UNIT-III

Deadlocks - System Model, Deadlocks Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock

Process Management and Synchronization - The Critical Section Problem, Synchronization Hardware, Semaphores, and Classical Problems of Synchronization, Critical Regions, Monitors
Interprocess Communication Mechanisms: IPC between processes on a single computer system, IPC between processes on different systems, using pipes, FIFOs, message queues, shared memory.
****

In a multiprogramming environment, process requests resources; 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 by other waiting processes. This situation is called a **deadlock**.

➢ A system consists of a finite number of resources to be distributed among a number of competing processes. The resources are partitioned into several types of instances.

➢ If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request.

➢ If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

For example, a system may have two printers. These two printers may be defined to be in the same resource class if no one cares which printer prints which output. However, if one printer is on the ninth floor and the other is in the basement, then people on the ninth floor may not see both printers as equivalent, and separate resource classes may need to be defined for each printer.

➢ A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.

➢ Obviously, the number of resources requested may not exceed the total number of resources available in the system.

➢ In other words, a process cannot request three printers if the system has only two.

Example: Bridge Crossing
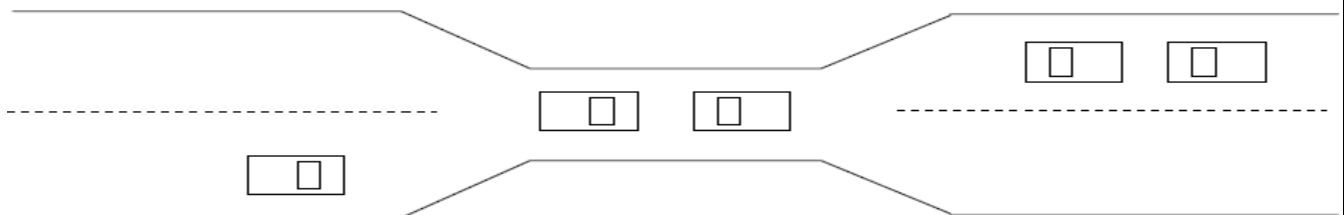
➢ Traffic only in one direction.

➢ Each section of a bridge can be viewed as a resource.

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

➢ Several cars may have to be backed up if a deadlock occurs.

➢ Starvation is possible

A process may utilize a resource in only the following sequence:

❖ **Request:** The process requests the resource. If the request cannot be granted immediately (for example, 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 (for example, if the resource is a printer, the process can print on the printer).

❖ **Release:** The process releases the resource.

Deadlock Characterization

Necessary Conditions:-

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

1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait:** A set { P0 , Pl, ... , Pn } of waiting processes must exist such that Po is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by Po.

We emphasize that **all four conditions must hold for a deadlock** to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

Resource-Allocation Graph:-

- ➢ Deadlocks can be described more precisely in terms of a directed graph called a graph. This graph consists of a set of vertices V and a set of edges E.

- ➢ The set of vertices V is partitioned into two different types of nodes:

    $P == \{ P1, P2, ..., P_n\}$, the set consisting of all the active processes in the system, and

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

- ➢ A directed edge from process P_i to resource type R_j is denoted by **P_i→** 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 →P_i**; it signifies that an instance of resource type R_j has been allocated to process P_i.

- ➢ A directed edge **P_i →** R_j is called **a request edge**; a directed edge **R_j → P_i** is called an **assignment edge.**

- • Pictorially we represent each process Pi as a circle and each resource type Rj as a rectangle.

- • Since resource type Rj may have more than one instance, we represent each such instance as a dot within the rectangle.

- • Note that a request edge points to only the rectangle R1, whereas an assignment edge must also designate one of the dots in the rectangle.

- • When process Pi requests an instance of resource type Rj, a request edge is inserted in the resource-allocation graph.

- • When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, **it releases the resource**; as a result, the assignment edge is deleted.

The resource-allocation graph shown in Figure: A depicts the following situation.

The sets P, K and E:

Set of processes   P = {P1, P2, P3}

Set of resorcesses R= {R1, R2, R3, R4}

Edges E = {Pl →Rl , P2→ R3, Rl →P2, R2→P2, R2 →Pl, R3 →P3}

Resource instances:

- ✓ One instance of resource type R1
- ✓ Two instances of resource type R2
- ✓ One instance of resource type R3

3

✓ Three instances of resource type R4

Process states:

✓ Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.

✓ Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
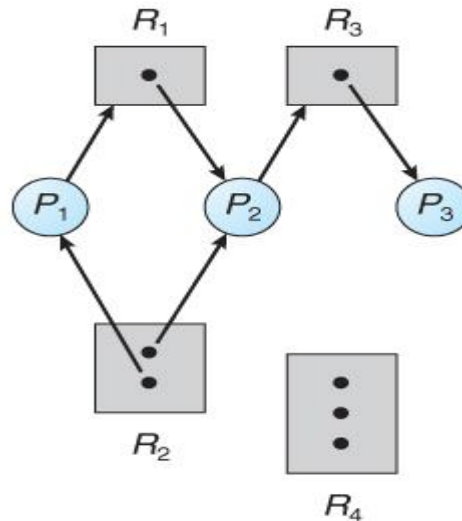
✓ Process *P3* is holding an instance of R3

Figure: A Resource-allocation graph.

- If a resource-allocation graph contains no cycles, then the system is not deadlocked.
- If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the possibility of a deadlock, but does not guarantee one.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure A.

Suppose that process P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3 → R2 is added to the graph (Figure: B). At this point, two minimal cycles exist in the system:

$$P1 \rightarrow R1 \rightarrow P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P1$$

$$P2 \rightarrow R3 \rightarrow P3 \rightarrow R2 \rightarrow P2$$

4

- Processes P1, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by processP3. Process P3 is waiting for either process P1 or process P to release resource R2.   In addition, process P1 is waiting for process P2 to release resource R1.
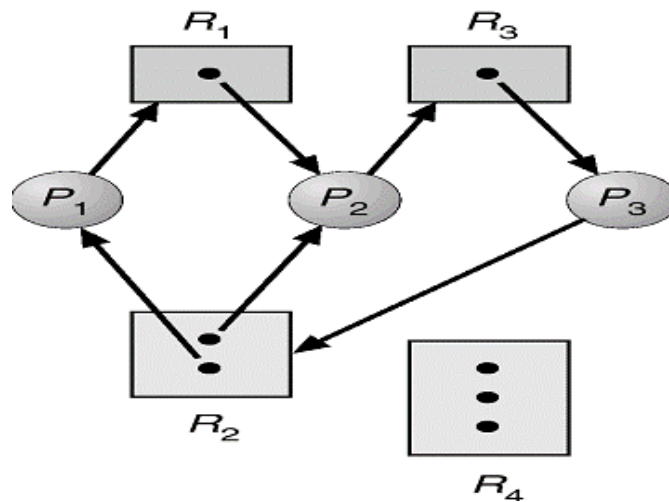

Figure B: Resource-allocation graph with a deadlock.

Now consider the resource-allocation graph in Figure C. In this example, we also have a cycle:
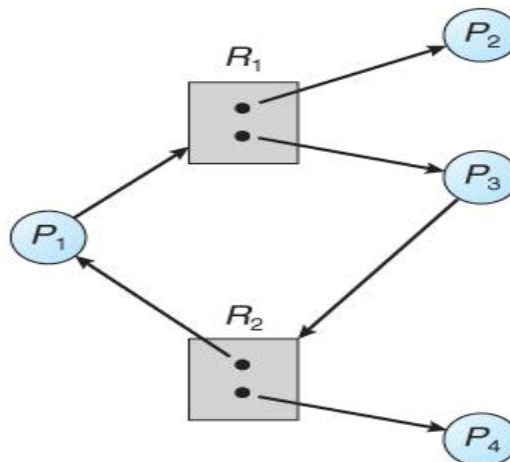
P1 ⟼ R1 ⟶ P3 ⟶ R2 ⟶ P1

Figure C: Resource-allocation graph with a cycle but no deadlock.

- However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

- In summary, if a resource-allocation graph does not have a cycle, then the system is <u>not in a deadlocked state.</u>

- If there is a cycle, then the system may or may not be in a deadlocked state.

- This observation is important when we deal with the deadlock problem.

5

Methods of handling Deadlock

Generally speaking, we can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
2. We can allow the system to enter a deadlocked state, detect it, and recover.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

Deadlock Prevention

❖ Deadlock provides a set of methods for ensuring that at least **one of the necessary conditions cannot hold**. We elaborate on this approach by examining each of the four necessary conditions separately.

1. **Mutual Exclusion:**

➢ The mutual-exclusion condition must hold for non-sharable resources.

➢ For example, a printer cannot be simultaneously shared by several processes.

➢ Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.

➢ Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

➢ A process never needs to wait for a sharable resource.

➢ In general, however, we cannot prevent Deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

2. **Hold and Wait:**

➢ To ensure that the **hold-and-wait condition never occurs in the system**, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

➢ One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.

➢ An alternative protocol allows a process to request resources only when it has none. A process may request some resources and use them.

➢ Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

Example: To illustrate the difference between these two protocols, we consider --

- **First method**, a process that copies data from <u>a DVD drive to a file on disk, sorts the file, and then prints the results to a printer.</u>

- If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer.

- It will hold the printer for its entire execution, even though it needs the printer only at the end.

- **The second method** allows the process to request initially only the DVD drive and disk file.

- It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file.

- The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

➢ Both these protocols have two main disadvantages.

- **First,** resources may be allocated but unused for a long period. In the example we can release the DVD drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. Otherwise, we must request all resources at the beginning for both protocols.

- **Second,** starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3. **No Preemption:**

➢ The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that this condition **does not hold,** we can use the following protocol.

➢ If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.

➢ Alternatively, if a process requests some resources, we first <u>check whether they are available</u>. If they are, we allocate them. If they are not, we <u>check whether they are allocated to some other process</u> that is waiting for additional resources.

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

➤ If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

➤ If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them.

➤ A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

4. **Circular Wait:**

➤ The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this **condition never holds**.

➤ **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1.

➤ P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock Avoidance:

The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. Such an algorithm defines the deadlock-avoidance approach.

I. Safe state:

- A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.

- A sequence of processes <P1, P2, ... , Pn> is a safe sequence for the current allocation state if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i.

- In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished.

- When they have finished, Pi can obtain all of its needed resources, complete it, and return its allocated resources, and terminate.

- If no such sequence exists, then the system state is said to be unsafe.

- A safe state is not a deadlocked state. Not all unsafe states are deadlocks, however (Figure D).

