CHAPTER 3:

Process Synchronization

Synchronization is a good word used for controlling & coordinating of processes.

A large amount of CPU bound & IO bound processes can be working concurrently in a system.

All the processes want the resources at the same time.

OS must manage them.

sync means controlling them.

when multiple concurrent processes are competing for shared resources there'll be a problem.

In a pgm, there will be a critical section (part of code) (ie segment of code) which will be actually doing something (like ~~reading~~, updating, writing, deleting, etc) (whose action really brings a change).

common variables, common files, common tables.

common means they are shared.

process is ntg but program in execution

part of code which actually does something, which utilizes resources to do its task we can say.

So, all processes will have a critical section.

OS → $P_0, P_1, P_2, \ldots P_{n-1}$

Critical Section → CS

update

Same data & same resource

When concurrent processes want to enter the critical section it creates critical section problem.

P_0 & P_1 simultaneously ಇರಬೇಕಾಗುವಂತೆ data ನ್ನು update/modify ಮಾಡಲಕ್ಕೆ try ಮಾಡಿದಾಗ, ಆಗ ಮಾಡಲಕ್ಕೆ ಆಗದೆ data inconsistency ಆಗುತ್ತದೆ. ಆಗ ಹೀಗಾಗುವುದು ಮುಂದೆ mutual exclusion concept was brought into action.

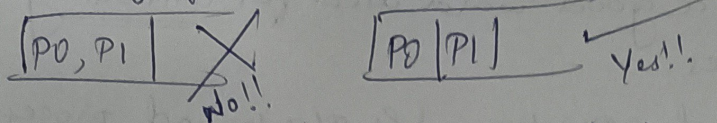
If concurrent processes try to do changes/update same data at the same time, then it leads to data inconsistency.

So, this problem of synchronization is called critical section problem.

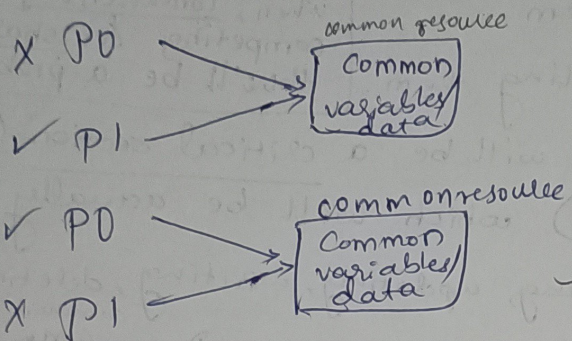
It is because of critical section of the code. It is only modifying the shared data (ie common data).

multiple & concurrent processes are different

We never saw two processes executing at the same time.
<Take eg. of Gantt chart only:



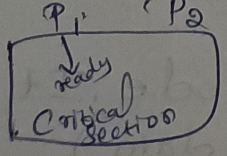
While one process is updating, the other process shouldn't be allowed to update the same data.
It is called as mutual exclusion.

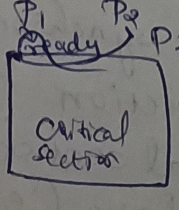


mutual exclusion is a property which should be present in the solution for critical section problem.
means the solution should satisfy three things for Critical Section Problem

1] Mutual Exclusion
2] Progress
3] Bounded Waiting

* Progress - When no processes are in their critical section (means when they're not executing that code segment which does actual work / change),
The other processes, which are ready to execute their critical section must be allowed to do so. They shouldn't be stopped by the processes which are not executing their critical section.





P2 is stopping it means there is no progress.

P2 should not stop P1, who is ready to execute its critical section.
P2 stop execution P1 both, we say there is Progress.

Bounded Waiting

No process should wait endlessly. They should get a chance to enter their critical section.

when any process enters its critical section, we say that it has actually executed. Bcz it is a core part of that process. (completes its task & exit from the critical section)

But whenever it wants to execute its ~~execute~~ critical section, before entering its critical section, it has to ask permission of OS. Bcz at that time any other process might be executing its critical section.

request general structure of a process:

entry section ← segment of code which asks permission to OS.

If allowed, it executes / implements.

