mood-book

UNIT-3: DEADLOCKS

→ In multiprogramming all processes share resources
→ A process requests resources, if the resources are not available the process moves into waiting state.
→ Waiting process never change their state. Because the resources may be held by other waiting process. This is called deadlock

Now we discuss the methods that an OS can use to prevent (or) deal deadlocks.
→ A process requests a resource before using it, and must release the resource after using it.
→ A process can request any no. of resources to perform a task i.e no. of resources requested may not exceed the total no. of resources available in the system.
Ex: A process cannot request 2 printers if only 1 printer is available
→ A process may utilise a resource iff only the following sequence :
1. Request: The process requests a resource, if not available (i.e, it is being used by another process) then the process has to wait until it can get the resources.
2. Use: The process can operate on the resource
3. Release: The process releases the resource
→ The request and release of resources is system calls.
Ex: request( ) & release( ) for devices
open( ) & close( ) for file
allocate( ) & free( ) for memory
wait( ) & signal( ) on Semaphores for other resources etc.,

Ex: consider a system with one printer and one tape drive. Suppose P_i is holding the tape driver and P_j is holding the printer. If P_i requests the printer and P_j requests the tape drive, a deadlock occurs.

Deadlock characterization:

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

Features that characterisation deadlocks (or) A deadlock situation arise if the four conditions hold simultaneously in a system:

1) Mutual Exclusion: Only one process at a time can use the resource. If another process requests that resource, the requesting process must wait until the resource is released.

2) Hold and wait: A process is holding a resource and waiting to acquire additional resource that are currently being held by other processes.

3) No premption: Resources cannot be prempted that is a resource can be released voluntarily by process holding it, after the process has completed its task

4) Circular Wait: A set $\{P_0, P_1, \ldots . P_n\}$ of waiting processes exist such that

P_0 is waiting for resource held by P_1

P_1 is waiting for resource held by P_2

P_{n-1} is waiting for resource held by P_n

P_n is waiting for resource held by P_0

Resource - Allocation Graph:

Deadlocks can be described precisely in terms of directed graph called system resource allocation graph.

The graph consists of set of vertices 'v' and set of edges E.

Set of vertices 'v' is partitioned into two different types of nodes.

$P = \{P_1, P_2, \ldots\ldots, P_n\}$ - Set of all active processes in system

$R = \{R_1, R_2, \ldots\ldots, R_m\}$ - Set of all resources types in system

→ directed edge from process pi to resource type Rj is denoted by $P_i \rightarrow R_j$

i.e, Pi requested resource Rj and is waiting for that resource also called "request edge".

→ directed edge from resource type Rj to process Pi is denoted by $R_j \rightarrow P_i$ i.e, resource type Rj has been allocated to Pi also called "Assignment edge"

Pictorically Pi is represented as $\boxed{P_i}$

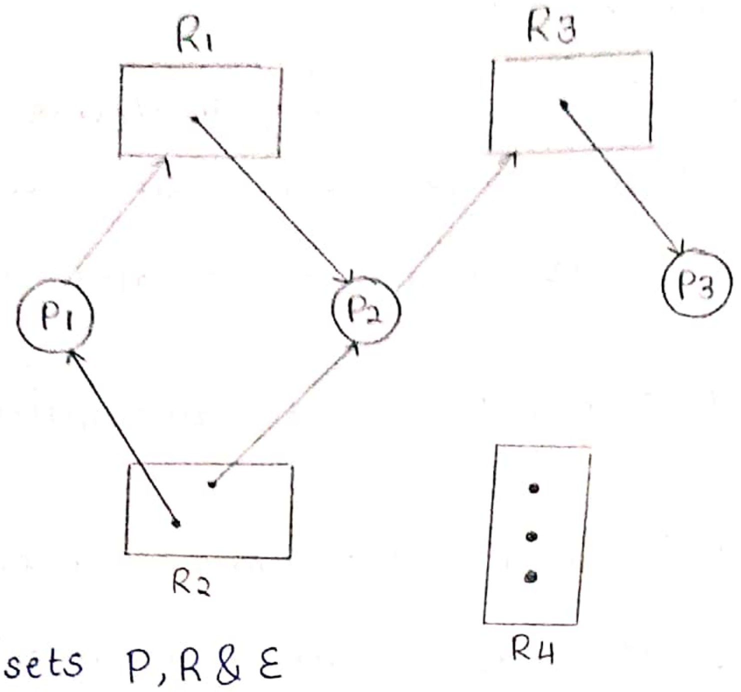
Rj is represented as $\boxed{R_j}$

If resource type has more than one instance. We represent such instance as a dot within square.

When the process no longer needs access to the resource it releases the resource, and as a result assignment edge is deleted.

Requested edge is instantaneously transformed into assignment edge when resource is allocated.

Ex:



The sets P, R & E

$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

→ 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 R2 and is waiting for R3.

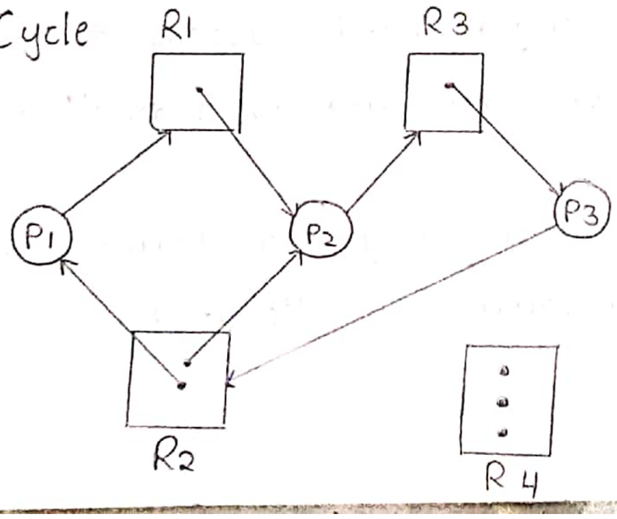
→ Process P3 is holding R3.

→ If cycles exist then deadlock occurs

(or)

If instances of resources are only one, then deadlock occurs

Ex: Cycle



We add a edge $P_3 \rightarrow P_2$

Here two cycles exist

① $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

② $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

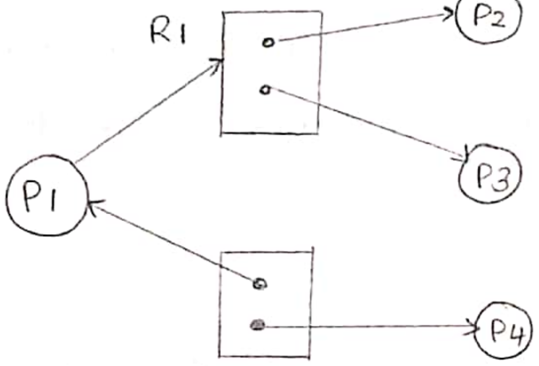
∴ P_1, P_2, P_3 are deadlocked

P_2 waiting for R_3 held by P_3

P_3 waiting for P_1

P_1 waiting for P_2

Cycle with no deadlock



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

No cycle – no deadlock

if cycle – may (or) may not be in deadlock state.

Methods for handling deadlocks:

3 Ways for handling deadlocks.

1) We can use a protocol to prevent (or) avoid deadlocks.

2) We can allow the system to enter deadlock state, detect it, recover it.

3) We can ignore the protocol

To see that deadlocks never occur, the system can use either

Deadlock prevention scheme :

Set of methods for ensuring that atleast one of the necessary conditions cannot hold.

Deadlock avoidance scheme : Requires OS be given an additional info that which resources a process will request and use using its life time.