8

- An unsafe state may lead to a deadlock. As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.
- In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs.
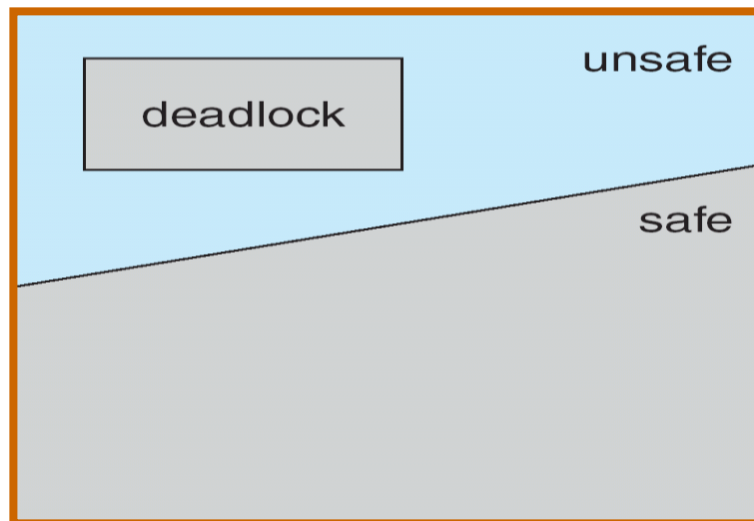

Figure D: Safe, unsafe, and deadlocked state spaces.

To illustrate, we consider a system with twelve magnetic tape drives and three processes: P0, P1, and P2.

- Process P0 requires ten tape drives, process P1 may need as many as four tape drives.
- Process P2 may need up to nine tape drives.
- Suppose that, at time $t0$, process P0 is holding five tape drives, process P1 is holding two tape drives, and process P2 is holding two tape drives. (Thus, there are three free tape drives.)

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

➢ At time $t0$, the system is in a safe state. The sequence **<P1, P0, P2>** satisfies the safety condition.

➢ Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives).

➢ Then process P0 can get all its tape drives and return them (the system will then have ten available tape drives). Finally process P2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

9

➢ Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state.

➢ Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

II. The resource-allocation-graph :

- Observe The resource-allocation-graph (above mentioned topic), In addition to the **request** and **assignment** edges already described, we introduce a new type of edge, called a **claim edge.**

- A claim edge $P_i \longrightarrow R_j$ indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.

- When process Pi requests resource R1, the claim edge $P_i \longrightarrow R_j$ is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge $R_j \longrightarrow P_i$ is reconverted to a claim edge $P_i \longrightarrow R_j$.

The resource-allocation-graph algorithm:is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme. This algorithm is commonly known as the banker's algorithm
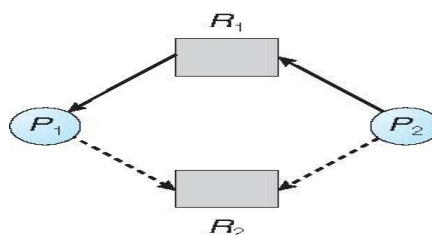
Resource-Allocation Graph

III. Banker's Algorithm:

❖ The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

❖ When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

❖ When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement the banker's algorithm. We need the following data structures, where n is the number of processes in the system and m is the number of resource types:

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

  If Available[j] equals k, then k instances of resource type Rj are available.

- **Max:** An n x m matrix defines the maximum demand of each process.

  If Max[i] [j] equals k, then process Pi may request at most k instances of resource type Rj.

- **Allocation:** An n x m matrix defines the number of resources of each type currently allocated to each process.

  If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.

- **Need:** An n x m matrix indicates the remaining resource need of each process.

 If Need [i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task.

Note that Need[i][j] equals Max[i][j] - Allocation [i][j].

 a) **Safety Algorithm:**

   We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize

   Work= Available and Finish[i] = false for i = 0, 1, ... , n - 1.

2. Find an index i such that both

         a. Finish[i] = =false

         b. $Need_i \leq Work$

   If no such i exists, go to step 4.

3. Work = Work + Allocation;

   Finish[i] = true

   Go to step 2.

4. If Finish[i] = = true for all i, then the system is in a safe state.

11

This algorithm may require an order of m x n2 operations to determine whether a state is safe.

b)   Resource-Request Algorithm:

Next, we describe the algorithm for determining whether requests can be safely granted.

Let $Request_i$ be the request vector for process P_i. If $Request_i$ [j] = = k, then process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi , the following actions are taken:

1. If **Request$_i$ ≤ Need$_i$**, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If **Request$_i$ ≤ Available**, go to step 3. Otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifyil1.g the state as follows:

    Available= Available - $Request_i$;

    $Allocation_i$ =$Allocation_i$ + $Request_i$;

    $Need_i$ =$Need_i$ - $Request_i$;

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then Pi must wait for $Request_i$, and the old resource-allocation state is restored.

Illustrative Example:

To illustrate the use of the banker's algorithm,

Consider a system with five processes Po through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T0 , **the following snapshot of the system has been taken**:

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

The content of the matrix Need is defined to be Maximum – Allocation.

Need = Maximum - Allocation

12

Need

A B C

P0   7 4 3

P1   1 2 2

P2   6 0 0

P3   0 1 1

P4   4 3 1

We need to find the safety sequence:

P0: $Need_0 \leq Work \Rightarrow (7,4,3) \leq (3,3,2)$; Finish[0] = false

P1: $Need_1 \leq Work \Rightarrow (1,2,2) \leq (3,3,2)$; Finish[1] = True then

Work = Work + Allocation

Work = (3, 3, 2) + (2,0,0) = (5,3,2)

P2: $Need_2 \leq Work \Rightarrow (6,0,0) \leq (5,3,2)$; Finish[2] = false

P3: $Need_3 \leq Work \Rightarrow (0, 1, 1) \leq (5, 3, 2)$; Finish[3] = True then

Work = Work + Allocation

Work = (5, 3, 2) + (2, 1, 1) = (7, 4, 3)

P4: $Need_4 \leq Work \Rightarrow (4, 3, 1) \leq (7, 4, 3)$; Finish [4] = True then

Work = Work + Allocation

Work = (7, 4, 3) + (0, 0, 2) = (7, 4, 5)

P0: $Need_0 \leq Work \Rightarrow (7, 4, 3) \leq (7, 4, 5)$; Finish [0] = True then

Work = Work + Allocation

Work = (7, 4, 5) + (0, 1, 0) = (7, 5, 5)

P2: $Need_2 \leq Work \Rightarrow (6, 0, 0) \leq (7, 5, 5)$; Finish [2] = True then

Work = Work + Allocation

Work = (7, 5, 5) + (3, 0, 2) = (10, 5, 7).

- ❖ We claim that the system is currently in a safe state. Indeed, the Sequence <**P1, P3, P4, P0, P2>** satisfies the safety criteria.

- ❖ **Suppose** now that process P1 requests one additional instance of resource type A and two instances of resource type C, so Request1 = (1,0,2).

- ❖ **To decide** whether this request can be immediately granted, we first check that **Request$_1$ ≤ Available,** that is, that $(1,0,2) \leq (3,3,2)$, which is true.

- ❖ We then pretend that this request has been fulfilled, and we arrive at the following new state:

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

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

❖ We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence **<P1, P3, P4, P0, P2>** satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

❖ However, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available.

i.e. we first check that **Request4 ≤ Available,** that is, that $(3,3,0) \leq (2,3,0)$, which is false.

❖ Furthermore, a request for (0,2,0) by P0 cannot be granted, even though the **resources are available**, since the resulting **state is unsafe.**

i.e we first check that **Request0 ≤ Available,** that is, that $(0,2,0) \leq (2,3,0)$, which is true. But we don't have a safety sequence if we grant the resources for P0 immediately.

Example 2: Consider the following snapshot of a system:

	Allocation	Max
	A B C D	A B C D
P0	3 0 1 4	5 1 1 7
P1	2 2 1 0	3 2 1 1
P2	3 1 2 1	3 3 2 1
P3	0 5 1 0	4 6 1 2
P4	4 2 1 2	6 3 2 5

Using the banker's algorithm, determine whether or not each of the following states is unsafe. If the state is safe, illustrate the order in which the processes may complete. Otherwise, illustrate why the state is unsafe.

a. **Available** = (0, 3, 0, 1)

b. **Available** = (1, 0, 0, 2)

Example 3: Consider the following snapshot of a system:

	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	2 0 0 1	4 2 1 2	3 3 2 1
P1	3 1 2 1	5 2 5 2	
P2	2 1 0 3	2 3 1 6	
P3	1 3 1 2	1 4 2 4	
P4	1 4 3 2	3 6 6 5	

Answer the following questions using the banker's algorithm:

a. Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.

b. If a request from process P1 arrives for (1, 1, 0, 0), can the request be granted immediately?

c. If a request from process P4 arrives for (0, 0, 2, 0), can the request be granted immediately?

15

Example 4: Consider the following snapshot of a system:

	Allocation ABCD	*Max* ABCD	*Available* ABCD
Po	0012	0012	1520
Pl	1000	1750	
P2	1354	2356	
P3	0632	0652	
P4	0014	0656	

Answer the following questions using the banker's algorithm:

    a. What is the content of the matrix Need?

    b. Is the system in a safe state?

    c. If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

Deadlock Detection:

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide:

❖ An algorithm that examines the state of the system to determine whether a deadlock has occurred

❖ An algorithm to recover from the deadlock

1. Single Instance of Each Resource Type:

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, **called a wait-for graph**. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

❖ More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.

❖ An edge Pi → Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi → Rq and Rq → Pj for some resource Rq . In Figure 5-A, we present a resource-allocation graph and the corresponding wait-for graph.
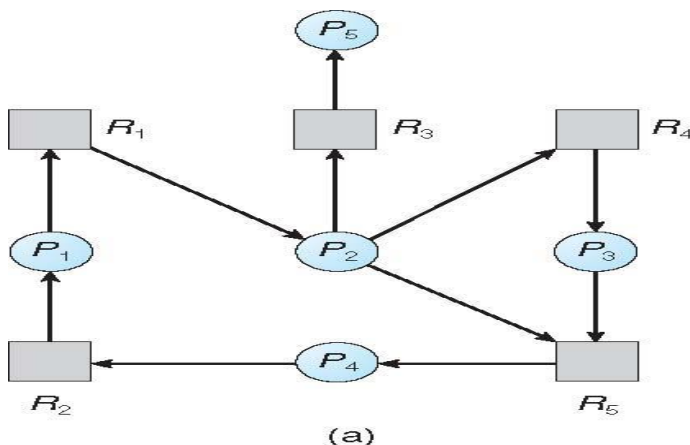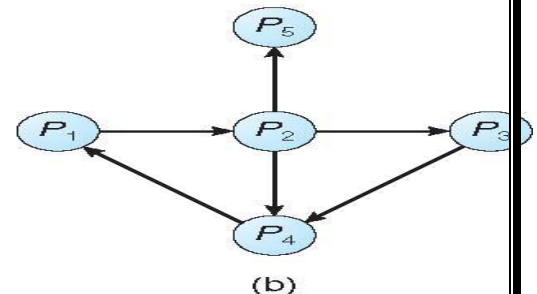
(a) Resource-allocation graph.   **(b) Corresponding wait-for graph**

Figure 5-E

2. Several Instances of a Resource Type:

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type. We turn now to a deadlock detection algorithm that is applicable to such a system.