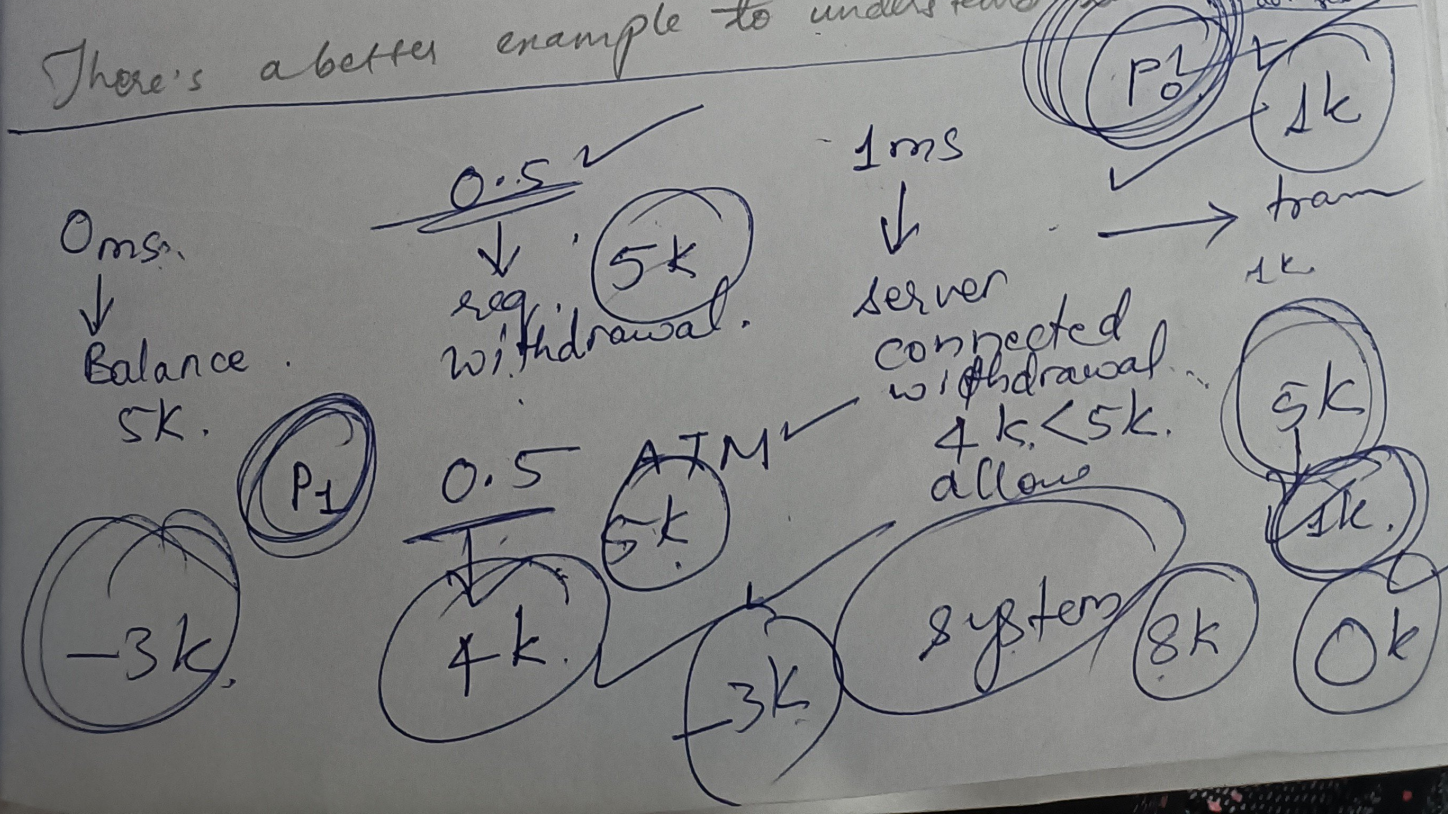
critical section → operation of the process

informing

exit section ← tells everyone that this process has completed its execution & tells that it is coming out of the exit critical section. (Now any one who needs that resource can use, it will tell.)

remainder section } code, like remaining part of code remaining at last after main action is over. Last part is remaining, dummy part which such code which don't need any resources

There's a better example to understand this



The designer has to write algorithm to solve this problem. OS don't have brain right!!

- The-File System.

* Process Synchronization

Textbook :- Operating System Concepts 8th Edition
by Abraham Silberschatz
Peter B. Galvin
Greg Gagne

Synchronization

- A cooperating process is one that can affect or be affected by other processes executing in the system.
- Concurrent access to shared data may result in data inconsistency. However, if we use the various mechanisms available to ensure the orderly execution of cooperating processes that share a logical address space (i.e. both code & data), so that data consistency is maintained.
- A situation in which several processes access & manipulate the same data concurrently & the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard the race conditions, we need to ensure that only one process at a time can be manipulating the data.

* Situations such as the one just described occur frequently in OS as different parts of the system manipulate resources. Furthermore with the growth of multicore systems, there is an increased emphasis on developing multithreaded applications wherein several threads - which are quite possibly sharing data - are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. So only we'll see process synchronization & coordination amongst coopelating processes.

The Critical-Section Problem

Consider a system consisting of n processes $[P_0, P_1, \ldots P_{n-1}]$. 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. The critical-section problem is to design a protocol that the processes can use to cooperate (ie to ensure that the Race condition among the processes will never arise.)

