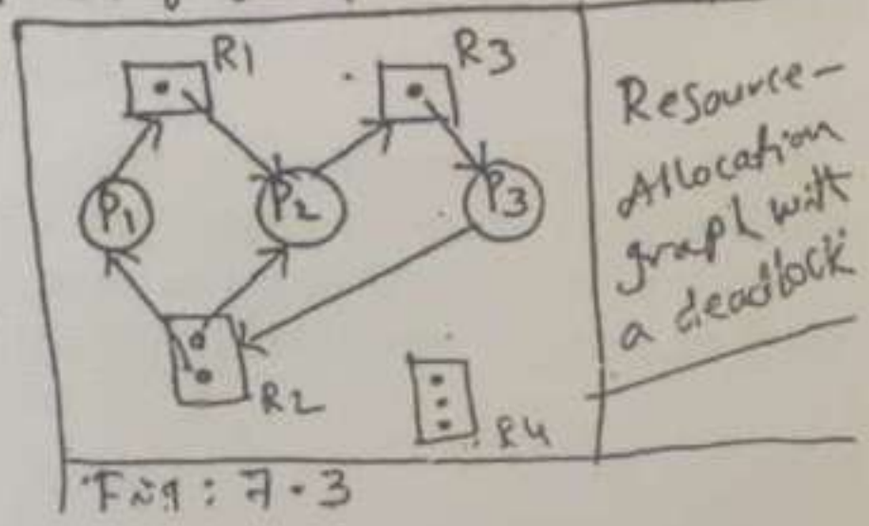


Fig: 7-3

Deadlock Handling Approaches/Methods

I. Deadlock Prevention

II. Deadlock Avoidance

III. Deadlock Detection

IV. Deadlock Recovery

II. DEADLOCK AVOIDANCE :

The general idea behind deadlock avoidance is to prevent deadlocks from ever happening. It is better to avoid a deadlock instead of taking action after the Deadlock has occurred. It needs additional information, like how resources should be used. Deadlock avoidance is the simplest and most useful model that each process declares the maximum number of resources of each type that it may need. The deadlock-avoidance algorithm helps you to dynamically assess the resource-allocation state so that there can never be a circular-wait situation.

1. Safe State:

A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.

More formally, a state is safe if there exists a ***safe sequence*** of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)

If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

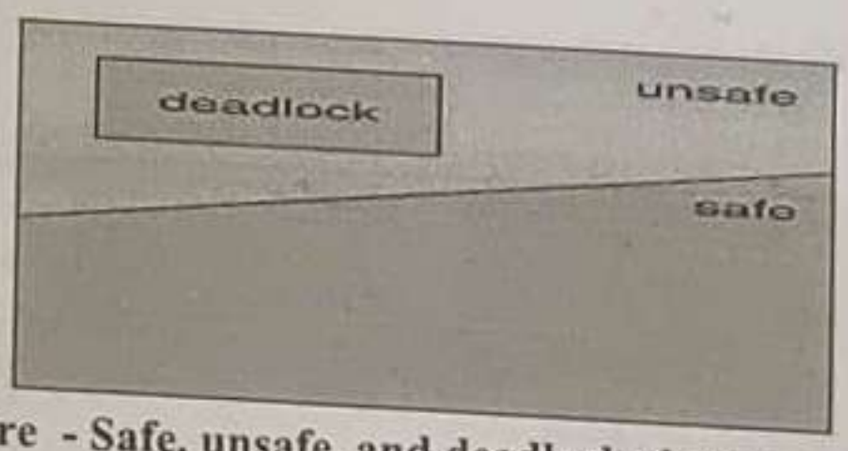


Figure - Safe, unsafe, and deadlocked state spaces.

For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the safe sequence?

	Maximum Needs	Current Allocation
P0	10	5
P1	4	2
P2	9	2

What happens to the above table if process P2 requests and is granted one more tape drive?

Key to the safe state approach is that when a request is made for resources, the request is granted only if the resulting allocation state is a safe one.

2. Resource-Allocation Graph Algorithm:

If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.

In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with ***claim edges***, noted by dashed lines, which point from a process to a resource that it may request in the future.

In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources. (Alternatively, processes may only make requests for resources for which they have already established claim edges, and claim edges cannot be added to any process that is currently holding resources.)