Deadlock Detection Algorithm:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize

   Work = Available. For i = 0, 1, ..., n–1, if $Allocation_i \neq 0$, then

   Finish[i] = false. Otherwise, Finish[i] = true.

2. Find an index i such that both

   a. Finish[i] == false

   b. $Request_i \leq Work$

   If no such i exists, go to step 4.

3. Work = Work + $Allocation_i$

   a. Finish[i] = true

   b. Go to step 2.

4. If Finish [i] = = false for some i, 0≤i<n, then the system is in a deadlocked state.

   Moreover, if Finish[i] = = false, then process Pi is deadlocked.

> You may wonder why we reclaim the resources of process Pi (in step 3) as soon as we determine that $Request_i \leq Work$ (in step 2b). We know that Pi is currently not involved in a deadlock (since $Request_i \leq Work$). Thus, we take an optimistic attitude and assume that Pi will require no more resources to complete its task; it will thus soon return all currently allocated resources to the system. If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

To illustrate this algorithm, we consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T0, we have the following resource-allocation state:

	Allocation	Request	Available
	A B C	A B C	A B C
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

❖ We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence **<P0, P2, P3, P1, P4>** results in Finish[i] = = true for all i.

❖ Suppose now that process P2 makes one additional request for an instance of type C. The Request matrix is modified as follows:

Request

	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

❖ We claim that the system is now deadlocked. Although we can reclaim the resources held by process P0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, P2, P3, and P4.

Detection-Algorithm Usage:

When should we invoke the detection algorithm? The answer depends on two factors:

1. How *often* is a deadlock likely to occur?

2. How *many* processes will be affected by deadlock when it happens?

If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.

Recovery from Deadlock:

When a detection algorithm determines that a deadlock exists, several alternatives are available.

- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system recover from the deadlock automatically.

❖ There are two options for breaking a deadlock.

➢ One is simply to abort one or more processes to break the circular wait.
➢ The other is to preempt some resources from one or more of the deadlocked processes**.**

1. Process Termination:

To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

a) Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at great expense. The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

19

b) Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Many factors may affect which process is chosen, including:

I. What the priority of the process is

II. How long the process has computed and how much longer the process will compute before completing its designated task

III. How many and what types of resources the process has used (for example, whether the resources are simple to preempt)

IV. How many more resources the process needs in order to complete

V. How many processes will need to be terminated

VI. Whether the process is interactive or batch.

2. Resource Preemption:

❖ To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim.** Which resources and which processes are to be preempted?
   - As in process termination, we must determine the order of preemption to minimize cost.
   - Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

2. **Rollback**. If we preempt a resource from a process, what should be done with that process?
   - Clearly, it cannot continue with its normal execution; it is missing some needed resource.
   - We must roll back the process to some safe state and restart it from that state.

   > Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.

3. **Starvation.** How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

20

- In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim.
- As a result, this process never completes its designated task, a starvation situation any practical system must address.
- Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times.
- The most common solution is to include the number of rollbacks in the cost factor.

Process Synchronization

Process Synchronization means sharing system resources by processes in a such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

Process Synchronization was introduced to handle problems that arose while multiple process executions. Some of the problems are discussed below.

Critical Section Problem

➢ Consider a system consisting of n processes {Po, P1, ..., Pn-1 }. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.

➢ The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

➢ That is, no two processes are executing in their critical sections at the same time.

➢ Each process must request permission to enter its critical section. The section of code implementing this request is the entry section.

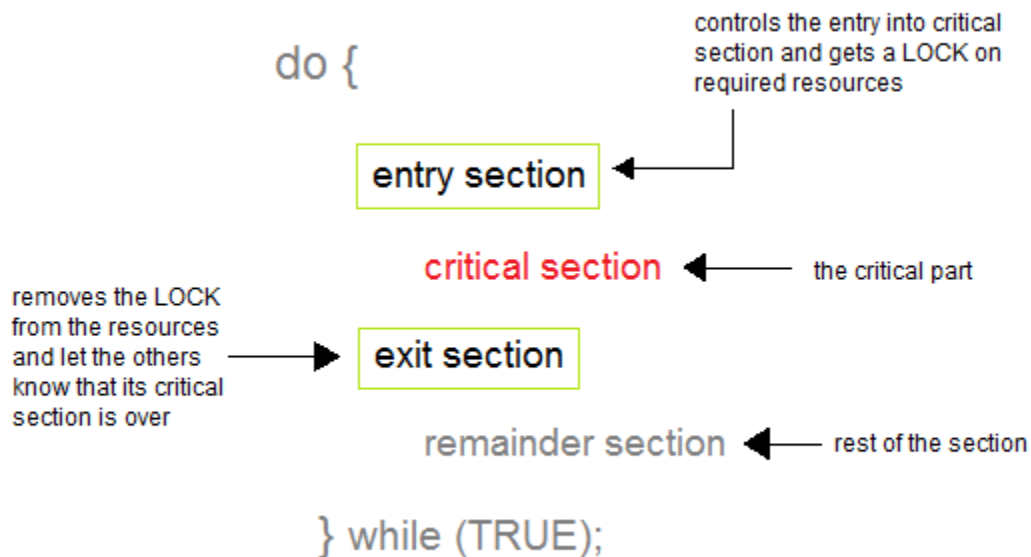➢ The critical section may be followed by an exit section The remaining code is the remainder section

Figure : General Structure of a typical process

Solution to Critical Section Problem

A solution to the critical section problem must satisfy the following three conditions:

1. **Mutual Exclusion**

Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. **Progress**

If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

3. **Bounded Waiting**

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.

Peterson's solution:

- A classic software-based solution to the critical-section problem known as Peterson's solution.

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *Po* and P1. For convenience, when presenting *Pi,* we use *Pj* to denote the other process; that is, j equals 1 - i.

Peterson's solution requires the two processes to share two data items:

int turn;

boolean flag[2];

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn = = i, then process Pi is allowed to execute in its critical section.

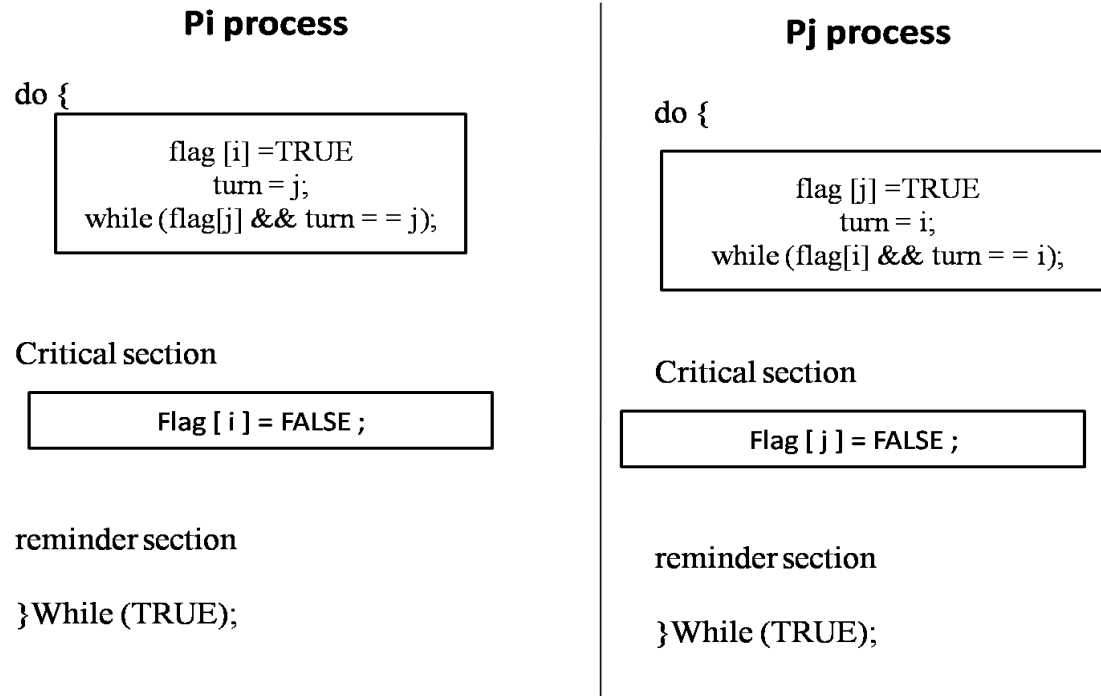The flag array is used to indicate if a process is ready to enter its critical section.

For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.

To enter the critical section, process *Pi* first sets flag [i] to be true and then sets turn to the value j,

Thereby asserting that if the other process wishes to enter the critical section, it can do so.

 If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

Only one of these assignments will last; the other will occur but will be overwritten immediately.

23

Pi process	**Pj process**
do {	do {

Pi process:

do {

```
flag [i] =TRUE
turn = j;
while (flag[j] && turn = = j);
```

Critical section

```
Flag [ i ] = FALSE ;
```

reminder section

}While (TRUE);

Pj process:

do {

```
flag [j] =TRUE
turn = i;
while (flag[i] && turn = = i);
```

Critical section

```
Flag [ j ] = FALSE ;
```

reminder section

}While (TRUE);

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We need to show that:

- ✓ Mutual exclusion is preserved.
- ✓ The progress requirement is satisfied.
- ✓ The bounded-waiting requirement is met.

- To prove property 1, we note that each *Pi* enters its critical section only if either flag [j] = = false or turn = = i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [0] = = flag [1] = =true.

- These two observations imply that Po and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.

- Hence, one of the processes-say, *Pi* -must have successfully executed the while statement, whereas *Pi* had to execute at least one additional statement ("turn = = j").

- However, at that time, flag [j] = = true and turn = = j, and this condition will persist as long as Pi is in its critical section; as a result, **mutual exclusion is preserved.**

- To prove properties 2 and 3, we note that a process *P;* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] = = true and turn= = j; this loop is the only one possible.

- If Pj is not ready to enter the critical section, then flag [j] = = false, and Pi can enter its critical section. If P_j has set flag [j] to true and is also executing in its while statement, then either turn = = i or turn = = j.

24

- If turn = = i, then Pi will enter the critical section. If turn = = j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag [j] to false, allowing Pi to enter its critical section

- If P_j resets flag [j] to true, it must also set turn to i.

- Thus, since P_i does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

Synchronization Hardware:

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a **lock**.