The OS must consider

→ The resources currently available

→ A resources currently allocated to each process.

→ The future requests and releases of each process

→ if deadlocks are not detected it leads to deteriosation of system performance.

Deadlock prevention:

For a deadlock to occur each of the four conditions must hold. By ensuring that atleast one of the condition cannot hold, we can prevent deadlock

1) Mutual Exclusion :

Mutual exclusion must hold for non-sharable resources only.

Ex: Printer which cannot be simultaneously shared by several processes

Sharable resources – do not require mutually exclusive access.

∴ deadlock will not occur

Ex: Read-only files.

If several processes make attempt to open a read-only-file at same time, they can be granted Simultaneously.

∴ A process never needs to wait for sharable resources.

2) Hold and Wait:

To ensure, Hold and wait never occurs, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

→ One protocol is each process to request and be allocated all its resources before it begins execution.

→ Second protocol is a process request some resources and use them. Before it can request ~~resources~~ any additional resource, it must release all the resources that it is currently allocated.

Ex: [diff b/w two protocols]

A process copies data from a tape drive to a disk file, and then prints the results to a printer

- According to first protocol, all resources must be requested at the beginning of the process i.e tape drive, disk file and printer it will hold printer for its entire execution, even though it needs the printer at the end

Disadvantage: Resource utilisation

→ many resources are allocated but not used.

- According to second protocol, the process initially requests tape drive and disk file. It copies from tape drive to the disk, then releases both. The process must then again request the disk file and printer After copying, it releases these two resources and terminate.

[Starvation - A process that needs several resources may have to wait indefinitely, because atleast one of the resources that it needs is always allocated to some other process]

3) No premption:

To ensure this condition does not hold, if a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are prempted. The prempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that is requesting.
The resources are prempted only if another process requests them.

4) Circular Wait:

To ensure that this condition never holds is to impose a total ordering of all resource types, and each process must request resources in an increasing order of enumeration.

Let $R = \{R_1, R_2, \ldots, R_m\}$ be the set of resource type and each resource type is assigned a unique integer number, which allows us to compare two resources and to determine whether one preceeds another in our ordering

Let us define a function

$$\boxed{F : R \longrightarrow N}$$

$N \rightarrow$ set of natural number

Ex: F(tape drive):1

F(disk drive) = 5,

F(printer) = 12

→ Each process requests the resource in the increasing order of enumeration

→ If there are two protocols are used, circular wait cannot hold

→ initially a process can request R_i

After that a process can request R_j

if $F(R_j) > F(R_i)$

i.e, first request tape drive then printer

→ Whenever a process requests an instance of resource type R_j, it has released any resources R_i

Deadlock Avoidance:

In deadlock prevention - We see that any one of the necessary conditions for deadlock cannot occur. The disadvantages are low device utilization and reduced system throughput.

So, instead of preventing deadlock, we ensure that the system will never enter into deadlock state. also defines - deadlock - avoidance approach. So, additional info about how resources are to be requested.

Safe state:

A system is in safe state if the system can allocate resources to each process and avoid deadlock.

i.e., A system is in a safe state only if there exists a "safe sequence".

A sequence of processes $< P_1, P_2, \ldots, P_n >$ is a safe sequence, if

Ex: If the resources that process P_i needs are not immediately available, then P_i can wait until all P_j has finished. When they have finished, P_i can obtain all of its needed resources, complete its task, return the resources and terminate when P_i terminates P_{i+1} can obtain its needed resources and so on.

If no such sequence exists the system system is unsafe

Ex: Let there be 3 processes P_1, P_2, P_3 and system has 12 magnetic tape drives.

Process P_0 requires 10 tape drives.

P_1 requires 4 tape drives.

P_2 requires 9 tape drives.

Suppose at time t_0 (initially)

P_0 is holding 5 tape drives

P_1 is holding 2 tape drives

P_2 is holding 2 tape drives

	max needs	current needs
P_0	10	5
P_1	4	2
P_2	9	2

Thus 3 tape drives are free.

At time "t_0" the system is in safe state.

Now, in which sequence the process must be allocated resources such that the system remains in safe state.

The sequence is $< P_1, P_0, P_2 >$

P_1 can be immediately allocated 2 tape drives and it completes its task and returns 4 tapedrives + 1 remain

$\therefore$ 5 tape drives with system. These 5 tape drives are allocated to P_0. So, P_0 completes its task and returns 10 to system. P_2 uses 8 tape drives and returns 10.

$\therefore$ At last system has 12 tape drives and system is in safe state.

Suppose,

At t_1 - P_2 is allocated one more tape drive & P_1 is allocated all tape drives (i.e $2t_2$) after P_1 completes its releases 4 tape drives. But it cannot satisfy P_0 (or) P_2.

∴ System is in unsafe state. So, if P_2 is made to wait until either of the process had finished and released resources, deadlock can be avoided.

Resource - Allocation Graph algorithm:

In this the process must first intimate the OS. What are the resources it may claim in future.

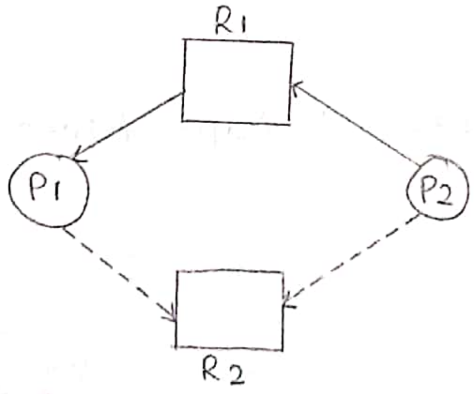
So, in addition to request and assignment edges, we introduce a new type of edge called "claim edge".

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at sometime in future. It is same as request edge in direction by represented with dashed line.

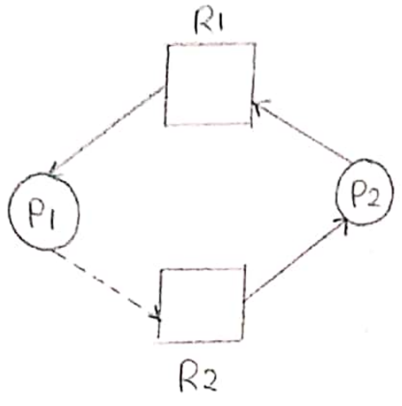
When P_i requests R_j the claim edge is converted to request edge.

i.e. Before P_i starts executing, all its claim edger must already appear in resource-allocation Graph.

Ex:



Suppose R_2 is allocated to P_2



then deadlock occurs

$R_2 \rightarrow P_2 \rightarrow R_1 \rightarrow P_1 \rightarrow R_2$

Banker's Algorithm

Safety Algorithm:

1. Let Work and finish be vertices of length m and n respectively. Initialise

Work := Available and finish[i] := false
for i = 1, 2, ..., n

2. Find an i such that both
   a. Finish[i] = false
   b. Need i ≤ Available[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 safe state.

Resource - Request Algorithm:

Let Request i be the request vector for process Pi

Request i[j] = K → process Pi wants K instances of resource type Rj.