When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when a resource is released, the assignment reverts back to a claim edge.

This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.

Consider for example what happens when process P2 requests resource R2:

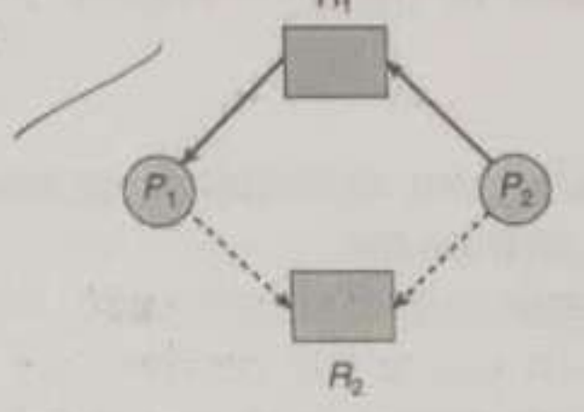


Figure - Resource allocation graph for deadlock avoidance

The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

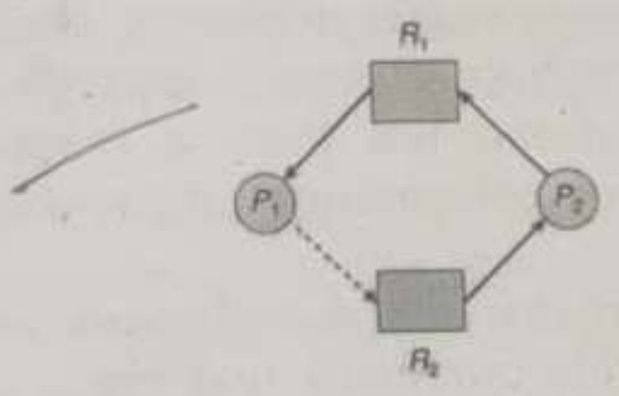


Figure - An unsafe state in a resource allocation graph

3. Banker's Algorithm:

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.
- The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won't loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)

- When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
- The banker's algorithm relies on several key data structures: (where n is the number of processes and m is the number of resource categories.)
    - Available[m] indicates how many resources are currently available of each type.
    - Max[n][m] indicates the maximum demand of each process of each resource.
    - Allocation[n][m] indicates the number of each resource category allocated to each process.
    - Need[n][m] indicates the remaining resources needed of each type for each process. (Note that Need[i][j] = Max[i][j] - Allocation[i][j] for all i, j.)
- For simplification of discussions, we make the following notations / observations:
    - One row of the Need vector, Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
    - A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

III. DEADLOCK DETECTION:

- If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
- In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

1. Single Instance of Each Resource Type:

- If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
- A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
- An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