Race conditions are prevented by requiring that critical regions be protected by locks. That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

do {

> Acquire lock

Critical section

> Release lock

remainder section

} while (TRUE);

> Figure: Solution to the critical-section problem using locks

Semaphores:

- The hardware-based solutions to the critical-section problems are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

Semaphores are of two types:

- ✓ Binary semaphore
- ✓ Counting semaphore

- Binary semaphore can take the value 0 & 1 only. Counting semaphore can take nonnegative integer values.

- Two standard operations, wait and signal are defined on the semaphore. Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care by signal operation. The wait, signal operations are also called P and V operations. The manipulation of semaphore (S) takes place as following:

  1. The wait command P(S) decrements the semaphore value by 1. If the resulting value becomes negative then P command is delayed until the condition is satisfied.
  2. The V(S) i.e. signals operation increments the semaphore value by 1.

Mutual exclusion on the semaphore is enforced within P(S) and V(S). If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting. These operations are defined as under-

P(S) or wait(S):

If S > 0 then

  Set S to S-1

Else

  Block the calling process (i.e. Wait on S)

V(S) or signal(S):

If any processes are waiting on S

  Start one of these processes

Else

  Set S to S+1

- The semaphore operation are implemented as operating system services and so wait and signal are atomic in nature i.e. once started, execution of these operations cannot be interrupted.
- Thus semaphore is a simple yet powerful mechanism to ensure mutual exclusion among concurrent processes.

26

Classic Problems of Synchronization

- We present a number of synchronization problems as examples of a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme.

1. The Bounded-Buffer(also called the Producer-Consumer) Problem:
2. The Readers-Writers Problem.
3. The Dining-Philosophers Problem

1. The Bounded-Buffer(Producer-Consumer) Problem:

➢ We assume that the pool consists of *n* buffers, each capable of holding one item.
➢ The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
➢ The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
   ✓ A producer cannot produce unless there is an empty buffer slot to fill.
   ✓ A consumer cannot consume unless there is at least one produced item.

Semaphore empty=N, full=0, mutex=1;

```
do {
// produce an item in nextp
wait(empty);
wait(mutex);
// add nextp to buffer
signal(mutex);
signal(full);
} while (TRUE);
```
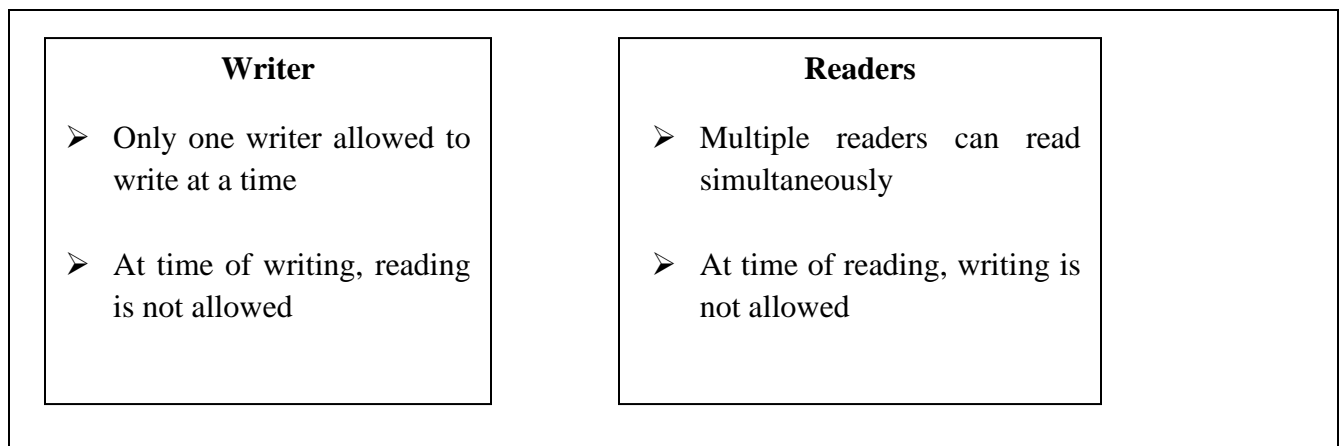
```
do {
wait (full);
wait (mutex) ;
// remove an item from buffer to nextc
signal(mutex);
signal(empty);
// consume the item in nextc
} while (TRUE);
```

CONCEPT: Producers produce items to be stored in the buffer. Consumers remove and consume items which have been stored. Mutual exclusion must be enforced on the buffer itself. Moreover, producers can store only when there is an empty slot, and consumers can remove only when there is a full slot.

27

2. The Readers-Writers Problem:

- Suppose that a database is to be shared among several concurrent processes.

- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.

- Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos (confusion) may ensue.

- ❖ To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers-writers problem.**

- ➢ The simplest one, referred to as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.

- ➢ The second readers - writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

Writer	Readers
➢ Only one writer allowed to write at a time	➢ Multiple readers can read simultaneously
➢ At time of writing, reading is not allowed	➢ At time of reading, writing is not allowed

- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

    semaphore mutex, write;

     int readcount;

- The semaphores mutex and write are initialized to 1;
- readcount is initialized to 0.
- The semaphore write is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore write functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is shown here   The code for a reader process is shown here

```
do {
wait(write);
// writing is performed
signal(write);
} while (TRUE);
```

Fig : The structure of a writer process

```
do {
wait (mutex);
readcount++;
if (readcount 1)
wait (wrt);
signal(mutex);
// reading is performed
wait(mutex);
readcount--;
if (readcount 0)
signal(wrt);
signal(mutex);
} while (TRUE);
```

- The readers-writers problem and its solutions have been generalized to provide locks on some systems. Acquiring a reader-writer lock requires specifying the mode of the lock either read or write access.
- When a process wishes only to read shared data, it requests the reader-writer lock in read mode;
a process wishing to modify the shared data must request the lock in write mode.

29

- Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader-writer locks are most useful in the following situations:

> - In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
> - In applications that have more readers than writers. This is because reader -writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader-writer lock

3. The Dining-Philosophers Problem:

- Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (forks). When a philosopher thinks, he does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him.

- A philosopher may pick up only one chopstick at a time. Obviously, he cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he is finished eating, he puts down both of his chopsticks and starts thinking again.

Fig: The situation of the dining philosophers.

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

The dining-philosophers problem is considered a classic synchronization problem .it is an example of a large class of concurrency-control problems.

- Suppose that all five philosophers become hungry simultaneously and each grabs his left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab his right chopstick, he will be delayed forever. **This is deadlock problem.**

Deadlock   **Starvation**

- Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat.
- In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. **This is a starvation problem**.

- Note that it is not a deadlock because there is no circular waiting, and everyone has a chance to eat!
Several possible remedies to the deadlock problem are listed next.
- ➢ Allow at most four philosophers to be sitting simultaneously at the table.
- ➢ Allow a philosopher to pick up his chopsticks only if both chopsticks are available (to do this, he must pick them up in a critical section).
- ➢ Use an asymmetric solution; that is, an odd philosopher picks up first hid left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Dining-Philosophers Solution Using semaphores

❖ <u>One simple solution is to represent</u> each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; he releases his chopsticks by executing the signal () operation on the appropriate semaphores.

Thus, the shared data are

semaphore chopstick[5];

Where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown below.

```
do {

wait (chopstick[i]);

wait(chopstick[(i+l) % 5]);

// eat

Signal (chopstick[i]);

Signal (chopstick[(i+l) % 5]);

// think

} while (TRUE);
```

Fig: The structure of philosopher *i*.

Monitors

<u>A monitor is a set of multiple routines</u> which are protected by <u>a mutual exclusion lock</u>. None of the routines in the monitor can be executed by a thread until that thread acquires the lock. This means that only ONE thread can execute within the monitor at a time. Any other threads must wait for the thread that's currently executing to give up control of the lock.

What is semaphore

➢ A semaphore is a simpler construct than a monitor because it's just a lock that protects a shared resource – and not a set of routines like a monitor. The application must acquire the lock before using that shared resource protected by a semaphore.

➢ A semaphore 'S' is a synchronization tool which is an integer value that, apart from initialization, is accessed only through two standard atomic operations; wait and signal. Semaphores can be used to deal with the n process critical section problem. It can be also used to solve various synchronization problems.

32

Differences between Monitors and Semaphores:

➢ Both <u>Monitors and Semaphores are</u> used for the same purpose – thread synchronization.

➢ But, monitors are simpler to use than semaphores because they handle all of the details of lock acquisition and release.

➢ An application using semaphores has to release any locks a thread has acquired when the application terminates – this must be done by the application itself. If the application does not do this, then any other thread that needs the shared resource will not be able to proceed.

Is there a cost to using a monitor or semaphore?

Yes, there is a cost associated with using synchronization constructs like monitors and semaphores. And, this cost is the time that is required to get the necessary locks whenever a shared resource is accessed.

Structure of monitor :A monitor has four components as shown below:

Initialization, private data, monitors procedures, and monitor entry queue.

➢ The initialization component contains the code that is used exactly once when the monitor is created.

➢ The private data section contains all private data, including private procedures that can only be used within the monitor.

➢ Thus, these private items are not visible from outside of the monitor.

➢ The monitor procedures are procedures that can be called from outside of the monitor.

➢ The monitor entry queue contains all threads that called monitor procedures but have not been granted permissions. We shall return to this soon.

Dining-Philosophers Solution Using Monitors

❖ We illustrate monitor concepts by presenting <u>a deadlock-free solution</u> to the dining-philosophers problem.

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher.

For this purpose, we introduce the **following data structure**:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher i can set the variable state [i] = EATING only if his two neighbors are <u>not eating</u>:

(state [(i +4) % 5]! = EATING) and (state [(i +1) % 5] != EATING).

We also need to declare,   condition self [5];

in which philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.

We are now in a position <u>to describe our solution to the dining-philosophers problem.</u>

- The distribution of the chopsticks is controlled by the monitor Dining Philosophers. Each philosopher, before starting to eat, must invoke the operation <u>pickup ( ).</u>

```
monitor dp
{
      enum {THINKING, HUNGRY, EATING} state[5];
      condition self[5];
      void  pickup (int i)
      {
            state[i] =HUNGRY;
            test (i);
            if (state [i] ! = EATING)
            self [i]. wait ( );
      }
      void  putdown (int i)
      {
            state[i] =THINKING;
```

34

```
            test((i + 4) % 5);
            test((i + 1) % 5);
    }
    void test(int i)
     {
  if ((state[(i + 4) % 5] !=EATING) && (state[i] = =HUNGRY) &&  (state[(i + 1)%5]!=EATING)
    {
    state[i] =EATING;
    self[i] .signal();
    }
    }
    initialization_code( )
    {
    for (int i = 0; i < 5; i++)
    state[i] =THINKING;
    }
}
```

- This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the put down ( ) operation.

- Thus, philosopher i must invoke the operations pickup ( ) and put down ( ) in the following sequence:

<div align="center">

DiningPhilosophers.pickup (i);

eat

DiningPhilosophers.putdown (i);

</div>

- It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that **no deadlocks will occur.**

We note, however, that it is possible for a philosopher to starve to death. We do not present a solution to this problem.