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

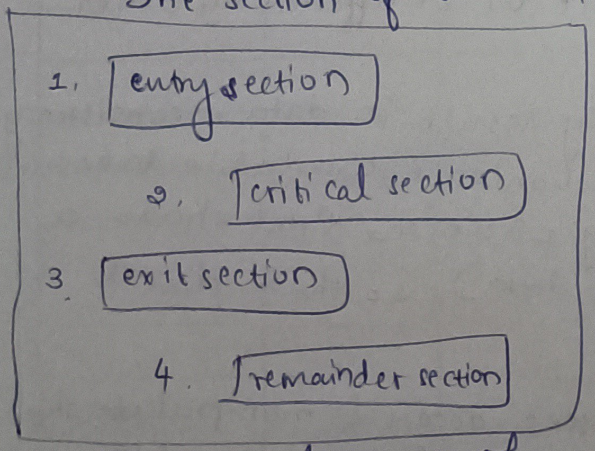


fig:- General structure of a process

written neatly like this

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

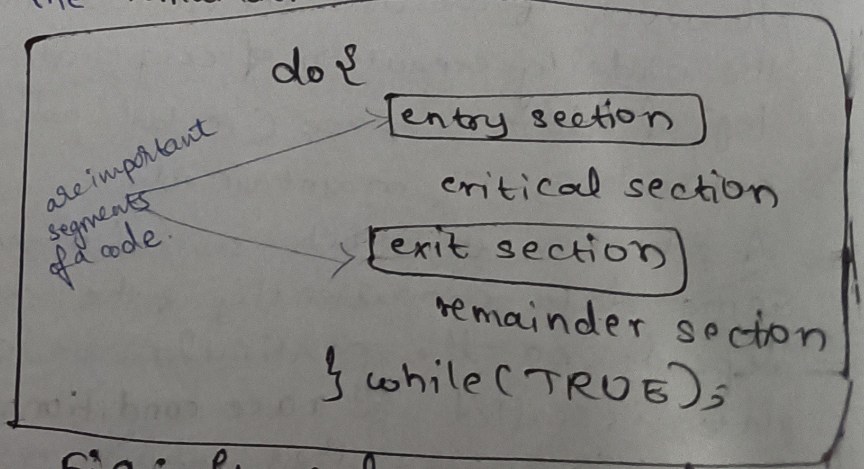


Fig: General structure of a typical process P_i.

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

1. Mutual exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical section.
2. Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded waiting: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before the request is granted.

Peterson's solution

* Peterson's solution is one among the available solutions for the Critical section problem, which is very old/obsolete.
* It doesn't work on modern computer architectures.
* Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
* The 2 processes will share 2 variables/data items:

```
int turn;
boolean flag[2];
```

The variable **turn** indicates whose turn it is to enter its critical section.
The flag array is used to indicate if a process is ready to enter its critical section. For eg, if flag[i] = true, then this value indicates that process P_i is ready to enter its critical section.

The structure of process P_i in Peterson's solution.

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

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

The structure of process P_j in Peterson's solution.

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

        critical section

    flag[j] = FALSE;