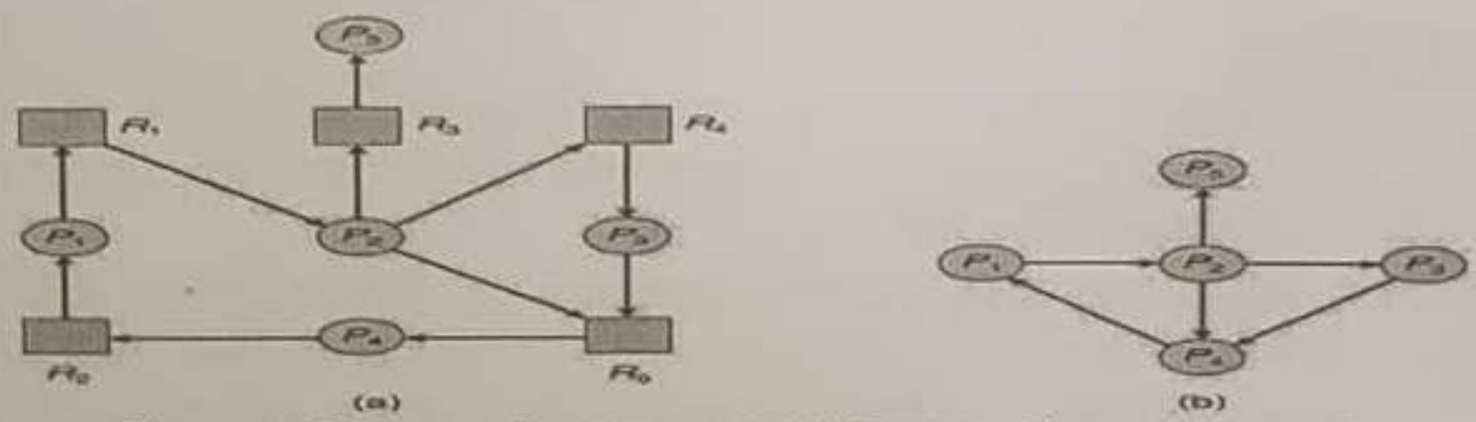


Figure - (a) Resource allocation graph. (b) Corresponding wait-for graph

- As before, cycles in the wait-for graph indicate deadlocks.
- This algorithm must maintain the wait-for graph, and periodically search it for cycles.

2. Several Instances of a Resource Type:

- The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
  - In step 1, the Banker's Algorithm sets Finish[i] to false for all i. The algorithm presented here sets Finish[i] to false only if Allocation[i] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[i] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
  - Steps 2 and 3 are unchanged
  - In step 4, the basic Banker's Algorithm says that if Finish[i] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[i] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.

****

IV. DEADLOCK RECOVERY:

There are three basic approaches to recovery from deadlock:

- Inform the system operator, and allow him/her to take manual intervention.
- Terminate one or more processes involved in the deadlock
- Preempt resources.

1. Process Termination:

Two basic approaches, both of which recover resources allocated to terminated processes:

- Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
- Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

In the latter case there are many factors that can go into deciding which processes to terminate next:

- Process priorities.
- How long the process has been running, and how close it is to finishing.
- How many and what type of resources is the process holding. (Are they easy to preempt and restore?)
- How many more resources does the process need to complete.
- How many processes will need to be terminated

2. Resource Preemption:

When preempting resources to relieve deadlock, there are three important issues to be addressed:

a. **Selecting a victim** - Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
b. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. (I.e. abort the process and make it start over.)
c. **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

I. DEADLOCK PREVENTION

* By ensuring that at least one of the necessary conditions cannot hold, we can "prevent" the occurence of a deadlock

a) Mutual Exclusion :-

* The mutual-exclusion condition must hold for nonsharable resources.
* For example, a Printer cannot be simultaneously shared by several processes.
* Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
* For example, "Read-only files" are good example of a sharable resource
* A process never needs to wait for a sharable resource.
* However, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically nonsharable

b) Hold and wait :-

* To ensure that the hold-and-wait condition never occurs in the system:
  - → one protocol is that a process to request resources only when it has none
  - → Second Protocol is that each process should request all its needed resources before it begins execution

c) No Preemption :-

* To ensure that this condition does not hold, we can use the following protocols.

If a process is holding some resources and requests another resource and that cannot be immediately allocated to it, then all resources currently being held are preempted

The preempted resources are allocated to the waiting Processes that requires.

Alternatively, if a Process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other Process that is waiting for additional resources. If so, we preempt the desired resources from the waiting Process and allocate them to the requesting Process.

d) Circular wait :-

* To ensure that this condition never holds is to impose a total ordering of all resource types and to require that each Process requests resources in an increasing order of enumaration

F (Tape drive) = 1
F (Disk drive) = 5
F (Printer) = 12

For example, a Process that wants to use the tape drive and Printer at the same time, it must first request the tape drive and then request the Printer.

PROCESS SYNCHRONIZATION

Processes executing concurrently (simultaneously) in the system may be either independent processes or co-operating processes.

* A process that donot share data with any other process is known as Independent process.
* A process that shares data with other processes is known as Co-operating process.
* A process is co-operating if it can affect or be affected by the other processes executing in the system.
* Concurrent access to shared data may result in data inconsistency.
* Process synchronization was introduced to handle problems that arise during the execution of co-operating processes.

The Critical Section Problem:-

General Structure of a Typical Process 'Pi':

```
do {
    Entry Section
        Critical Section
    Exit section
        Remainder section
} while (TRUE);
```

Consider a system consisting of the set of processes ($P_0, P_1, P_2, P_3 \dots P_{n-1}$). Each process has a segment of code called a Critical Section, in which

the process may access the common data, shared variables, updating a table data (DATA BASE), writing to a file and so-on.