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

Inter process communication

Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

Communication can be of two types −

- Between related processes initiating from only one process, such as parent and child processes.

- Between unrelated processes, or two or more different processes

Following are some important terms that we need to know before proceeding further on this topic.

➢ **Pipes** − Communication between two related processes. The mechanism is half duplex meaning the first process communicates with the second process. To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.

➢ **FIFO** − Communication between two unrelated processes. FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

➢ **Message Queues** − Communication between two or more processes with full duplex capacity. The processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, the message is no longer available in the queue.

➢ **Shared Memory** − Communication between two or more processes is achieved through a shared piece of memory among all processes. The shared memory needs to be protected from each other by synchronizing access to all the processes.

➢ **Semaphores** − Semaphores are meant for synchronizing access to multiple processes. When one process wants to access the memory (for reading or writing), it needs to be locked (or protected) and released when the access is removed. This needs to be repeated by all the processes to secure data.

➢ **Signals** − Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.

IPC between processes on a Single System:

It includes four general approaches:

- ➢ Shared memory
- ➢ Messages
- ➢ Pipes
- ➢ Sockets

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

Pipes-creation and IPC between Related Processes using unnamed pipes/Ordinary pipes

❖ Pipe is used to combine two or more command and in this the output of one command act as input to another command and this command output may act as input to next command and so on.You can make it do so by using the pipe character '**|**'.

❖ Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between **related processes** (inter-process communication).

➢ When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.
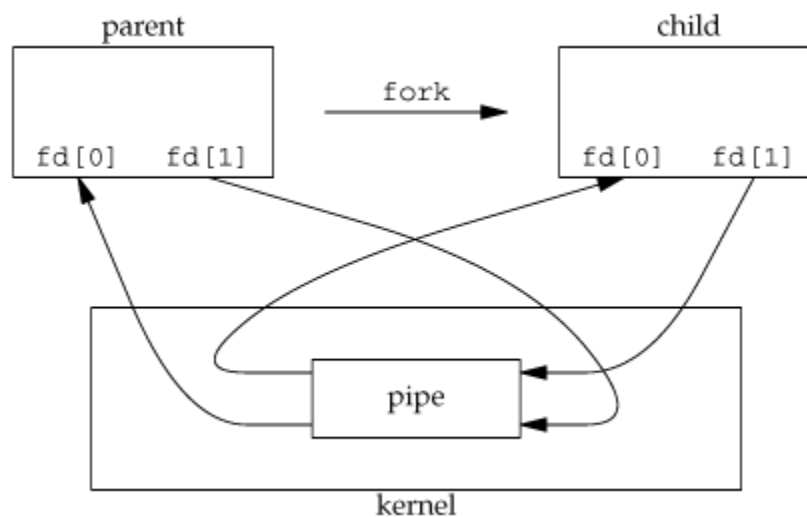
pipe( ) - create pipe

<u>**Synopsis:**</u>

> **#include <unistd.h>**
> **int pipe(int** filedes**[2]);**

Description:

pipe( ) creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes:

✓ filedes[0] is for reading,

✓ filedes[1] is for writing.

The following rules apply to processes that read from a pipe:

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

- ➢ If a process reads from a pipe whose write end has been closed, the read ( ) returns a 0, indicating end-of-input.
- ➢ If a process reads from an empty pipe whose write end is still open, it sleeps until some input becomes available.
- ➢ If a process tries to read more bytes from a pipe than are present, all of the current contents are returned and read ( ) returns the number of bytes actually read.

The following rules apply to processes that write to a pipe:

- ➢ If a process writes to a pipe whose read end has been closed, the write fails and the writer is sent a SIGPIPE signal.
  - ✓ The default action of this signal is to terminate the writer.
- ➢ If a process writes fewer bytes to a pipe than the pipe can hold, the write ( ) is guaranteed to be atomic; that is, the writer process will complete its system call without being preempted by another process.
- ➢ If a process writes more bytes to a pipe than the pipe can hold, no similar guarantees of atomicity apply.

Return Value:

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

Example: pipe program for ordinary pipe or unnamed pipe

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
int main(int argc,char *argv[])
{
int fd[2],pid,k;
k=pipe(fd);
if(k==-1)
{
perror("pipe");
exit(1);
}
pid=fork();
if(pid==0)
{
close(fd[0]);
dup2(fd[1],1);
```
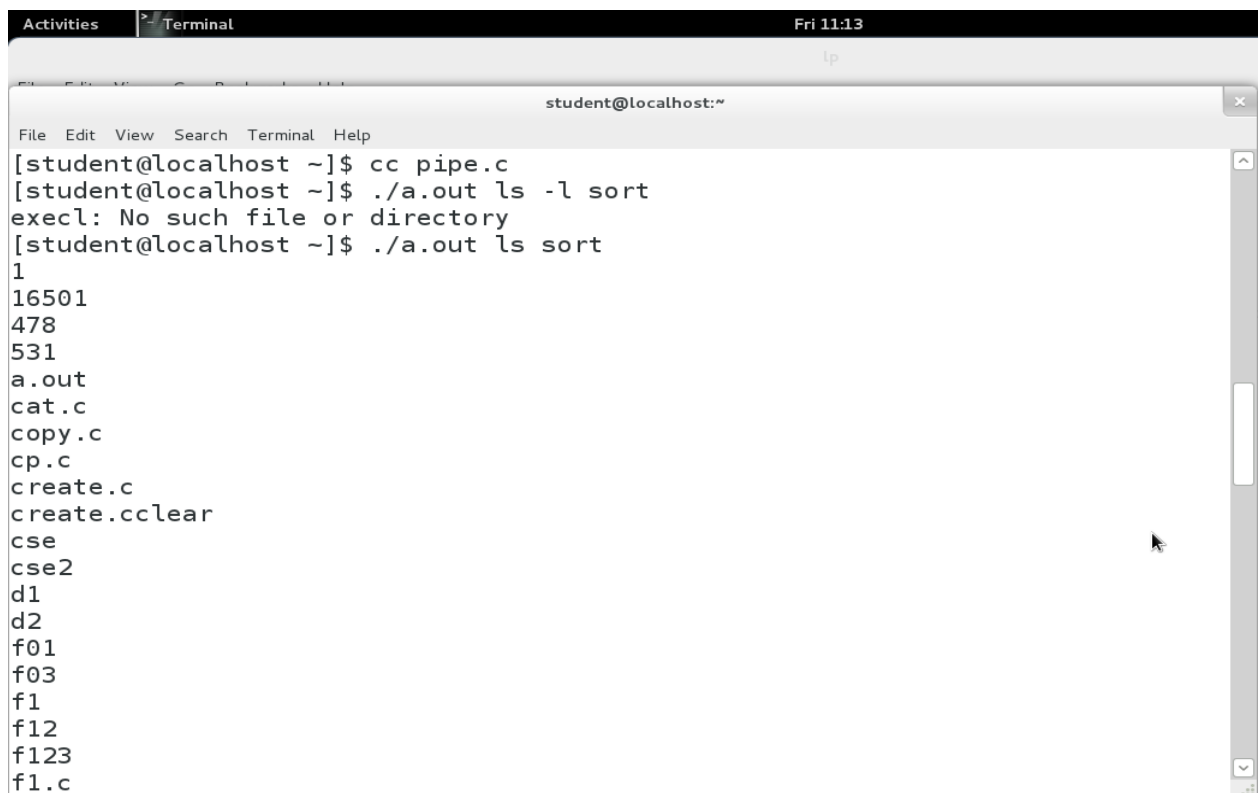
```
close(fd[1]);
execlp(argv[1],argv[1],NULL);

perror("execl");
}
else
{
wait(2);
close(fd[1]);
dup2(fd[0],0);
close(fd[0]);
execlp(argv[2],argv[2],NULL);
perror("execl");
}
}
```

OUTPUT:

❖ <u>One of the major disadvantages of pipes is that</u> the they cannot be accessed using their names by any other process other than child and the parent as they do not get listed in the directory tree.

❖ The work around for this problem is to create a named pipe which is also called as a FIFO, which stands for First in First out, meaning the data that is written into the pipe first will be read out first always

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

FIFOs – Creation and IPC between unrelated Processes using Named Pipes:

> ➢ FIFOs are created using the function **mkfifo( ).** Which takes as arguments.

>> ✓ The name of the fifo that has to be created

>> ✓ The permissions for the file.

> ➢ The FIFOs get listed in the directory tree and any process can access it using its name by providing the appropriate path.

❖ **mkfifo ( )** - make a FIFO special file (a named pipe)

> ➢ Here i want create two named pipe files or special files using mkfifo system call or command.

cse@nnrg] $ mkfifo fifo_server

cse@nnrg] $ mkfifo fifo_client

❖ Now to use these pipes, we will write two program's one to represent the server and other to represent the client. The server will read from the pipe fifo_server to which the client will send a request. On recieving the request, the server will send the information to the client on fifo_client.

server.c

```c
#include<stdio.h>
#include<fcntl.h>
main( )
{
FILE *file1;
int fds,fdc;
int choice;
char *buf;
fds = open("fifo_server",O_RDWR);
if(fds<1) {
 printf("Error opening file");
 }
read(fds,&choice,sizeof(int));
sleep(10);
fdc = open("fifo_client",O_RDWR);

if(fdc<1) {
 printf("Error opening file");
 }
switch(choice) {

case 1:
 buf="Linux";
 write(fifo_client,buf,10*sizeof(char));
```

40

```
 printf("\n Data sent to client \n");
 break;
case 2:

 buf="Fedora";
 write(fifo_client,buf,10*sizeof(char));
 printf("\nData sent to client\n");
 break;
case 3:
 buf="2.6.32";
 write(fifo_client,buf,10*sizeof(char));
 printf("\nData sent to client\n");
}

close(fds);
close(fdc);
}
```

10*sizeof(int)

To allocate block of memory dynamically.

sizeof is greatly used in dynamic memory allocation. For example, if we want to allocate memory for which is sufficient to hold 10 integers and we don't know the sizeof(int) in that particular machine. We can allocate with the help of sizeof.

➢ The above code,server.c, reads the choice from fifo_server to which the client writes and depending on the request,the server responds with the relevant data by writing to fifo_client.

Save the file and server.c and compile it as follows

```
$cc server.c
```

client.c:

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
main()
{
 FILE *file1;
 int fds,fdc;
 char str[256];
 char *buf;
 int choice=1;
 printf("Choose the request to be sent to server from options below");
 printf("\n\t\t Enter 1 for O.S.Name \n \
          Enter 2 for Distribution \n \
          Enter 3 for Kernel version \n");
 scanf("%d",&choice);

 fds=open("fifo_server",O_RDWR);
 if(fds < 0) {
 printf("Error in opening file");
 exit(-1);
 }

 write(fds,&choice,sizeof(int));
```

41

```
fdc=open("fifo_client",O_RDWR);

if(fdc< 0) {
 printf("Error in opening file");
 exit(-1);
 }

buf=malloc(10*sizeof(char));
read (fifo_client,buf,10*sizeof(char));
printf("\n ***Reply from server is %s***\n",buf);
close(fds);
close(fdc);
}
```

➢ The above code,client.c, sends a request to the server by writing to the pipe fifo_server, and recieves
the reply from the server by reading the pipe fifo_client.