        remainder section

} while(TRUE);
```

tells that
Whenever the flag = true, the process is ready to go into the critical section to OS.

Structure of process P_i in Peterson's solution.

flag[i] = true; ← P_i is ready to enter its critical section.

turn = j; ← P_i is ready but if P_j wants to go execute its critical section, let it execute.

P_i is humble.

If P_j wants to execute & turn भी उसका ही है, तो let it do.

while (flag[j] && turn == [j]);

↑ infinite loop.

when flag[j] becomes false,

P_i will executes its own critical section.

when it completes its task,

flag[i] is set to false.

then it'll enter its remainder section & finally exits from the system.

Structure of process P_j in Peterson's solution.

flag[j] = true; ← means P_j is ready to enter its critical section.

turn = i; ← P_j is ready but P_i ~~का~~ turn ~~है~~ ✓ तो let P_i enter its critical section & complete its task, it'll tell.

while (flag[i] && turn == [i]);

(> means ~~अगर~~ P_i ready ~~है~~ ✓ ~~और~~ P_i ~~का~~ ~~ही~~ turn ~~है~~ ✓ ~~तो~~ let P_i execute & complete its task.

& P_j will be stuck here only.

when the condition in the loop become false, then P_j will enter its critical section.

When it completes its task,

flag[j] is set to false.

then it'll enter its remainder section & finally exit from the system.

Take a Scenario when 2 processes come at same time.

P_i — array

flag[i] = true;

Bcz of its humbleness,

turn = j;

P_i ने turn का value j कर दिया

P_j — array

flag[j] = true;

P_j is also humble. So it updates value of

turn = i;

P_j ने turn का value i कर दिया

turn = i

Now turn = i bcz it is recently updated. (changed recently by P_j)

(So the process P_j which is not in its remainder section decided that P_i should get a chance. So this is progress.)

while (flag[j] && turn == [j]);

True, × false

as the condition is false, it'll break the infinite loop & P_i enters its critical section.

critical section

after finishing its task, it'll set

flag [i] = false;

(> means resource is released.

exit section

If you don't do this P_j will keep on waiting, waiting & will be in waiting state.

P_i enters its remainder section

& later go out of memory / system.

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

True, True,

as semicolon is there, it is infinite loop.

So P_j is stuck here.

Now, P_i has finished its task & flag[i] = false,

Now condition in while loop becomes false

(flag[i] && turn[i]);

→ false, → True

False && True = false

So infinite loop is broken & P_j enters its critical section.

So,

& ne

Pete

So

w

O

th

P_j after coming out of critical section, will set

flag[j] = false ; ← means resource is released.

& enters its remainder section & later go out of system.

So, you can see that P_j waited only for some time & not infinitely. So bounded wait is achieved in Peterson's solution.

So Peterson's solution satisfies all 3:
1) mutual exclusion, 2) Progress, 3) Bounded wait.

When both processes P_i & P_j ~~are~~ flag=false, OS will come to know that both have completed their task & it will kill them.

Semaphores

* Semaphores is also a solution to the critical section problem.
* It was proposed by Edsger Dijkstra, a Dutch computer scientist.
* It is also software based.
* A semaphore S is an integer variable (which takes only non-negative values) that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

The wait() operation (orginally termed P) means "to test".
→ To test whether other processes are into their critical section.

The signal() operation (originally termed V) means "to increment".
→ To signal other processes that this particular process has finished executing its critical section & other processes are free to enter their critical section.

Semaphores

Semaphores is also a solution to Critical section problem.

It is also software based.

Proposed by Edsger Dijkstra, a Dutch computer scientist.

He told that, lets use only one integer variable with non-negative value/number.

Whichever process wants to enter its critical section, must make use of 2 functions wait() & signal().

→ atomic operations

↓

separate / independent or indivisible.

means when any process is invoking wait(), it should not invoke signal() & vice-versa.

wait() is indicated by letter P.

↳ "means to test"

to check or test whether other processes are into their critical section

signal() → letter V → means to "increment"

↳ to signal other processes that particular process has finished executing its critical section & other processes are free to enter their critical section.

* P & V must work on an integer variable S. S → semaphore

while (S <=0);

If S value -ve or zero this while loop = true. Infinite loop. This process is stuck here. means other processes are into their critical section. means it is locked.

when this condition becomes false, first thing what this process does is, it'll decrement the value of S. So that other processes will come to know that this particular process is executing its critical section.

---

2 Types of Semaphore
1) Binary semaphore
2) Counting semaphore

* wait() & signal() must be used independently & not concurrently by multiple processes.

* S → non-negative value

Binary Semaphores → Binary semaphores never take on values other than zero or one, and counting semaphores on the other hand take on arbitrary non-negative values.

S=1 means resources are free

Binary 0/1

We'll consider initial value of S as 1 bcz when wait() is invoked by any process, while(s<=0); loop becomes false. It'll break the while loop & S-- will happen.

i.e 1-- will result zero i.e. 0.

Now, S is having a non-negative value.

When S becomes zero i.e S=0, the resources are busy.

Illustration using Binary Semaphore:

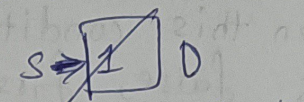
S=1 assume initially resources are free.

- P1 will come
- P1 will test first by using → or invoking wait() operation.

P1 → wait(Semaphore S){

while(1<=0); False so break it

S--;



Now S=0

}

- P1 has locked its critical section & is executing it.

Now P2 comes.

P2 will invoke wait() operation.

P2 → wait(Semaphore S){

while(0<=0); → True so, it'll get stuck over here.

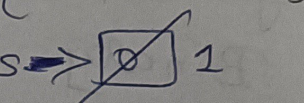
S--; ← { it'll not come here means it is not supposed to execute its criticalsection

}

Like this we achieved mutual exclusion → by using Semaphore value. So they are called as mutex locks.

- P1 → signal(Semaphore S){

S++;



Now S=1

}.

busy.

• P2 को wait() operation में जो while loop है वो false हो जायेगा and P2 starts executing, by making

S--; ie S = 0 now.
1--

means P2 has locked its critical section.

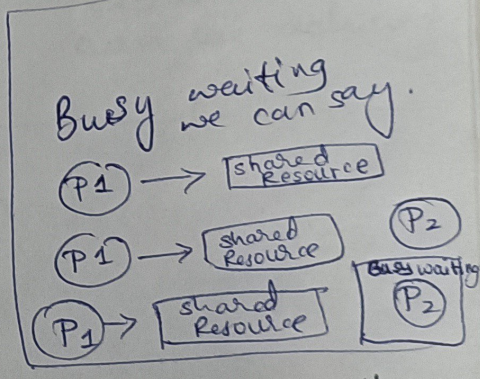
• Progress & bounded wait are also achieved.

• Here, processes who wish to execute their critical section are only taking decision that whether they've to execute their critical section or not.

Disadvantage of Semaphores

1) They (semaphore) create busy waiting
The CPU has to process this while statement again & again. The precious CPU time is stuck into one statement until while loop condition becomes false (or we can say semaphore is released).