When request for resources is made by Pi then the actions are:

1. If $Request_i \leq Need_i$, go to step-2, Otherwise, rise an error condition, since process has exceeded its max claim

2. If $request_i \leq Available$, go to step 3

Otherwise, P_i must wait, since resources are not available

3. Available: = Available - $Request_i$;

Allocation : = $Allocation_i$ + $Request_i$;

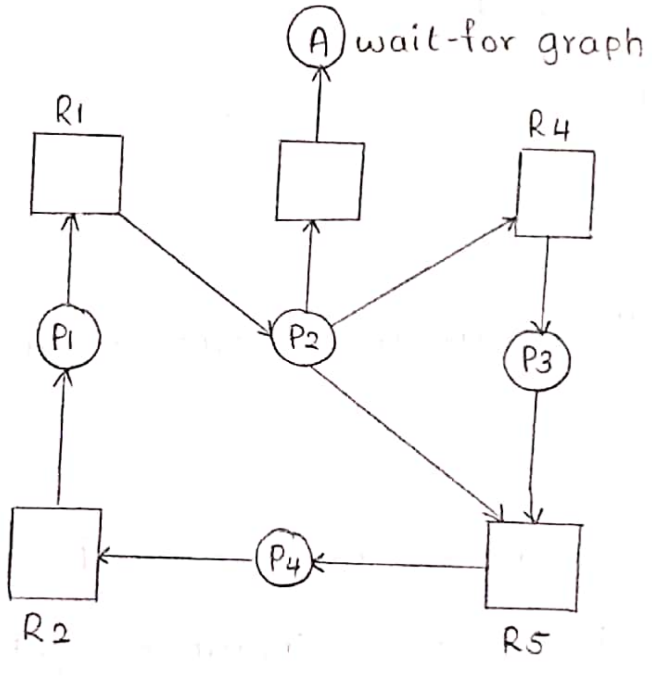
$Need_i$: = $Need_i$ - $Request_i$;

If the resulting resource - Allocation state is safe, the transaction is completed a process P_i is allocated its resources.

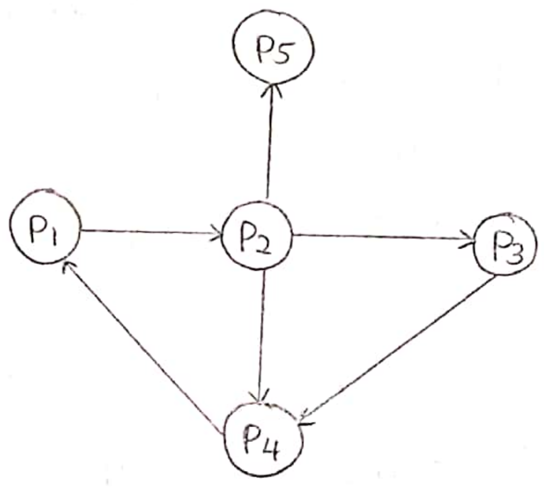
If the state is unsafe, P_i must wait for $Request_i$

Deadlock detection:

Single instance of each resource type:



Resource - all graph



Wait-for

Several Instances of a Resource type:

Available – A vector of length m indicates the no. of available resources of each type

Allocation – An n x m matrix defines the no. of resources of each type

Request – An n x m matrix indicates the current request of each process

if request[i, j] = K, then P_i is requesting K more instances of resource type R_j

Algorithm:

1. Let work & finish be vectors of length m & n respectively. Initialise

   Work: = Available, for i = 1, 2, ..., n, if

   $Allocation_i \neq 0$, then finish[i]: = False;

   otherwise, finish[i]: = true.

2. Find ~~c~~an i such that

   a. Finish[i] = False

   b. $Request_i \leq$ Work[Available]

   if no such i, exists go to step 4.

3. Work : = Work + $Allocation_i$;

   Finish [i]: = true

   go to step 2

4. If finish[i] = false, then the system in a deadlock state

Recovery from Deadlock:

After deadlocks are detected, recover the system from deadlock

Two methods to recover from deadlocks

1) **Process Termination:**

→ Abort all deadlocked processes

→ Abort one process at a time until the deadlock cycle is elimination

2) **Resource Premption:**

→ To eliminate deadlocks using premption We prempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

3 issues to deal

1) Selecting a victim – incur min.cost

2) Rollback – prempt resources and rollback it to source safe state & restart it from that state.

But total rollback is done, since it is difficult to identify the safe state.

3) Starvation – Suppose some process is always picked as a victim then that process starves. So, choose a victim only some finite no. of times

→ process termination may incur more cost. So select a process which will incur min.cost

Eg:- To illustrate the use of banker's algorithm, Consider a system with 5 processes P_0 through P_4 & three resource types A, B & C. Resource type A has ten instances, B has 5 instances & C has 7 instances. Suppose that at time T_0, the following Snapshot of the system has been taken:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Sol:- for the Given Allocation matrix & Max Matrix find the need matrix as

Need = Max − Allocation

$$= \begin{bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6, & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

Step 1

Work = Available

= 3 3 2

finish[i] = false i = 0, . . 4

f	f	f	f	f
0	1	2	3	4

Step2

$Need_i \le Awork$

i = 0

$Need_0 \le Work$

$743 \le 332$ ✗

P_0 cannot Executes

Next, i = 1

$Need_1 \le Work$

$122 \le 332$ ✓

P_1 can Execute

finish[1] = True i.e,

F	T	F	f	F
0	1	2	3	4

$Work = Work + Allocation_1$

= 532 + 200

= 532

i = 2

$Need_2 \le Work$

$600 \le 532$ ✗

P_2 cannot Execute

i = 3

$Need_3 \le Work$

0	1	2	3	4
F	T	T	T	f

$011 \le 532$ ✓

P_3 can Execute & finish[3] = True

$Work = Work + Allocation_3 = 532 + 211$

= 743

$i = 4$

$Need_4 \leq Work$

$431 \leq 743$ ✓

P_4 can Execute & finish[4] = True

$Work = Work + Allocation_4$

F	T	F	T	T
0	1	2	3	4

$= 743 + 002$

$= 745$

Again start from $i = 0$, $Need_0 \leq Work$

$743 \leq 745$ ✓

P_0 can Execute & finish[0] = True

$Work = Work + Allocation_0$

T	T	F	T	T
0	1	2	3	4

$= 745 + 010$

$= 755$

$i = 2$, $Need_2 \leq Work$

$600 \leq 755$ ✓

P_2 can Execute & finish[2] = True

$Work = Work + Allocation_2$

T	T	T	T	T
0	1	2	3	4

$= 755 + 302$

$= 10\ 5\ 7$

Hence all processes has finished Execution system is in safe state & the safe sequence is

$< P_1, P_3, P_4, P_0, P_2 >$

Suppose, now that process P_1 request one additional instance of resource type A & 2 instances of C. Whether request can be granted if yes find safe sequence.

Sol:- Given that P_1 requests 1, 0, 2 additionaly. So, Run Resource-Request algorithm to find whether request can be granted or not.

First check $Request_i \leq Available$

i.e, $1\ 0\ 2 \leq 3\ 3\ 2$ ✓

The following condition is satisfied. So the request can be granted. (Update all the values corresponding to P_1)

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	2	3	0
P_1	3	0	2	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Need = Max − Allocation

$$= \begin{bmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

Now, run Safety algorithm, to find Safe sequence

<u>Step1</u> Work = Available
= 2 3 0

finish[i] = false , i = 0 to 4

F	F	F	F	F
0	1	2	3	4

<u>Step2</u> $Need_i \le Work$

i = 0, 743 ≤ 230 ✗

i = 1 020 ≤ 230 ✓

P_1 Executes & finish[1] = true

F	T	F	F	F
0	1	2	3	4

Work = Work + $Allocation_1$
= 230 + 302
= 5 3 2

i = 2, 600 ≤ 532 ✗

i = 3 011 ≤ 532 ✓

P_3 Executes & finish[3] = true

F	T	F	T	F
0	1	2	3	4

Work = Work + $Allocation_3$
= 532 + 211 = 743

$i=4$, $431 \le 743$ ✓

P_4 Executes & finish[4] = true

F	T	F	T	T
0	1	2	3	4

Work = Work + Allocation$_4$

$= 743 + 002 = 745$

Again start from $i=0$,

$743 \le 745$ ✓

P_0 Executes & finish[0] = true

T	T	F	T	T
0	1	2	3	4

Work = Work + Allocation$_0$

$= 745 + 010$

$= 755$

$i=2$, $600 \le 755$ ✓

P_2 Executes & finish[2] = true

T	T	T	T	T
0	1	2	3	4

Work = Work + Allocation$_2$

$= 755 + 302$

$= 10\ 5\ 7$

∴ System is in safe state & safe sequence is

$< P_1, P_3, P_4, P_0, P_2 >$

3.2 SYNCHRONISATION

Synchronization: In cooperative process there may be a situation for the occurance of data inconsistency when two processes executed parallely and perform modification operations on shareable resource.