Save the file and client.c and compile it as follows

```
$cc client.c
```

➢ To see the pipe in operation, open two terminals and go the folder where the pipes have been
created.

Note: The server.c and client.c codes assume that the pipes exist in the same directory as the
executables of server and client are, if they exist in any other directory you will have to provide the
path to the same in the system call open().

Terminal 1:

```
. / server
```

The terminal will go into a wait state with the cursor blinking, waiting for request from client.

Terminal 2:

```
. / clientt
```

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

The client will prompt to make a choice of request to be sent to the server, enter number 1,2 or 3

```
Choose the request to be sent to server from options below
   Enter 1 for O.S.Name
              Enter 2.Distribution
              Enter 3.Kernel version
```
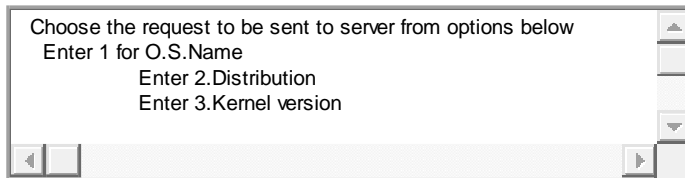
➢ This number will be sent to the server and the client will go into a wait state, waiting for the server to respond.

➢ After a few seconds you should see the reponse from the server being printed on the client terminal.
  Note: wait for 10 seconds for the output to appear as the server has a sleep for 10 seconds, this is required to allow some time for the client to start its read operation on the fifo_client pipe.

  ❖ **For eg:** if we entered option 2, the response would be

    output on **client terminal**:

```
***Reply from server is Fedora***
```
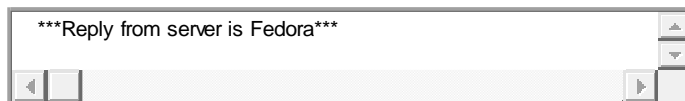
    Ouput on **server terminal** :

```
Data sent to client
```

  ❖ Thus we were able to communicate between the two processes using the FIFOs, even though the pipes were not created by them.

The major differences between named and unnamed pipes are:-

1. As suggested by their names, a named type has a specific name which can be given to it by the user. Named pipe if referred through this name only by the reader and writer. All instances of a named pipe share the same pipe name.
   On the other hand, unnamed pipes is not given a name. It is accessible through two file descriptors that are created through the function pipe(fd[2]), where fd[1] signifies the write file descriptor, and fd[0] describes the read file descriptor.

2. An unnamed pipe is only used for communication between a child and it's parent process, while a named pipe can be used for communication between two unnamed process as well. Processes of different ancestry can share data through a named pipe.

3. A named pipe exists in the file system. After input-output has been performed by the sharing processes, the pipe still exists in the file system independently of the process, and can be used for communication between some other processes.

   On the other hand, an unnamed pipe vanishes as soon as it is closed, or one of the process (parent or child) completes execution.

4. Named pipes can be used to provide communication between processes on the same computer or between processes on different computers across a network, as in case of a distributed system. Unnamed pipes are always local; they cannot be used for communication over a network.

5. A Named pipe can have multiple processes communicating through it, like multiple clients connected to one server. On the other hand, an unnamed pipe is a one-way pipe that typically transfers data between a parent process and a child process.

Message Queues:

Why do we need message queues when we already have the shared memory?

It would be for multiple reasons, let us try to break this into multiple points for simplification −

- As understood, once the message is received by a process it would be no longer available for any other process. Whereas in shared memory, the data is available for multiple processes to access.

- If we want to communicate with small message formats.

- Shared memory data need to be protected with synchronization when multiple processes communicating at the same time.

- Frequency of writing and reading using the shared memory is high, then it would be very complex to implement the functionality. Not worth with regard to utilization in this kind of cases.

- What if all the processes do not need to access the shared memory but very few processes only need it, it would be better to implement with message queues.

- If we want to communicate with different data packets, say process A is sending message type 1 to process B, message type 10 to process C, and message type 20 to process D. In this case, it is simpler to implement with message queues. To simplify the given message type as 1, 10, 20, it can be either 0 or +ve or –ve as discussed below.

- Of course, the order of message queue is FIFO (First In First Out). The first message inserted in the queue is the first one to be retrieved.

- ❖ Using Shared Memory or Message Queues depends on the need of the application and how effectively it can be utilized.

Communication using message queues can happen in the following ways −

- Writing into the shared memory by one process and reading from the shared memory by another process. As we are aware, reading can be done with multiple processes as well.
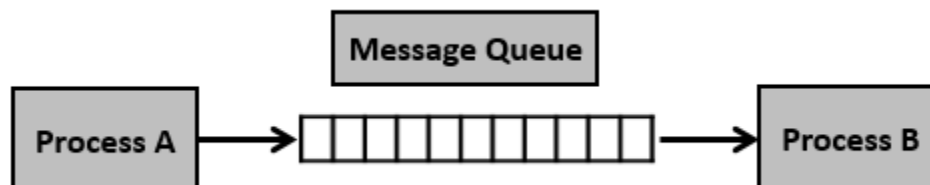
- Writing into the shared memory by one process with different data packets and reading from it by multiple processes, i.e., as per message type.

-

M.NAGARAJU ASSISTANT PROFESSOR (CSE)
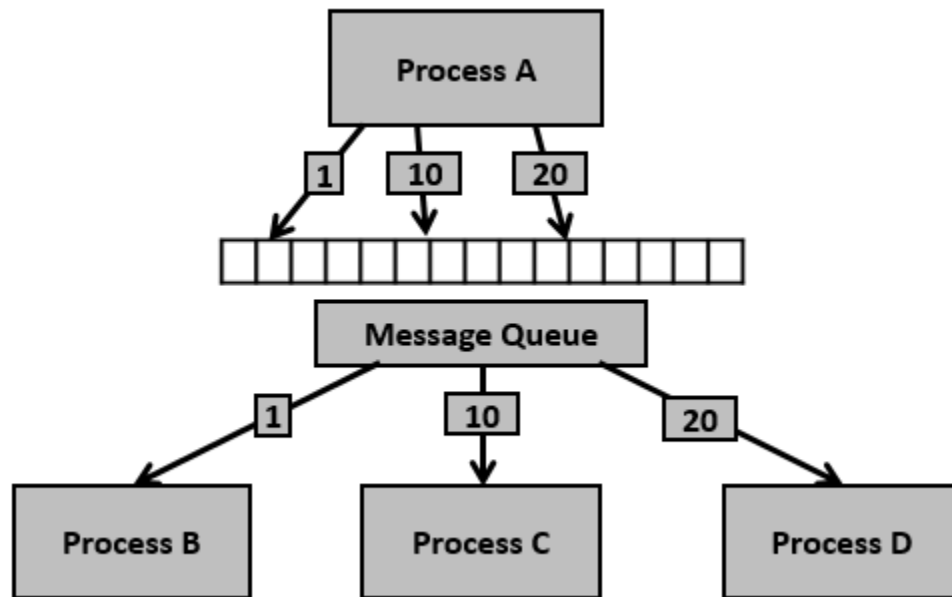
Having seen certain information on message queues, now it is time to check for the system call (System V) which supports the message queues.

> ❖ System V (System 5) was an early form of the Unix operating system, originally developed by AT&T (American Telephone and Telegraph). The first release, Release 1 (SVR1), appeared in 1983. Release 2 (SVR2) followed in 1984, Release 3 (SVR3) in 1987, and Release 4 (SVR4, the last and most popular version) in 1990.
>
> ❖ System V has been compared to BSD (which originally stood for Berkeley Software Distribution), another "flavor" of Unix. The first three versions of System V were preferred by businesses, while BSD was favored by university professors and research scientists.

❖ **To perform communication using message queues, following are the steps −**

    Step 1 − Create a message queue or connect to an already existing message queue (**msgget( )**)

    Step 2 − Write into message queue (**msgsnd( )**)

    Step 3 − Read from the message queue (**msgrcv( )**)

    Step 4 − Perform control operations on the message queue (**msgctl( )**)

Now, let us check the syntax and certain information on the above calls

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

API for message Queues:

Creation and accessing of a message queue:

❖ **mesgget( )** – Create a message Queue **or** get a System V message queue identifier.

Synopsis:

> **#include <sys/types.h>**
> **#include <sys/ipc.h>**
> **#include <sys/msg.h>**
>
> **int msgget(key_t** key**, int** msgflg**);**

Description:

This system call creates or allocates a System V message queue. Following arguments need to be passed –

- The first argument, key, recognizes the message queue. The key can be either an arbitrary value or one that can be derived from the library function ftok( ).

- The second argument, msgflg, specifies the required message queue flag/s such as IPC_CREAT (creating message queue if not exists) or IPC_EXCL (Used with IPC_CREAT to create the message queue and the call fails, if the message queue already exists). Need to pass the permissions as well.

Return Value:

If successful, the return value will be the message queue identifier (a nonnegative integer), otherwise -1 with errno indicating the error.

Sending a message queue:

❖ **msgsnd( ) -** sends/appends a message into the message queue

Synopsis:

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

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)
```

47

OPERATING SYSTEMS

Description:

This system call sends/appends a message into the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., message queue identifier. The identifier value is received upon the success of msgget()

- The second argument, msgp, is the pointer to the message, sent to the caller, defined in the structure of the following form −

```
struct msgbuf {

  long mtype;

  char mtext[1];

};
```

➢ The variable mtype is used for communicating with different message types, explained in detail in msgrcv( ) call.

➢ The variable mtext is an array or other structure whose size is specified by msgsz (positive value). If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.

- The third argument, msgsz, is the size of message (the message should end with a null character)

- The fourth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in queue or MSG_NOERROR (truncates message text, if more than msgsz bytes)

Return Value:

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror( ) function.

Retrieving a message:

❖ **msgrcv( )** - retrieves the message from the message queue.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)
```

48

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

Description:

This system call retrieves the message from the message queue (System V). Following arguments need to be passed −

- The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget( )

- The second argument, msgp, is the pointer of the message received from the caller. It is defined in the structure of the following form –

```
struct msgbuf {
  long mtype;
  char mtext[1];
};
```

- ➢ The variable mtype is used for communicating with different message types. The variable mtext is an array or other structure whose size is specified by msgsz (positive value).
- ➢ If the mtext field is not mentioned, then it is considered as zero size message, which is permitted.
  - The third argument, msgsz, is the size of the message received (message should end with a null character)

  - The fouth argument, msgtype, indicates the type of message −

    - o **If msgtype is 0** − Reads the first received message in the queue

    - o **If msgtype is +ve** − Reads the first message in the queue of type msgtype (if msgtype is 10, then reads only the first message of type 10 even though other types may be in the queue at the beginning)

    - o **If msgtype is –ve** − Reads the first message of lowest type less than or equal to the absolute value of message type (say, if msgtype is -5, then it reads first message of type less than 5 i.e., message type from 1 to 5)

  - The fifth argument, msgflg, indicates certain flags such as IPC_NOWAIT (returns immediately when no message is found in the queue or MSG_NOERROR (truncates the message text if more than msgsz bytes).