```
P(semaphore S)
{
    while(s<=0);   → True
                   ← So CPU is stuck here.
    s--;
}
```



This type of semaphore is also called a spinlock bcz the process "spins" while waiting for the lock.

So, how this spinlock is resolved?
-> is in textbook. (Just one line of code is added to previous code).
Find out.

Counting Semaphores

Assume Now P1 requires 4 number of resources.

S = 4

whenever, it gets one resource, S is decremented

S—— ie S = 3

maybe these 3 resources are busy, then it has to wait in while loop.

when any one becomes free

S-- ie S=2

means 2 resources are locked by P1 & waiting for 2 resources.

finally when it gets all 4 resources, S = 0

& it starts executing its critical section.

Semaphores are called as mutex locks. bez, by using semaphores, mutual exclusion can be achieved.

mutual exclusion

↘ ↙

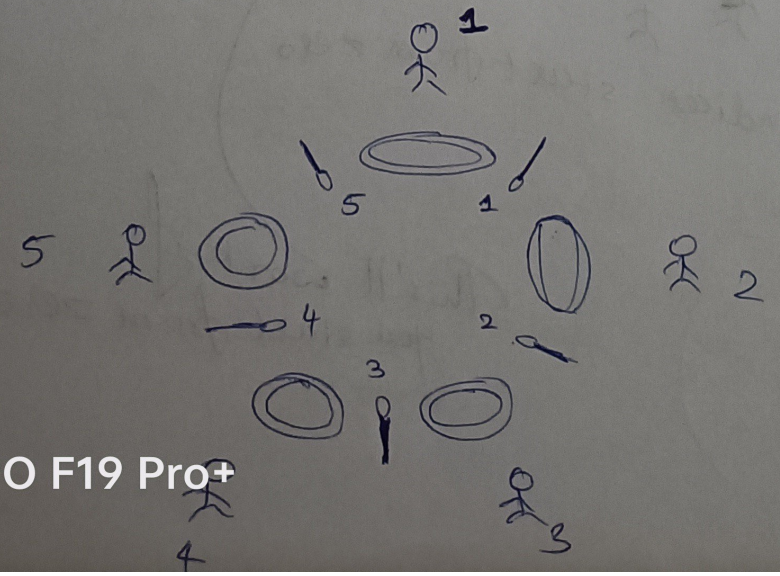
mutex

Dining-Philosophers Problem

Persons are processes.

Bowl is critical section

chopsticks are resources which are limited



Philosophers

↳ entire life they think or ~~eat~~ but not do both simultaneously.

III rly

Processes wait or execute but don't do both simultaneously.

* Processes don't interact with one another.
* chopsticks are assumed as semaphores. (ie Binary Semaphores)
  We are using binary semaphores.
  we'll initialize all 5 chopsticks
  ie semaphores to 1 initially.

S = 1 free
S = 0 busy

* Person 1 can utilize chopstick no. 5 & 1.

Person 2 " " 1 & 2
" 3 " " " 2 & 3
" 4 " " " 3 & 4
" 5 " " " 4 & 5

This will not work if you consider 1 to 5 chopstick.

1	2 %. 5 = 2
2	3 %. 5 = 3
3	4 %. 5 = 4
4	5 %. 5 = 0

This will not work

If we do (i+1) %. 5

But as it is array, you consider from zero.

0, 1, 2, 3, 4, 0

By default indices start from zero

(i %. 5 + 1) will work if you consider 1 to 5 chopsticks

1 %. 5 = 1	1 + 1 = 2 — 1 & 2
2 %. 5 = 2	2 + 1 = 3 — 2 & 3
3 %. 5 = 3	3 + 1 = 4 — 3 & 4
4 %. 5 = 4	4 + 1 = 5 — 4 & 5
5 %. 5 = 0	0 + 1 = 1 — 5 & 1

if you start from 1

This will work in the algorithm.
[i] & [(i %. 5) + 1]

This'll work if you start from zero.

Algorithm