Ex: In production Consumer problem both producer and Consumer executed simultaneously, they need to modify the value of global variable "counter" shared by both producer and Consumer procedures. In this the Counter Value is either 4 or 6 but not 5.

To avoid data inconsistency use the concept of Synchronization Co-operative processes. Synchronization is a mechanism to ensure that orderly execution of Cooperative processes. While perform modifications on shareable resources. By using synchronization it provides data consistency.

Consumer

```
while (true)
{
 while (counter == 0)
   ;
 next Consumed = buf [out];
 out = (out + 1) % BUFSIZE;
   Counter --;
 }
```

Producer

```
while (true)
{
while (counter == BUFSIZE)
   ;
buf [in] = next produced;
in = (in + 1) % BUFSIZE
Counter ++;
}
```

```
reg 1 = Countee;     }            {  reg 2 = Countee;
reg 1 = reg1-1;      }  Internal  {  reg 2 = reg2 +1;
Countee = reg 1;     }  Operation {  Countee = reg2;
```

Critical Section Problem: A system contains 'n' no. of processes $P_1, P_2, \ldots P_n$ share a common file or try to modify a common variable. Then the system need to allow only one process to modify file content and Variable Content does not allow any other process until the completion of first process Execution. In this the code in a process used to implement Modification Operation on a file or Variable is called Critical Section. So the system design a protocol used to prevent anyother process execute its critical section until current process completes the critical Section execution. The protocol provides 2 features.

1. The process make request for the permission to enter into its critical section. The code to implement this request is called Entry Section.

2. The process need to release the resources when exit from the critical Section. The code to implement the exit Operation is called exit section. The remaining code in process is called Remainder Section.

→ The structure of a process contains a critical section is shown below:

```
do
{
    Entry Section
    Critical Section
    Exit Section
  Remainder Section;
} while
```

→ A solution designed for critical section problem must satisfies given 3 requirements.

1. Mutual Exclusion: It allows only one process into execution

2. Progress: If anyone process not enter into critical section, then select one process from the list of waiting processes allows it into critical section.

3. Bounded wait: There is a limited no. of times a process enter into its critical section when another process make a request for enter into critical section. A process completes its execution with in a limited waiting time for the arrival of the request.

→ The critical section problem also occurs in OS. programs The kernel can avoid critical section problem by executes the kernel in non-primitive mode, while executes the critical section i.e; the kernel can't be pre emitted its control until process is completed.

Peterson's Solution For Critical Section Problem:

Peterson provides a software Solution to solve the critical section problem implement Synchronization but the peterson's problem is used to Synchronize between any 2 processes for example: pi, pj are 2 processes. In peterson problem we can use 2 datastructures.

int turn;

→ used to specify which process turn is currently enters into the Critical Section.

boolean flag[2];

→ flag is used to specify the readyness (or) the interest of a process to enter into its critical Section.

flag[0];

→ specifies interest of pi:

flag[1];

→ Specifies interest of pj:

pi:

```
do
{
    turn = i;
    flag[i] = true;
    while ((flag[j] == True) && (Turn == j))
    ;
    Critical Section
    flag[j] = false;
} while (1);
```

Pi:

```
do
{
    turn = j;
    flag[i] = true;
    While ((flag[i] == True) && (Turn == i))
    ;
    Critical Section
    flag[i] = false;
} while (1);
```

Peterson solution satisfies

1. Mutual Exclusion
2. Progress
3. Bounded waiting

Drawbacks:

1. The solution is applicable only for 2 processes.
2. Busy waiting

Hardware Approach For Synchronization

Hardware approach makes easier to develop program and improves the system efficiency. There are two types of Hardware instructions are provided available with all Hardware configurations.

1. Test and Set()
2. Swap()

1. Test and Set: It is a atomic instruction. It is executed without any interrupts. It is used to test value of its argument and set the argument with a new value as

shown below:

```
Test and set ( boolean * target)
{
 boolean rv = * target ; (previous value of target
                                  assigned to rv)
     * target = True;
       return rv;
```

```
Pi:
do
{
while(Test and set (&Lock))
    ; do nothing
  // critical Section;
    lock = false;
 } while (true)
```

```
Pj:
 do
 {
 while (Test and Set(& Lock))
      ; do nothing
   // critical Section;
      lock = false;
   } while (true)
```

Mutual Exclusion implementation with test and set()

-All processes share a global variable lock and is initializ -ed with "False".

When one process. Test and Set address of is passed, it returns "FALSE" and set lock as "True" then it enters into the Critical section and prevent all the remaining processes by set the lock. to a "False" value i.e., removes the lock.-Any another process try to enters into critical section will enters into critical section and set the lock by set lock with "True".

```
  do
 {
waiting [i] = true;
 key = true;
while (waiting [i] && key)
  key = test and set (& lock);
waiting [i] = false;
// critical Section
j = (i+1) % n;
while ((j! = i) && ! waiting [j])
    j = (j+1) % n;
    if (j = = i)
        lock = false;
    else
        waiting [j] = false;
        //remainder section
    } while (true);
```

Bounded_waiting mutual Exclusion with testandset()

The above code satisfies mutual Exclusion, progress & bounded waiting.

Swap:

It is also a hardware instruction used to implement mutual exclusion between two (or) more processes. It is also a atomic instruction i.e; it is executed without any interrupts. It is used to interchanges the Contents of given two parameters as shown below:

```
Swap (boolean * lock, boolean * key)
{
    boolean * temp = * lock;
            * lock = * key;
            * key  = * temp;
}
boolean lock = false;
```

definition of Swap() instruction.

```
Pi
do
{
  boolean key = true;
  While (key == True)
  Swap (& lock, & key);
  // critical Section;
      lock = false;
  // remainder Section;
  } while (True).
```

```
Pj
do
{
  boolean key = True;
  while (key == True)
  swap (& lock, & key);
  // critical Section;
      lock = false;
  // remainder Section;
  } while (True).
```

mutual Exclusion implementation of swap() instruction.

In this we can use two variables. The global variable lock is shared by all the processes. It is initialized to "false" key is a local variable to each process. It is initialized to "True". A process check key value and swaps the context of lock and key variables. If key is always true the process doesn't enter into critical section, if key value is false then process enter into critical section.

Both test and set, swap instructions are used to only implement mutual exclusion. But they can't implemented bounded waiting. So, there are not implement Synchronization.

Software Approach:

Semaphore: It is a software to implement Synchronization. It reduces the complexity in implementation of synchronization. It is a integer variable. It is represented by 's'. It is used to Synchronizes the accessibility of a single resource. The

Semaphore value can be modified by executing two types of operations on a Semaphore.

i) wait: It is represented by 'p' (proberen) probeeen means to wait. It is used to test the Semaphore value. If value is less than (or) equal to '0' then put the process in waiting state otherwise decrements the semaphore value.

ii) Signal: It is represented by v (verhogen). verhogen means to increment. It is used to increment the Semaphore value by 1.