Return Value:

This call would return the number of bytes actually received in mtext array on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror( ) function.

49

Controlling the message queue:

❖ **msgctl ( )** - message control operations

Synopsis:

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int *msqid***, int** *cmd***, struct msqid_ds ****buf***);**

Description:

This system call performs control operations of the message queue (System V). Following arguments need to be passed −

- ➢ The first argument, msgid, recognizes the message queue i.e., the message queue identifier. The identifier value is received upon the success of msgget( )

- ➢ The second argument, cmd, is the command to perform the required control operation on the message queue. Valid values for cmd are −

    - ✓ **IPC_STAT** − Copies information of the current values of each member of struct msqid_ds to the passed structure pointed by buf. This command requires read permission on the message queue.

    - ✓ **IPC_SET** − Sets the user ID, group ID of the owner, permissions etc pointed to by structure buf.

    - ✓ **IPC_RMID** − Removes the message queue immediately.

    - ✓ **IPC_INFO** − Returns information about the message queue limits and parameters in the structure pointed by buf, which is of type struct msginfo

    - ✓ **MSG_INFO** − Returns an msginfo structure containing information about the consumed system resources by the message queue.

- ➢ The third argument, buf, is a pointer to the message queue structure named struct msqid_ds. The values of this structure would be used for either set or get as per cmd.

Return Value:

- • This call would return the value depending on the passed command.

50

- Success of IPC_INFO and MSG_INFO or MSG_STAT returns the index or identifier of the message queue or 0 for other operations and -1 in case of failure.

Clent/server Example:
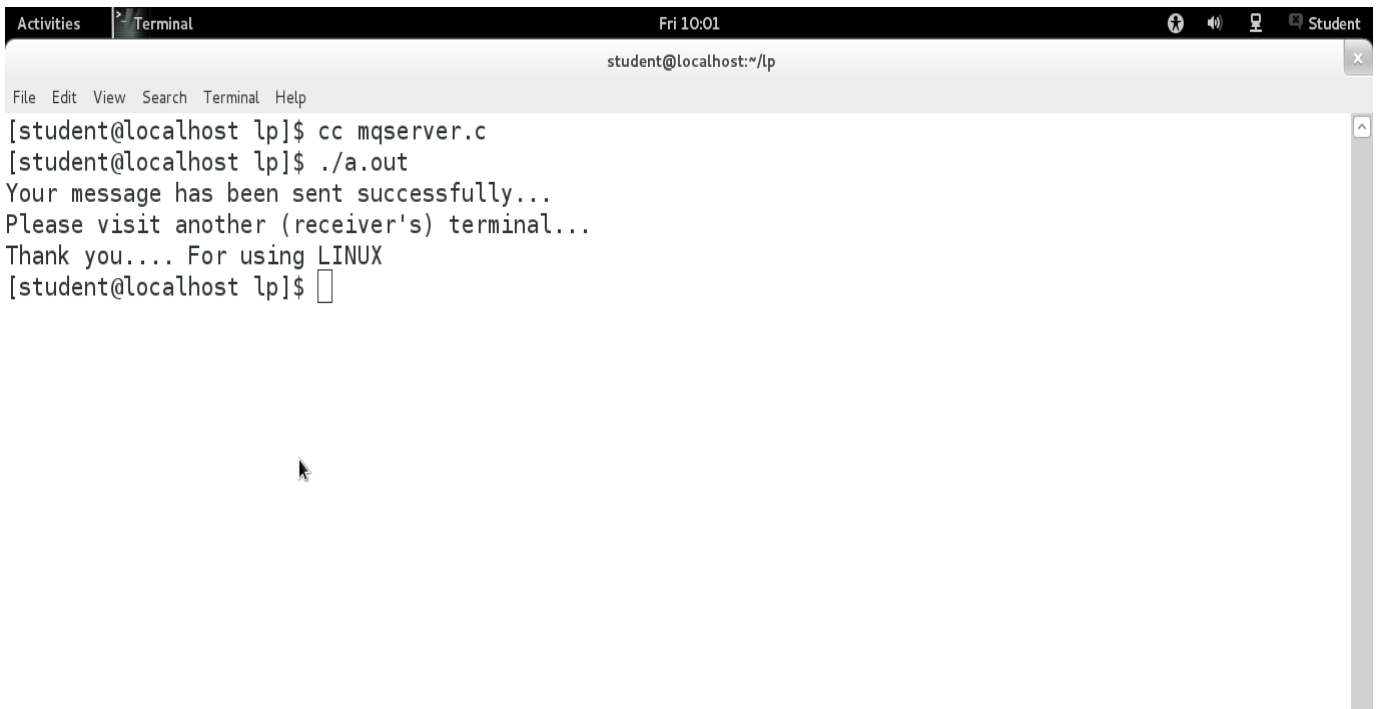
mqserver.c

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
#include<stdlib.h>
#define SIZE 2000
void main()
{
int mfd,mfd2,mfd3;
struct
{
double mtype;
char mtext[2000];
}s1,s2,s3;
if((mfd=msgget(1000,IPC_CREAT|0666))==-1)
{
perror("msgget:");
exit(1);
}
s1.mtype=1;
sprintf(s1.mtext,"%s","Hi friends... My name is message1");
if(msgsnd(mfd,&s1,1000,0)==-1)
{
perror("msgsnd");
exit(1);
}
if((mfd2=msgget(1000,IPC_CREAT|0666))==-1)
{
perror("msgget:");
exit(1);
}
s2.mtype=1;
sprintf(s2.mtext,"%s","Hi friends... My name is message2");
if(msgsnd(mfd2,&s2,1000,0)==-1)
{
perror("msgsnd");
exit(1);
}

if((mfd3=msgget(1000,IPC_CREAT|0666))==-1)
{
perror("msgget:");
```

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

```
exit(1);
}
s3.mtype=1;
sprintf(s3.mtext,"%s","Hi friends... My name is message3");
if(msgsnd(mfd3,&s3,1000,0)==-1)
{
perror("msgsnd");
exit(1);
}
printf("Your message has been sent successfully...\n");
printf("Please visit another (receiver's) terminal...\n");
printf("Thank you.... For using LINUX\n");
}
```
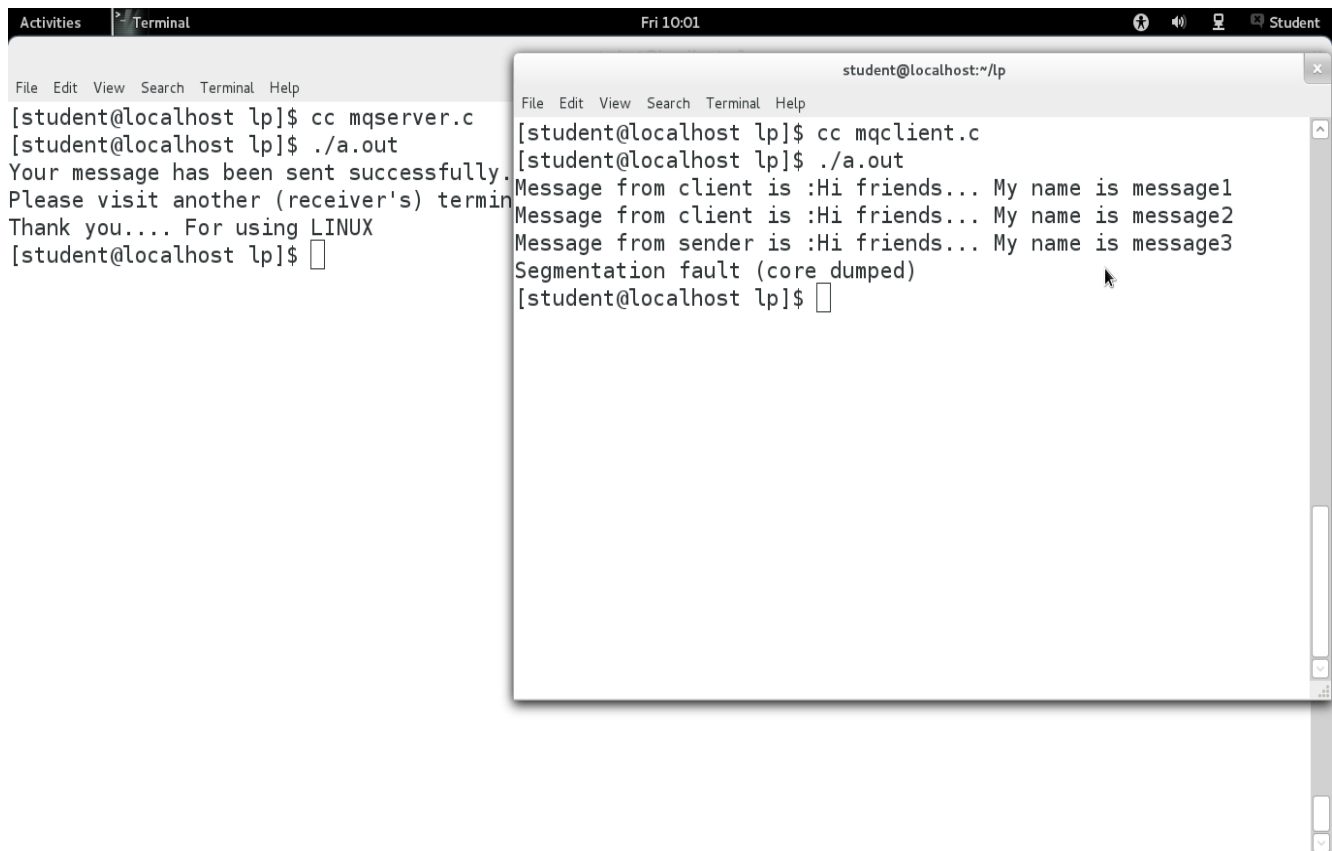
OUTPUT:

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

mqclient.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
#define SIZE 40
void main()
{
int mfd,mfd2,mfd3;
struct
{
long mtype;
char mtext[6];
}s1,s2,s3;
if((mfd=msgget(1000,0))==-1)
{
perror("msgget");
exit(1);
}
if(msgrcv(mfd,&s1,SIZE,0,IPC_NOWAIT|MSG_NOERROR)==-1)
{
perror("msgrcv");
exit(1);
}
printf("Message from client is :%s\n",s1.mtext);
if((mfd2=msgget(1000,0))==-1)
{
perror("msgget");
exit(1);
}
if(msgrcv(mfd2,&s2,SIZE,0,IPC_NOWAIT|MSG_NOERROR)==-1)
{
perror("msgrcv");
exit(1);
}
printf("Message from client is :%s\n",s2.mtext);
if((mfd3=msgget(1000,0))==-1)
{
perror("msgget");
exit(1);
}
if(msgrcv(mfd3,&s3,SIZE,0,IPC_NOWAIT|MSG_NOERROR)==-1)
{
perror("msgrcv");
exit(1);
}
printf("Message from sender is :%s\n",s3.mtext);
}
```

OUTPUT:

Shared Memory :

❖ <u>Inter Process Communication</u> through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by anther process.