* In a group of co-operating processes, at a given point of time only one process must be executing its critical section.
* When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
* It means, no two processes are executing in their critical sections at the same time.
* If any other process also wants to execute its critical section, it must wait until the first one finishes.
* Each process must request permission to enter its critical section, this is implemented as "Entry Section".
* The Critical section followed by an "exit section", in which the process releases its resources.
* The rest of the code is of a process is specified in "remainder section".
* The critical section problem must satisfy the following requirements.

a) Mutual Exclusion → Only one process at a time can be executing in the critical section.

b) Progress → The processes cannot be blocked forever waiting to get into their critical sections.

c) Bounded Waiting → A process requesting entry into their critical section will get a turn finally and there is a limit as to how many other processes get to go first.

Synchronization Hardware :-

```
do {
    acquire lock
        critical section
    Release lock
        remainder section
} while (TRUE);
```

Solution to the C.S problem using locks

Many systems provide hardware support for critical section code.

* Here, a process must acquire (get) a "lock", before entering a critical section it releases the lock when it exits the critical section.
* It could be easily implemented in a single-processor environment.
* Unfortunately, this solution is not as feasible in multi-processor environment.

---

(12) SEMAPHORES

* A semaphore poovide a convenient and effective mechanism for process synchronization.
* Semaphores can be used to solve various synchronization problems.
* A semaphore is a variable used to control access to a common resource by multiple processes.
* A semaphore 's' is an integer variable whose value indicates the status of a common resource (Exe:- Printers.

Scanners, Storage drives, data files etc).

* Semaphore is accessed only through two standard "atomic" operations — wait() , signal().

The definition of wait() is as follows:

```
wait (semaphore s)
{   while(s==0)
        ; /* wait until s>0*/
    s--;
}
```

The definition of signal() is as follows:

```
signal (semaphore s)
{
    s=s+1;
}
```

* The wait() is called when a process wants access to a resource. If s==0, process must wait until available.
* The wait() decrement the value of 's' as it would become non-negative or not zero (if s>0).
* The signal() is called when a process is done using a resource. It means the process utilizes the assigned resource. The signal function increments the value of 's'.
    * All the modifications to the integer value of semaphore s in all the wait() and signal() operations must be executed as a unit (atomic).
    * That is, when one process using the semaphore value, no other process can use that same semaphore value simultaneously.
    * In other words, no two processes can access wait() and signal() at the same time.

There are two types of semaphores.

1. Binary Semaphore :—

* Binary Semaphores can take only two values – 0 or 1.
* They are used to acquire (get) locks.
* When a resource is available, the semaphore (s) set to '1' else "0".
* If there is only one count of a resource, a binary semaphore is used.
* Binary semaphores are known as "Mutex locks", as they are locks that provide "mutual exclusion".
* We can use binary semaphores to deal with critical section problem for multiple processes.

Initially, the semaphore s initialized to 1

```
do
{  wait (mutex s)
   // critical section
   signal (mutex s)
   // remainder section
} while (TRUE);
```

Mutual Exclusion Implementation with B.S

2. Counting Semaphore :—

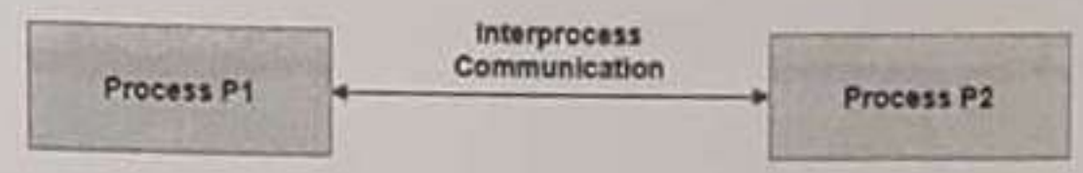
* Counting semaphore represent multiple resources.
* It can be used to control access to a given resources consisting of finite number of instances
* The semaphore is initialized to the number of resources available.
* Each process that wishes to use a resource performs a wait() operation [decrement the count];

* When a process releases a resource, it performs a signal() operation [increment the count].
* When the count for semaphore goes to zero (0), it means all resources are being used.
* After that, processes that wish to use a resource will block @ until the count becomes greater than 0.