Both wait and Signal operations are atomic instructions so, no two processes are allowed at a time to perform either wait (or) signal operations on a Semaphore value.

<u>wait</u>

```
wait (int s)
{
 while (s<=0)
  ; do nothing
  s--;
}
```

<u>Signal</u>

```
Signal (int s)
{
  s++;
}
```

Now we take a Semaphore called "mutex" and initialized to 1.

<u>Implementation of Semaphore:</u>

<u>Pi</u>

```
do
{
wait (mutex);
// critical Section;
  Signal (mutex);
} while (True).
```

<u>Pj</u>

```
do
{
 wait (mutex);
 // critical Section;
   Signal (mutex);
} while (true)
```

In this when the 1st process executes wait operation with mutex value as 1 then the wait instruction decrements the Semaphore value by

i.e mutex is 0, then allows the process into Critical Section.

Now all the remaining processes that executes wait operation with mutex value '0' are put in busy waiting i.e., Continuously test the Semaphore value until greater than '0'.

Once process pi completes critical Section and execute signal Operation on mutex then the mutex value is incremented by 1. Now any one of the process is waiti-ng state are allowed to enter into critical Section.

In Semaphore the database is busy waiting.

There are 2 types of Semaphores available

1. Binary Semaphore: It is used to synchronize the accessing of single instance of a resource. Its value is either 0 (or) 1. If Semaphore value is 1 it means resource is available. If Semaphore value is 0 it means resource is not available.

2. Count Semaphore: It is used to Synchronize the accessibility of multiple instances of one resource. Its value is initialized to total no. of instances of a resource. In this we can able to execute 'n' no. of processes enters into critical section at a time. once S value becomes zero(0) all the remaining processes are put in waiting list & Semaphore value is decremented by 1.

It is used to stores the previous value target and load a new value into the target. It returns old value (rv) to the calling function. It is used to implement Synchronization among multiple processes as shown below:

```
boolean lock = false;
```

Deadlock :

In implementation of Synchronization using Semaphore there is a situation for the occuarance of deadlock. Deadlock is a state when a waiting process wait for an event performed by another process in the waiting list.

```
        Pi                              Pj

    wait (p);                       wait(q);
    wait (q);                       wait(p);
       |                               |
       |                               |
       |                               |
       |                               |
    signal (p);                     Signal(q);
    Signal(q);                      Signal (p);
```

Producer

```
do
{
// produce an item;
  wait(empty);
  wait(mutex);
// add item to buffer inpool;
  Signal(mutex);
  Signal(full);
}while(true)
```

Structure of producer process

Consumer

```
do
{
  wait(full);
  wait(mutex);
// Remove item from
   buffer pool;
  Signal(mutex);
  signal(empty);
// consume the item;
} while(true)
```

Structure of Consumer process.

Producer: First the producer checks empty space is available or not by executing wait (empty). If empty Space is available it again checks consumer is already uses the Buffer pool or not by executing wait (mutex). If the consumer is not using the Buffer pool. Now producer releases the lock applied on Buffer pool by executing Signal (mutex) & remove one item from the waiting consum-ers list & intimates to read item from the Buffer pool.

Consumer:

Consumer first verify is there any item is available in the buffer pool by executing wait (full). Once item is available in the buffer pool then it again checks if buffer pool is used by the producer or not by executing wait (mutex). If the buffer pool is free consumer remove one item from the Buffer pool.

After these unclocks the buffer pool using signal (mutex) & intimates to the producer for add or item to the buffer pool using signal (empty). Now the Consumer consumes the item.

In this way synchronization is implemented between producer & consumer using Semaphores.

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 & write) the database. We distinguish betwee-n these two types of processes by referring to the former as 'readers' and to the latter as 'writers'. 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 databa-se simultaneously, chaos 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 synch-ronization problem is referred to as the "readers- writers problem." Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers- writers problem has several variations, all involvi-ng priorities. 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 others readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve, in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Next, we present a solution to the first readers-writers problem. See the ~~bibliog~~ bibliographical notes at the end of the chapter for references describing starvation-free solutions to the second readers-writers problem.

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

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read.count = 0;
```

The semaphores mutex and rw_mutex are initialized to 1; read_count is initialized to 0. The semaphore rw_mutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read.count is updated.

```
do{
    wait (rw-mutex);
    /* writing is performed */
    signal (rw-mutex);
} while (true);
```

The structure of a writer process.

The read.count vaiicable keeps track of how many processes are currently reading the object. The Semaphore rw-mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that entees or exists the critical section. It is not used by readers who entee or exit while other readers are in their critical sections.

The code for a writer process is shown above, the code for a reader process to shown below. Note that if a writer is in the critical Section and n readers are waiting, then one reader is queued on rw_mutex, and n-1 readers are queued on mutex. Also observe that, when a writer executes signal (rw-mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers-writers problem and its solution have been generalized to provide reader-writer locks on some systems. Acquiring a reader-writer lock requires specifying the mode of the lock: either 'read' and (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. Multiple processes are permitted to concurrently acquire a reader-writer lock in reader mode, but only one process may acquire the lock for writing, as exclusive access is required for wish. writers.

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

```
do{
    wait(mutex);
    read.count ++;
    if (read.count == 1)
      wait (rw_mutex);
      signal(mutex);
  /* reading is peformed */
    wait (mutex);
    read.count--;
    if (read.count ==0)
        signal(rw-mutex);
      signal(mutex);
   }while (true);
```

The structure of a reader process.

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 ~~re~~ require more overhead to establish than semaphores or mutual exclusion locks. The increased concurrency of allowing multiple reader compensates for the overhead involved in setting up the reader-writer lock.

The Dining-philosophers Problem

Consider five philosphers who spend their lives thinking and eating. The philosphers share a circular table surrounded by five chairs, each belonging to one philosp-her. In the centre of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks she does not interact with collegues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbours). A philospher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbour. When a hungry philospher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

The 'dining-philosophers problem' is considered a classic synchronization problem neither because of its practical importance nor because computer scientists dislike philosophers but because it is an example of a large class of ency control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

One Simple Solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

```
Semaphore chopstick [5];
    do {
        wait (chopstick [i]);
        wait (chopstick [(i+1) % 5]);
      /* eat for a while */
       signal (chopstick [i]);
      signal (chopstick [(i+1) % 5]);
      /* think for a while */
    } while (true);
```

The structure of philosopher i.

where all the elements of chopsticks are initialized to 1. The structure of philospher i is shown above.

Although this solution guarantees that no two neighbo-rs are eating simultaneously, it nevertheless must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

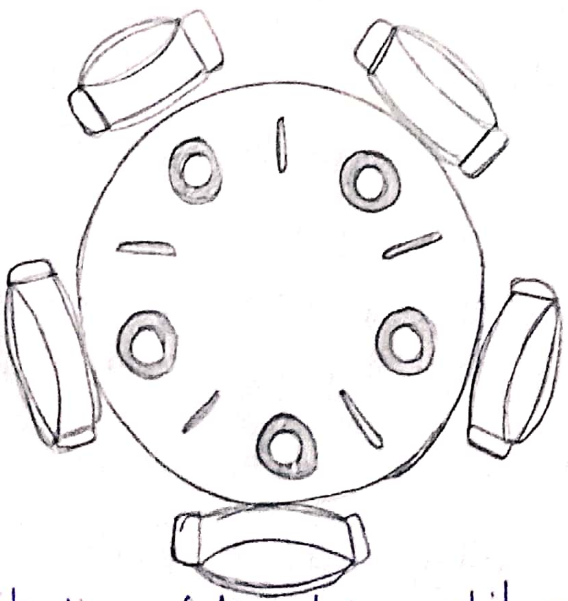
Several possible remedies to the deadlock problem are replaced by:

-Allow at most four philosophers to be sitting simultaneously at the table.

-Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).

Use an asymmetric solution - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

We present a solution to the dining-philosophers problem that ensures freedom from deadlocks. Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation.


The situation of the dining philosophers.

Monitors

A abstract data type or ADT encapsulates private data with public methods to operate on that data. A monitor type is an ADT which presents a set of programmer-defined opera-tions that are provided mutual exclusion within the moni-tor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functi-ons that operate on those variables. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local parameters.