```
wait (chopsticks[i])
wait (chopsticks[(i+1)%5])
    //eat
signal (  "  )
signal (  "  )
    //think
```

- One process cannot hold some processes & wait for some other processes which are held by other processes. It may lead to deadlocks.
- When all the resources needed for execution of a process are available to it, only then that process executes.
- Only when, both the forks/spoons are free, the person will eat, in Dining table problem.

Multiprogramming

↳ multiple programs are executing at the same time.

Unit 3

Deadlocks

Deadlock is a state, where all of the processes are stuck & none of them are executing.

In multiprogramming, when multiple processes are there, concurrent processes are there, deadlocks will come into picture.

- First problem was critical section problem.
  Solution was Semaphore.
- Second problem is Deadlocks.

→ First try to prevent deadlock.
Even then if it occurs try to recover deadlocks.

V. Imp Question

Deadlocks

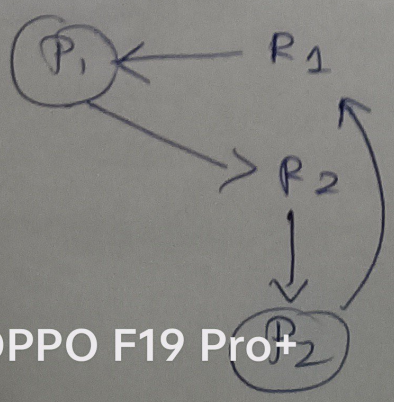
There are 4 conditions in which a deadlock can occur ←

Conditions : / characteristics of a deadlock.

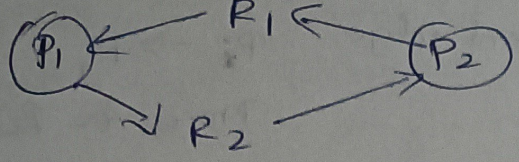
1] Mutual Exclusion (when mutual exclusion is absent).
2] Hold & wait (when a process holds / allocates a resource & at the same time wait for other resources)
3] No preemption
4] Circular wait

↑ If any one of these is true, then there is a chance of deadlock to occur. But surely it will occur we can't say.

★ Hold & wait



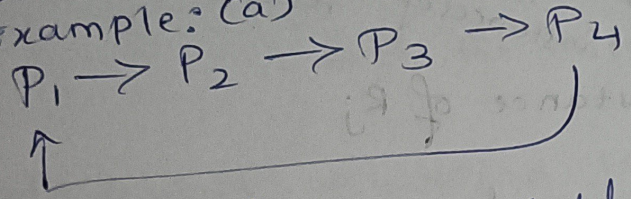
P_1 Holds R_1.
& also waits for R_2 to complete its execution but R_2 is being allocated to P_2 And P_2 is requesting R_1 which is held by P_1



So; both will not execute

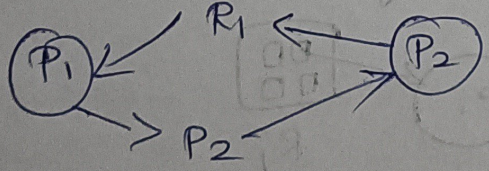
3) Preemption
↳ Cutting of execution of a process & giving chance to the other (may be of higher priority).
Sometimes Preemption is necessary.

4) Circular wait.
example: (a)

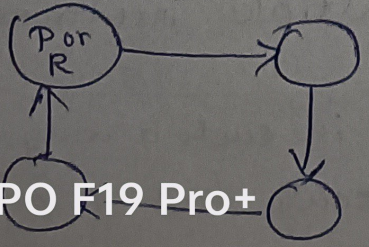


P1 waiting for resource held by P2
P2 " " " " P3
P3 " " " " P4
P4 " " " " P1

example: (b)



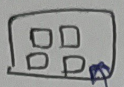
RAG : Resource - Allocation Graph
In OS,
A vertex of a graph represents a process or a resource.

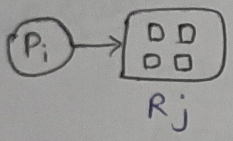


It can be a process or a resource.

edges are of 2 types
1) request edge.
2) assignment edge or allocation edge

- request edge - directed edge $P_i \rightarrow R_j$
  Process to Resource
- assignment edge - directed edge. $R_j \rightarrow P_i$ (Resource to Process)
  Process has already consumed or currently using a resource

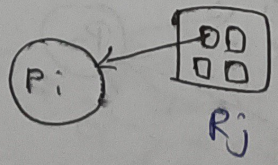
- Process ○ If a vertex is to be represented as a process
- Resource Type with 4 instances  resource
  different/multiple instances of same resource
- P_i requests instance of R_j

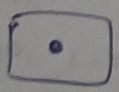


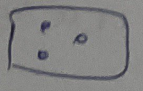
The arrow from P will not go inside boundary of the resource.

- P_i is holding an instance of R_j

Here arrow comes from inside the resource to boundary of the process.



 ← resource with single instance

 ← resource with multiple instances.

Sometimes OS are designed in such a way that deadlocks are allowed to occur but recovery methods are written for them.

Eg. of RAG

R1 → Unless & Until it is free, other processes can't take it. as it is single instance R

$P_1 \rightarrow R_1$ P_1 requests R_1

$P_1 \leftarrow R_2$ one instance of R_2 is held by P_1

P_2 P_2 is holding R_1 & requesting R_3 which is held by P_3

R_4 is completely freely.

Both the instances of R_2 are not free.

Hold & wait situation is present.

If a cycle exists then we can say a deadlock may occur in RAG.

> But only if a cycle is present, then we can't say that a deadlock will surely occur.

starts at one process & comes back to the same process

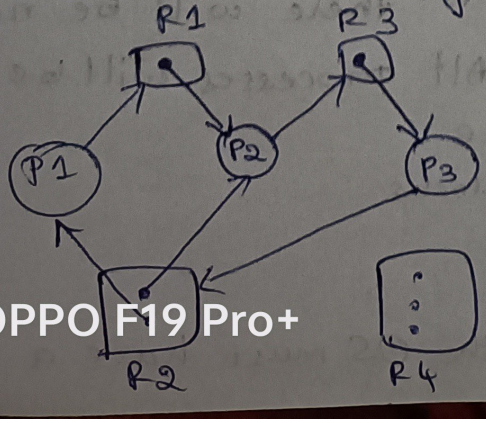
1st cycle:

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

P_1 is requesting R_1 which is held by P_2. P_2 is requesting R_3 held by P_3. P_3 is requesting R_2 which is held by P_1.

A minimum cycle has been formed.

So, we can say, P_1, P_2 & P_3 are in deadlock.



2nd cycle:-

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