❖ The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel, again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

❖ A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

We have seen the IPC techniques of Pipes and Named pipes and now it is time to know the remaining IPC techniques viz., Shared Memory, Message Queues, Semaphores, Signals, and Memory Mapping.
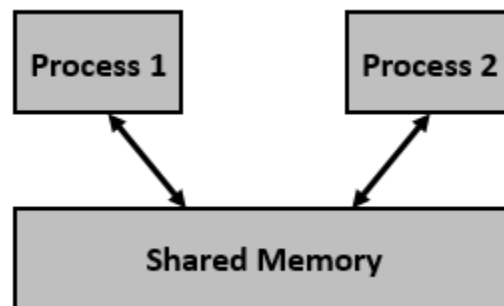
We know that to communicate between two or more processes, we use shared memory but before using the shared memory what needs to be done with the system calls, let us see this –

System Calls for Shared Memory:

- Create the shared memory segment or use an already created shared memory segment (shmget( ))
- Attach the process to the already created shared memory segment (shmat( ))
- Detach the process from the already attached shared memory segment (shmdt( ))
- Control operations on the shared memory segment (shmctl( ))

<u>Let us look at a few details of the system calls related to shared memory.</u>

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

1. **shmget ( )** - allocates a shared memory segment

Synopsis:

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg)

Description:

➢ The **first argument, key,** recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok( ). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communication. If the client wants to use shared memory with this key, then it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

➢ The **second argument, size,** is the size of the shared memory segment rounded to multiple of PAGE_SIZE.

➢ The **third argument, shmflg,** specifies the required shared memory flag/s such as IPC_CREAT (creating new segment) or IPC_EXCL (Used with IPC_CREAT to create new segment and the call fails, if the segment already exists). Need to pass the permissions as well.

Return Value: A valid segment identifier, *shmid*, is returned on success, -1 on error.

Errors:

Tag	Description
EACCES	The user does not have permission to access the shared memory segment, and does not have the**CAP_IPC_OWNER** capability.
EEXIST	**IPC_CREAT	IPC_EXCL** was specified and the segment exists.

2. **shmat ( )** - shared memory attach operation

Syntax:

#include <sys/types.h>

#include <sys/shm.h>

void * shmat(int shmid, const void *shmaddr, int shmflg)

56

Description:

➢ **The first argument, shmid,** is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget( ) system call.

➢ **The second argument, shmaddr,** is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

➢ **The third argument, shmflg,** specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

Return Value:

On success **shmat**( ) returns the address of the attached shared memory segment; on -1 is returned, and errno is set to indicate the cause of the error.

Errors:

When **shmat**( ) fails, *errno* is set to one of the following:

EACCES:

The calling process does not have the required permissions for the requested attach type, and does not have the **CAP_IPC_OWNER** capability.

EIDRM: *shmid* points to a removed identifier.

3. **shmdt ( )** - shared memory operations

Syntax:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr)
```

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

Description:

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

Return value:

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror( ) function.

4. **Shmctl ( ) -**shared memory control

Synopsis:

```
#include <sys/ipc.h>
 #include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Description:

- ❖ The first argument, shmid, is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget( ) system call.

- ❖ The second argument, cmd, is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are −

- **IPC_STAT** − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.

- **IPC_SET** − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.

- **IPC_RMID** − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.

- **IPC_INFO** − Returns the information about the shared memory limits and parameters in the structure pointed by buf.

- **SHM_INFO** − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

M.NAGARAJU ASSISTANT PROFESSOR (CSE)

❖ The third argument, buf, is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

Return value:

❖ This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other operations and -1 in case of failure

Shared memory example

Writer.c

```c
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/stat.h>

int main ( )

{

 int segment_id;

 char bogus;

 char* shared_memory;

 struct shmid_ds shmbuffer;

 int segment_size;

 const int shared_segment_size = 0x6400;


 /* Allocate a shared memory segment.  */

 segment_id = shmget (IPC_PRIVATE, shared_segment_size, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);

 /* Attach the shared memory segment.  */

 printf("Shared memory segment ID is %d\n", segment_id);

 shared_memory = (char*) shmat (segment_id, 0, 0);

 printf ("shared memory attached at address %p\n", shared_memory);

 /* Determine the segment's size. */
```

59

```
    /*

    shmctl (segment_id, IPC_STAT, &shmbuffer);

    segment_size  =           shmbuffer.shm_segsz;

    printf ("segment size: %d\n", segment_size);

    */

    /* Write a string to the shared memory segment.  */

    sprintf (shared_memory, "Hello, world.");

    /* Detach the shared memory segment.  */

    shmdt (shared_memory);

    printf("Wrote Hello World to the segment\n");


}
```

Reader.c

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/stat.h>

int main ()

{

 int segment_id;

 char bogus;

 char* shared_memory;

 struct shmid_ds shmbuffer;

 int segment_size;

 const int shared_segment_size = 0x6400;

 printf("Enter the shared memory id: ");

 scanf("%d", &segment_id);
```
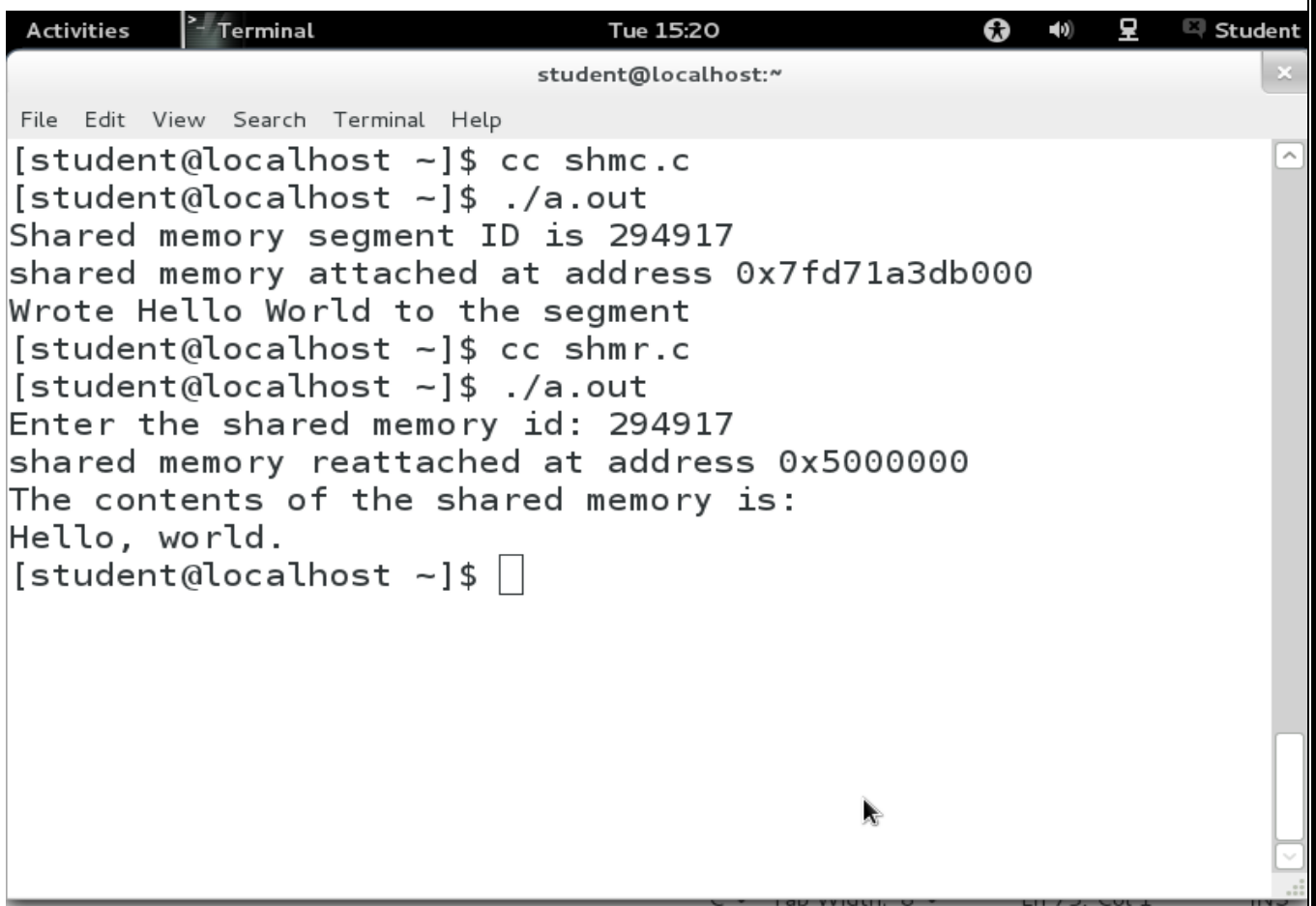
M.NAGARAJU ASSISTANT PROFESSOR (CSE)

```
/* Reattach the shared memory segment, at a different address.  */

shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);

printf ("shared memory reattached at address %p\n", shared_memory);

/* Print out the string from shared memory.  */

printf ("The contents of the shared memory is:\n%s\n", shared_memory);

/* Detach the shared memory segment.  */

shmdt (shared_memory);

return 0;

}
```

OUTPUT:

```
[student@localhost ~]$ cc shmc.c
[student@localhost ~]$ ./a.out
Shared memory segment ID is 294917
shared memory attached at address 0x7fd71a3db000
Wrote Hello World to the segment
[student@localhost ~]$ cc shmr.c
[student@localhost ~]$ ./a.out
Enter the shared memory id: 294917
shared memory reattached at address 0x5000000
The contents of the shared memory is:
Hello, world.
[student@localhost ~]$
```

M.NAGARAJU ASSISTANT PROFESSOR (CSE)