```
monitor monitor_name
{ // shared variable declarations
  procedure p1(....) {
     ....
  }
  procedure p2(....) {
     ....
  }
       .
       .
       .
  procedure pn(....) {
     .....
  }
  initialization code(....) {
     .....
  }
}
```

Syntax of a monitor

The monitor construct ensures that only one process at a time is active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly. However, the monitor construct, as defined so far, is not sufficiently powerful for modelling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal().

x.wait → means that the process invoking this operation is suspended until another process involves.

x.signal → means that resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect.

Note, however, that P and Q processes can conceptually continue with their execution. Two possibilities exist.

1. Signal and wait: P either waits until Q leaves the monitor or waits for another condition.

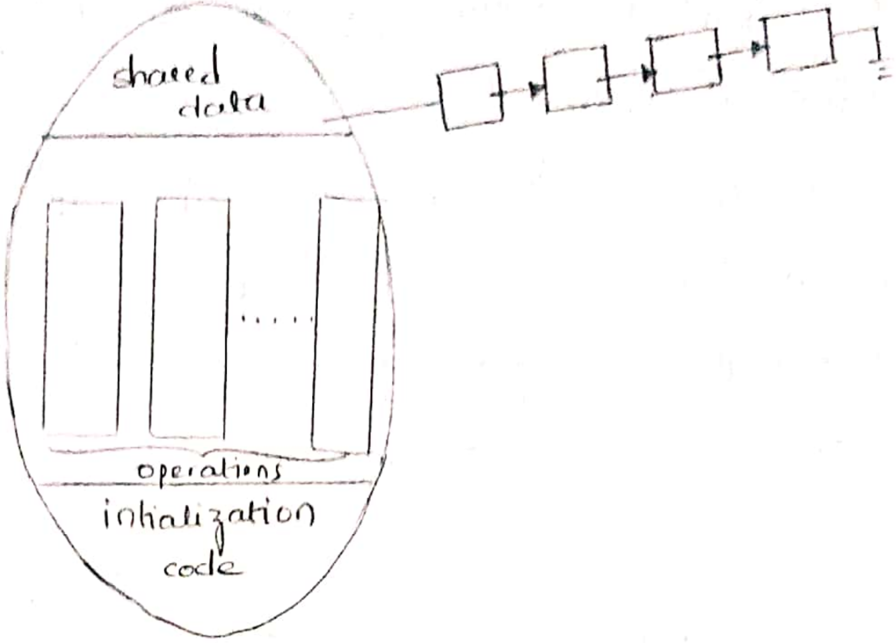
2. Signal and Continue: Q either waits until P leaves the monitor or waits for another condition.

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 ~~invol~~ invokes the putdown() operation. Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
        .....
         eat
        ......
DiningPhilosophers.putdown(i);
```

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

```
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;
    test((i+4) % 5);
    test((i+1) % 5);
}
 void test(int i) {
if ((state[(i+4) % 5] != EATING) &&
                  (state[i] ==HUNGRY) &&
```



Dining-philosophers Solution using Monitors

we illustrate monitor concepts by presenting a deadlock-free solution 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 datastructures.

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

Philosopher i can set the variable state [i] = EATING only if her two neighbors are not eating: (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 herself when she is hungry but is unable to obtain the chopsticks she needs.

We are now in a position to describe our solution to the dining-philosophers problem. The distribution of the chopsticks is controlled by the monitor Dining-philosophers. Each philosopher, before starting to eat, must invoke the operation pickup( ).

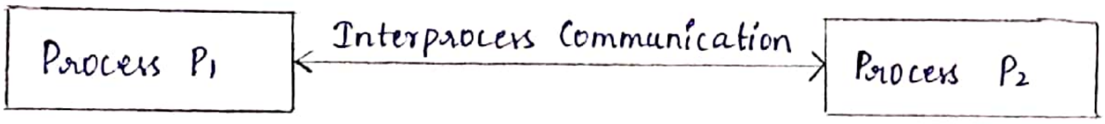
```
            (state [(i+1) % 5] != EATING)) {
          state [i] = EATING;
          self [i].signal ();
       }
    }
    initialization.code() {
       for (int i = 0; i<5; i++)
          state [i] = THINKING;
       }
    }
```

→ A monitor solution to the dining-philosopher problem.

UNIT - 3.3 Inter Process Communications (IPC)

→ Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other.

→ This communication could involve the transfering of data from one process to another.



Approaches to Interprocess Communication.

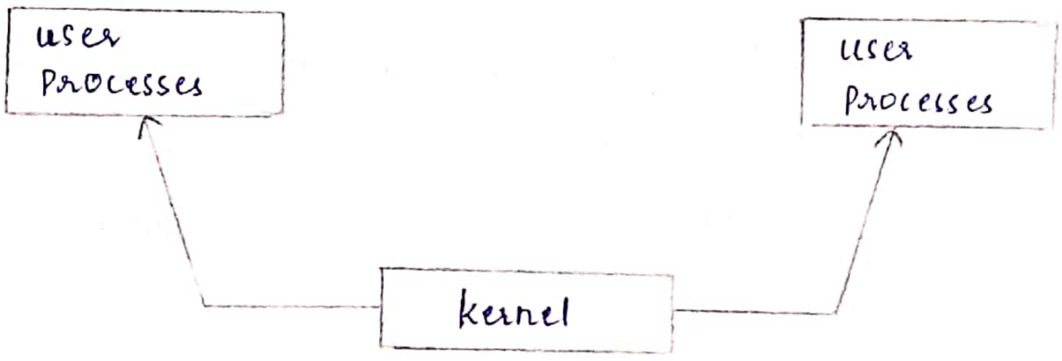
Communication can be of two types:-

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

→ Between unrelated processes, or two or more different processes.

IPC between processes on a single computer system:-

→ processes can share memory either in the user space or in the system space. This is equally true for uniprocessors and multiprocessors.

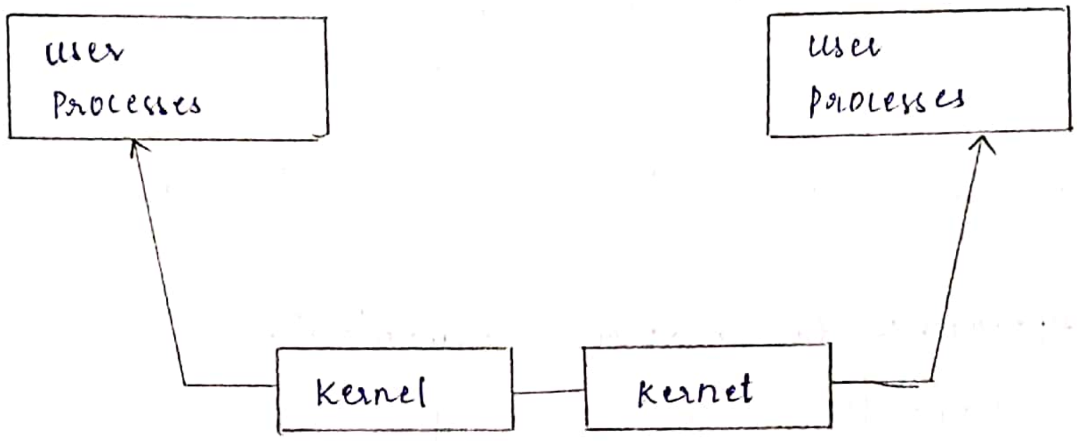


IPC on same host.