Only if a cycle is present we can't ~~surely~~ say that a deadlock will surely occur.

(* If all resources are of multiple instances in a graph then it is a no-deadlock graph. Even if a cycle occurs then there is a chance that no deadlock will occur.)

Eg:-

- If there are no cycles, then no deadlock will occur.
  ↳ in a graph
- If there are multiple instances resource then deadlock possibility is there. (not sure) if a cycle is present.
- If single instance resources are present then a deadlock will occur if a cycle is present.

If a deadlock occurs, then there will be no progress in the system. All processes will be waiting in the ready queue.

- Prevent the occurence of deadlocks
- Avoid " " " "
- Allow " " " "
- If deadlock occurs, the OS must have a

capability of detecting them & recovering them.

Methods for Handling Deadlocks

- Don't allow system to enter a deadlock
- If allowed, recover them
- Just Ignore them.

Deadlock Prevention

1. Mutual exclusion

2. Hold & wait
If a process is holding some resource & requesting for some other resource then OS will check whether the resource requested by that process is free or not. If its not free then OS will come back to the Process and tells that the resource which you are requesting is not free. Kindly release ~~the~~ resources which you are currently holding.

3. No Preemption

4. Circular wait
-> can be avoided when requesting the resources is in order by the process

restrain means avoid

* Deadlock Avoidance
* Deadlock recovery
* Safe/Unsafe state

Deadlock Avoidance

OS must have additional information (max. no. of resources that each process needs to complete its execution. & no. of resources currently used by the process). ← Programmer has to provide these.

Then, it'll lead to sequence of processes which are given resources is a safe sequence / safe state where deadlocks are being avoided.

- Bcz of safe sequence of processes, the system will be in safe state.

But we can not guarantee that the state of the system will be always safe. It can go to unsafe state.

We cannot guarantee Deadlock avoidance.

Example:

Assume total no. of resources of same type, available in the system = 10

is there are 12 instances of resource 'R'. no. of

Example for Deadlock Avoidance:

Assume, the total no. of resources of same type available in the system = 12. ie. there are 12 instances of resource 'R'.

Process	Maximum number of resources needed for the process to complete its execution. (Total or)	Number of currently allocated resources to the process.
P0	10	5
P1	4	2
P3	9	2

So, total how many resources are busy now?

$5+2+2=9$.

∴ 9 resources are busy

ie. by they are Allocated resources.

We have total 12 resources available as per the question. Now $12-9=3$. means (3) resources are free. Now.

Now, P0 needs total 10, out of which 5 are already given.

It still needs $10-5=5$ resources.

P1 still needs $4-2=2$ resources.

P3 " " $9-2=7$ resources.

You have 3 resources free. with this you can satisfy only the process P1 as it needs only 2 & $2<3$. (Now free resources $\Rightarrow 3-2=1$)

So, OS will give 2 resources to P1 & Now, P1 will have $2+2=4$ resources. So, it will execute & exit from the system & released the 4 resources.

Now no. of free resources = $1+4=5$ (↑ earlier one; ← Now released by P1)

Process	Total 'R' needed	Allocated 'R'
P0	10	5
~~P1~~	~~4~~	~~2+2=4~~
P3	9	2

Now, P0 needs 5 more (ie $10-5=5$) Resources

& P3 " 7 more (ie $9-2=7$) Resources.

No. of free resources available now = 5. With these 5, you can satisfy P0. So these 5 Resources are given to P0. now.

P0	10	5+5

So, now no. of free resources = 5-5=0.

P0 ← It executes & exits from the system releasing all the 10 resources which it utilized for its execution.

Now, no. of resources which are free = 0+10 = 10

So, now, P1 completed, P0 completed. P3 is remaining.

Process	Total 'P's needed	Allocated 'R'
~~P0~~	~~10~~	5+5
P1	4	2+2
P3	9	2

P3 needs still 7 resources (∵ 9-2=7). & available free resources are 10. So, as 7<10, you can give 7 resources to P3.

2+7=9. & Now, no. of free resources = 10-7=3