INTER-PROCESS COMMUNICATION [IPC]

Inter-process communication is the mechanism provided by the operating system that allows processes to communicate with each other. The main aim or goal of this mechanism is to provide communications in between several processes. In short, the intercommunication allows a process letting another process know that some event has occurred.

Definition: "Inter-process communication is used for exchanging useful information between numerous threads in one or more processes (or programs)." The Processes may be running on single or multiple computers connected by a network.



Methods for Interprocess Communication:

The different approaches to implement inter-process communication are given as follows:

Pipe:

A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.

Socket:

The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for inter-process communication.

File:

A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.

Signal:

Signals are useful in inter-process communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.

Shared Memory:

Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

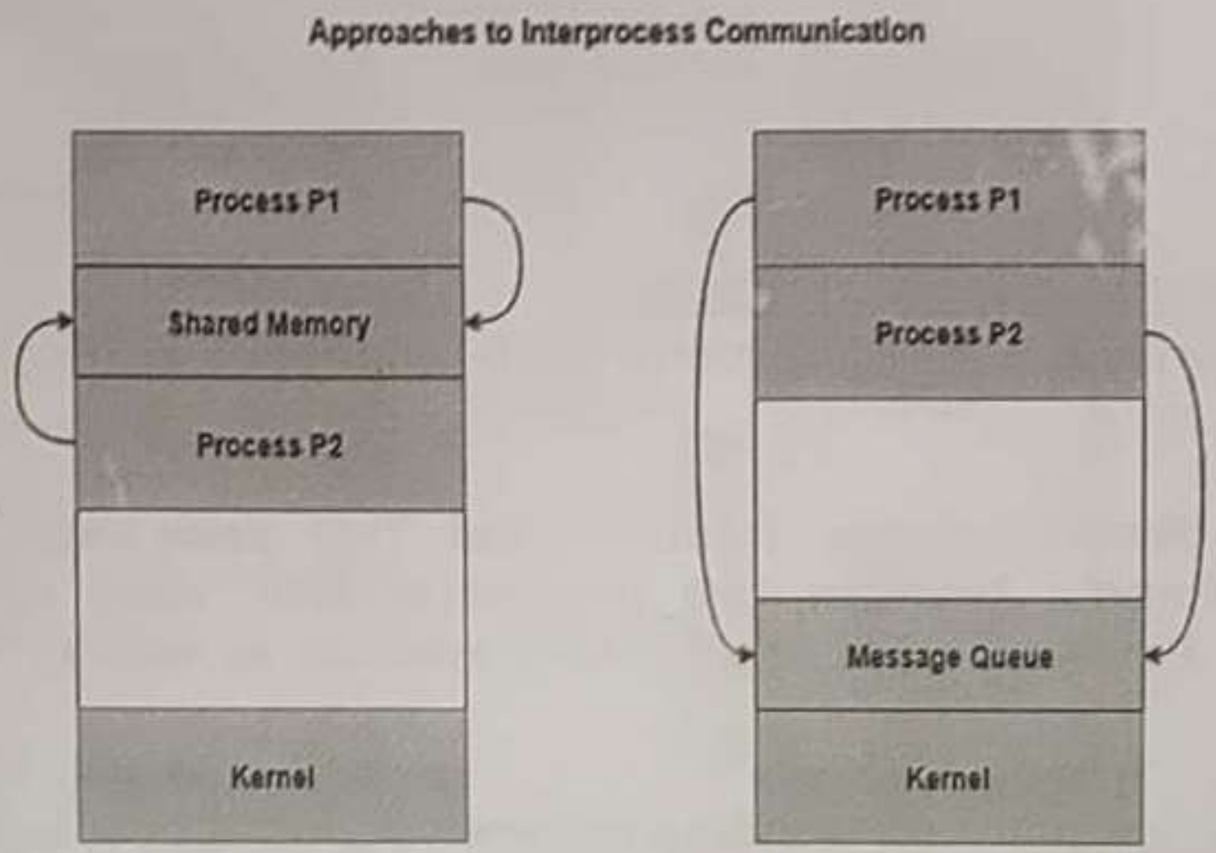
Message Queue:

Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored in the queue until their recipient retrieves them. Message queues are quite useful for inter-process communication and are used by most operating systems.

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.

A diagram that demonstrates message queue and shared memory methods of inter-process communication is as follows –



******