IPC between processes on different systems.

→ The computers donot share physical memory, they are connected via network.

→ Therefore the processes residing in different computers cannot use memory as a means for communication.



IPC on different hosts.

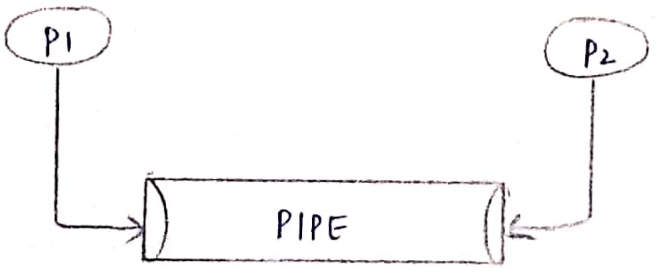
PIPES

→ pipes are unidirectional bytes streams which connect the standard output from one process into the standard input of another process.

→ A pipe is created using the system call pipe ( ) that returns a pair of file descriptors.

Syntax int pipe (int * filedes);

→ The pipe is mainly used to communicate between two threads in a single process or between parent and child process.



→ pipes can only connect the related process.

→ If the writing process is faster than the reading process which consumers the data the pipe cannot store the data

→ In this situation the writer process will block until more capacity becomes available

→ pipe is a main memory buffer.

→ If the reading process tries to read data when there is no data to read, it will be blocked until the data becomes available.

→ By this, pipes automatically synchronize the two processes

Description.

→ Call to the pipe() function which returns an array of file descriptors fd[0] and fd[1]

→ fd[1] connects to the write end of the pipe, fd[0] connects to the read end of the pipe.

→ Anything can be written to the pipe, and read from the other end in the order it came in.

→ It can be used only between parent and child processes (related process)

→ When we use fork in any process, file descriptonors. 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.

Algorithm

```
Step 1 : Start the program
Step 2 : Create a pipe using pipe ( ) system call
Step 3 : Create a child process.
         If the child process is created successfully then write
         the message into pipe otherwise go to step 2
Step4 :  Read the message from the pipe & display the message.
Step5 :  Stop.
```

```
#include <stdio.h>
#include <unistd.h>
# include <sys / types.h>
# include < sys / wait.h>
# include < string.h>
int main ( )
{
  int fd [2], child ;
  char a[ ] =(" Hello Sreyas\n");
  pipe (fd);
  child = fork ( );
  if ( ! child)
  {
    close (fd [0]);
    write (fd [1], a, strlen (a)) ;
    wait (0);
  }
  else
  {
    close (fd (1]);
    read (fd [0], a, size of (a));
    printf ("\n\n The string retrieved from pipe is %s", a);
  }
  return 0;
}
```

FIFO's (or) Named pipes.

→ communication between unrelated process can be achieved using FIFO (or) Named pipes

→ Processes communication with FIFO's do not need to have any kind of relationship. They can be independently executed programs on a system

→ FIFO is created on disk and has a name

→ Like a file, a FIFO has to be created and opend before using it for communication.

→ mkfifo( ) system call is used to create a FIFO special file ~~he~~ with pathname

Syntax. : int mkfifo (const * char pathname, mode_t mode);

mode specifies the FIFO permissions.

→ On success mkfifo ( ) return 0
In case of an error '-1' is returned

→ Once a FIFO special file is created, any process can open it for reading or waiting, in the same way as an ordinary file

→ However, it has to be open on both ends simultaneously before to proceed to do any input or output operations unit.

ALGORITHM

Step 1 : Create two processes one is FIFO writer and another one is FIFO reader

Step 2 : Write process performs the following:

→ Creates a named pipe (using system call mkfifo( )) with name "MY FIFO" if not created

→ opens the named pipe for write only purpose

→ writes data into the FIFO

Step 3 : Reader process performs the following
→ Opens the named pipe for read only purposes
→ Reads the content from the FIFO and put into the Buffer
→ write the content out from the buffer on the screen.

→ In client server applications FIFO's are used to pass data between a server processes and client process.

```
/* FILEname: fifowriter.c */
#include<stdio.h>
# include<sys/stat.h>
# include <sys/types.h>
# include <fcntl.h>
# include < unistd.h>
# inclue < string.h>
# define FIFO_FILE "MYFIFO"
int main( )
 {
   int fd;
   char buffer[50] = "WRITER DATA";;
   /* create the FIFO */
   mkfifo(FIFO_FILE,0666);
   fd=open( FIFO_FILE, O_WRONLY);
   write (fd, buffer, sizeof (buffer));
   close ( fd);
   return 0;
}
```

```
/* Filename: fifo read.c */
# include<stdio.h>
# include < sys/stat.h>
# include < sys/ types.h>
# include < fcntl.h>
# include < unistd.h>
# include < sting.h>
# define FIFO_FILE " myFIFO"
int main( )
{
  int fd1;
  char buffer[80];
    fd1 = open(FIFO_FILE,O_RDONLY);
    read (fd1, buffer, sizeof (buffer));
    printf(" has been sent by writer");
     write (1, buffer, sizeof (buffer));
     close (fd1);
     return 0;
 }
```

Output: " WRITER DATA has been sent by writer".

Step1: First compile the program for writing message into the fifo

Step2: Then run the program

Step 3: Now open another terminal and compile the program for reading the message from FIFO

Step4: Run the program.

MESSAGE QUEUES.

→ A message queue is a linked list of message stored within the kernel and identified by a message queue identifier

→ A new queue is created or an existing queue is opened by msgget( )

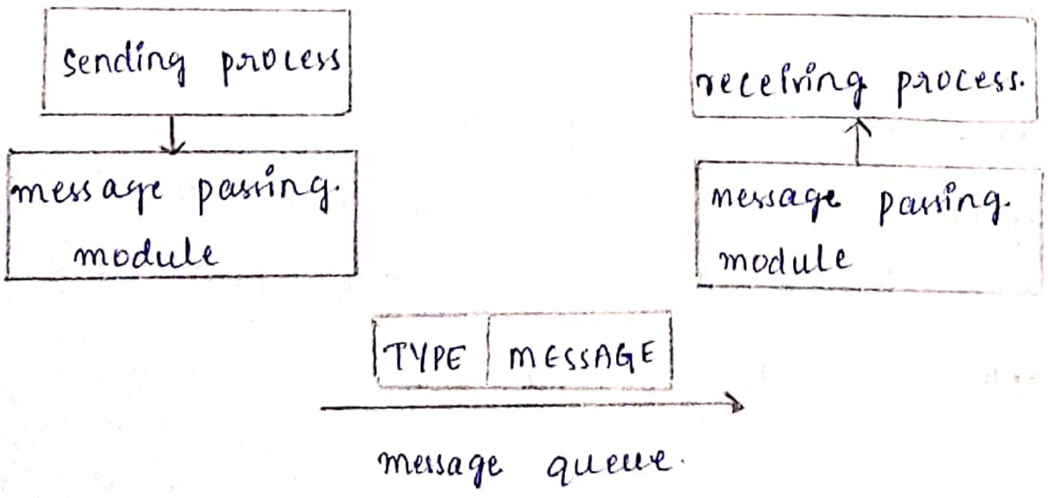
→ New message are added to the end of a queue by msgsnd( )

→ Messages are fetched from the queue by msgrcv( ). we can fetch the message based on their type feild. we don't have to fetch the messages in FIFO order

→ All processes can exchange information through access to a common system message queue. The sending process places the message (via OS message passing module) onto a queue which can be read by another process.