So, P3 will execute & exit from the system releasing all the 9 resources which it utilized for its execution.

So, now, no. of free resources in the system will be

3+9=12.

So, initially 12 resources which were present in the system have become free.

So, order of execution was P1, P0 & P3.

It is a safe sequence as it did not give rise to any deadlock & executed & exited smoothly.

But we cannot guarantee the Deadlock avoidance even when it is a safe sequence.

Now, let's see how an unsafe state can occur.

Let me take the same example.

Process	Max. no. of resources needed to complete ~~its~~ execution by the process.	No. of resources currently allocated to the process.
P0	10	5
P1	4	2
P3	9	2

So, now, total no. of resources which are busy = 5+2+2=9.
Total no. of resources which were present in our system initially = 12.
So, 12−9=3. 3 resources are free now.
Now, if suddenly P3 asks for another resource to OS, OS sees that 3 resources are free. So, It'll give 1 among 3 to P3. So free resources = 3−1=2 now.
So, no of busy resources will become 9+1=10.

Now,

P0	10	5
P1	4	2
P3	9	2+1

Now P1 needs 2 more resources (i.e 4−2=2) to finish/complete its execution.
So, OS will provide it 2. & P1 utilizes 2+2=4 resources in total & completes its task & exits, by releasing 4 resources which it utilized.
So, now no. of free resources = 0+4=4.
Now P0 needs 5 still &
P3 " 6 resources.
But available free resources are 4.
This will create a deadlock situation and none of the processes execute.
So, any process can suddenly ask ~~any~~ for resources like P3 at any point of time (say t1). whether it may be when other process has started execution or has not yet started execution, or anything.

→ even forplacements. Algorithms Banker's algorithm is v.v.Imp.

v.v.Imp

Recovery from Deadlocks

- Process Termination
  - Terminate the processes which are causing deadlocks.
- Resource Preemption

I) Process termination

Even while terminating the process, we must be very careful. What if a process has computed a lot of computations & mathematical calculations & at the end if it is causing deadlock then? what if a process is updating some important data in the database & leading to deadlock!!

II) Resource preemption

→ Selecting a victim
- Here, victim is the resource
- Here, cost matters

→ Roll back (ie Total roll back to the initial stage)
- cost matters.

→ Starvation
- If the same process using the same resource, is causing the deadlock again and again.

v. v. Imp
100%. Q.

Banker's Algorithm

used specially when → means type of the resource is same but multiple copies are there.
- there are Multiple instances of the resource are present

How this name came to this algorithm?
From the amount deposited by people only they'll provide loan to other people.
But what if on the same day all depositors withdraw all of their money?
The same thing is applied here.

- Data Structures for the Banker's Algorithm
1) Available resources
2) Maximum need ie. max. no. of resources a process need
3) Allocated (currently) resources
4) Need ie. still how many resources it needs to complete its task.

In Banker's Algorithm we have 2 algorithms
- Safety Algorithm → gives safe sequence of processes in the system.

10 marks

Safety Algorithm

1. Work = Available
   The available resources in the system.
   Just name change
   Initially none of the processes have executed.

2. whichever process has not executed, see its need.
   if need < work
   ie if need < available, then allocate.

   Allocate होने के बाद execute होगा।
   & later it releases all the resources & goes out of the system. which it utilized in total.

3. Update the total available resources now
   work = work + Allocation
   ↑ earlier no. of resources which were free
   ↑ no. of resources released by the executed process

4. Update the status of the process that it has executed as true (if not as false).

Repeat until all finish their execution

Eg:-

Note:- Snap Shot → means at that particular time what is the status of the system.

Max need cannot exceed total availability of resources

A = 10 total available

Allocation	Max	Available
0		A B C
2		3
3		↓
2		bcz 10 − 7 = 3
2+3+2 = 7		
0		