→ Each message is given an identification or type so that processes can select the appropriate message

→ process must share a common key in order to gain access to the queue,



System calls used for message queues:

ftok ( ): to generate a unique key

msgget ( ): returns the message queue identifier for a newly created message queue.

msgsnd ( ): Data is placed onto a message queue by calling msgsnd ( )

msgrcv ( ): messages are retrieved from a queue.

msgctl ( ): It perform various operations on a queue.

ALGORITHM (writer process)

Step 1: Generate an IPC key by invoking the ftok function A filename and ID are supplied while creating the IPC key.

Step 2 : Invoke the message msgget function to create the message queue The message queue is associated with the IPC key that was created in step 1.

Step 3 : Define the structure with two members mesg-type and mesg_text. Set the value of mesg-type to 1

Step 4 : Enter the message that is to be added to the message queue. The string that is entered is assigned to the mesg-text of the structure that was defined in step 3.

Step 5: Invoke the msgsnd function to send the entered message into the message queue.

ALGORITHM (Reader process)

Step 1 : Invoke the ftok function to generate the IPC key. The filename and ID are supplied while creating the IPC key. These must by same as what were applied while generating the key for writing the message in message queue

Step 2 : Invoke the msgget function to access the message queue that is associated with the IPC key. The message queue that's associated with this key already contains a message that we wrote through the writer's program.

Step 3 : Define a structure with two members, mesg-type and mesg_text.

Step 4 : Invoke the msgrev function to read the message from the associated message queue, The structure that was defined in step 3. is passed to this function.

Step 5: The read message is then displayed on the screen.

```
//C program for message queues (writer process).
#include <stdio.h>
# include <sys/ipc.h>
# include < sys/msg.h>
# define MAX 10
// Structure for message queue
  struct mesg_buffer
   {
      long mesg_type ;
      char mesg_text [100];
   } message ;
   int main( )
   {
      key_t key ;
      int msgid ;
      // ftok to generate unique key.
      key = ftok ("progfile", 65);
      // msgget creates a message queue and returns identifier
      msgid = msgget (key, 0666 /IPC_CREAT);
      message. mesg_type = 1 ;
      printf (" write Data :");
      fgets (message. mesg_text, MAX, stdin);
      //msgsnd to send message
      msgsnd (msgid, & message, size of (message), 0);
     // display the message.
      printf (" Data send is : %s\n", message.msg_text);
      return 0;
  }
```

```
// C program for message queues (reader process)
# include<stdio.h>
# include< sys/ipc.h>
# include< sys /msg.h>
// structure for message queue
  struct mesg- buffer
  {
     long mesg- type;
     char mesg- text [100];
  } message;
  int main( )
  {
     key_t key;
     int msgid ;
     // ftok to generate unique key.
     key = ftok ("progfile", 65);
     // msgget creates a message queue and returns identifier
     msgid = msgget (key,0666 |IPC_CREAT);
     // msgrcv to receive message.
     msgrcv (msgid, & message, sizeof (message), 1,0);
     // display the message
     printf ("Data received is %s\n", message.msg-text);
     // to destroy the message queue
     msgctl (msgid, IPC - RMID, NULL);
     return0;
  }
```

Output :

Step1: First compile the program for writing message into the queue

Step2: Then run the program.

→ output of the program for writing the message into the message queue :-

→ Enter a message to add to message queue: Good day.

→ The message id Goodday.

Step3: Now open another terminal and compile the program for reading the message from the queue.

Step4: Then run the program.

→ Output of the program for reading the message from the message queue.

→ The message received id Goodday.

SHARED MEMORY.

→ Shared memory is the interprocess communication mechanisim

→ IPC through shared memory is a concept where two or more processes can access the common memory, and communication is done via this shared memory where changes made by one process can be viewed by another process.

→ The drawback with pipes, fifo and message queue is that for two processes to exchange information has to go through kernel.

→ Server reads the input file and 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 clients buffer.

→ Finally the data is copied from the client's buffer.

→ So, a total of 4 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

System calls used are :-

ftok(): is used to generate a unique key.

shmget(): int shmget (key_t, size_t size, int shmflg) upon successful completion, shmget() returns an identifier for the shared memory segment.

shmat(): Before we can use a shared memory segment to prog must be attached to shared memory using shmat().

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

shmid is shared memory id.

shmaddr specifies specific address to use but we should set it to zero and os will automatically choose the address.

shmdt(): when done with the shared memory segment program should detach itself form it using shmdt()

int shmdt (void * shmaddr);

shmctl(): To destroy the shared memory shmctl() is used

shmctl (int shmid, IPC_RMID, NULL);

ALGORITHM (writer process)

Step1: Generate an IPC key by invoking the ftok function. A filename and ID are supplied while creating an IPC key.

Step2: Invoke the shmget function to create the shared memory. The shared memory is associated with the IPC key that was created in step 1

Step3: Invoke the shmat function to attach prog. to shared memory.

Step4: Enter the message that's going to be added to the shared memory The string thats entered is assigned to the str.

Step5: Invoke the shmdt function to detach prog from shared memory.

ALGORITHM (reader process)

Step1: Invoke the ftok function to generate the IPC key. The filename and ID are supplied while creating the IPC key. These must be the same as what were applied while generating the key for writing the message in the shared memory.

Step 2 : Invoke the shmget function to access the shared memory that is associated with the IPC key. The message queue that's associated with this key already contains a message that was written by writer process.

Step 3 : Invoke the shmat function

Step 4 : Display the message that's been written to shared memor

Step 5 : Invoke the shmdt function

Step 6 : Invoke the shmctl function.

// program for writing into shared memory.

```
# include < stdio.h>
# include < sys / ipc.h>
# include < sys / shm.h>
# include < stdio.h >
# include < stdlib.h >
int main( )
{
   char * str ;
   int shmid ;
   Key_t key = ftok ("shared mem", 'a');
   if ((shmid = shmget (key ,1024,0666 |IPC_CREAT)) <0)
    {
      perror ("shmget");
      exit(1);
    }
   if ((str = shmat (shmid, NULL, 0)) = =(char * -1)
     {
       perror ("shmat");
       exit(1);
     }
printf(" Enter the string to be written in memory :");
gets (str);
printf(" string written in memory: %s\n", str);
shmdt (str);
return 0;
}
```

// program for reading from the shared memory.

```
# include<stdio.h>
# include <sys/ipc.h>
# include <sys/shm.h>
# include <stdio.h>
# include <stdlib.h>
int main ( )
{ int shmid ;
  char * str ;
  key_t key = ftok (" shared mem", 'a');
  if ((shmid = shmget (key, 1024, 0666 |IPC_CREAT)) <0)
   {
     perror ("shmget"),
     exit (1);
   }
   if ((str == shmat (" shmid, NULL,0)) == (char*-1)
    {
     perror ("shmat");
     exit (1);
   }
printf("Data read from memory: %s\n", str);
shmdt(str);
shmctl(shmid, IPC_RMID, NULL);
return 0;
}
```

OUTPUT:

Step 1: First compile the program for writing message into the shared memory

Step 2: Then run the program.

→ Output of the program for writing the message into the shared memory.

```
→ Enter the string to be written in memory : Hello Sreyas
→ String written in memory : Hello Sreyas..
```

Step 3: Now open another terminal and compile the program for reading the message from the shared memory

Step 4: Then run the program

→ Output of the program for reading the message from the shared memory.

```
→ Data read from memory : Hello Sreyas.
```