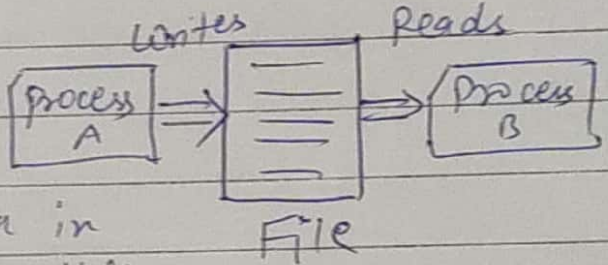
D - II MOD - II MOD - II MOD - II

Process Synchronisation

> It means sharing system resources by processes in such a way that concurrent access to shared data is handled by minimising the chance of inconsistent data.

> Several processes run in an O.S, some of them share resources due to which problems like data inconsistency may arise.

> A & B process do not occur simultaneously.



> One process changing data in a memory locaⁿ where another process is trying to read the data from same memory locaⁿ. It is possible that the data read by process B might be incorrect or erroneous. So, we have to follow synchronisaⁿ means one process will finish writing, then only another process will start reading.

> Process mngmt in OS can be divided into 3 parts:

- Multiprogramming
- Multiprocessing
- Distributed Processing (It involves multiple processes on multiple systems. All of these involve cooperation, competition, and communicaⁿ b/w processes that either run simultaneously).

Scanned by CamScanner

Sunil
Page
Date

Concurrency: It is the interleaving of processes in time to give the appearance of simultaneous execution. ∴ it differs from parallelism which offers simultaneous execution.

Difficulties and Issues in Concurrency

Issues: > Sharing global resources safely is difficult.
> Optimal allocation of resources is difficult.
> Locating programming errors can be difficult.

Q
```
P1() {
① C=B-1;        // B is a shared variable with initial value 2
② B=2*C;
}
P2() {
③ D=2*B
④ B=D-1;
}
```

① ② ③ ④	③ ④ ① ②	① ③ ② ④	③ ① ④ ②
C=1	D=4	C=1	D=4
B=2	B=3	D=4	C=1
D=4	C=2	B=2	B=3
B=3	B=4	B=3	B=2

Race Condition

> It is an undesirable situation that occurs when a device or system attempts to perform 2 or more operations at the same time but because of the nature of the device or the system, the operations must be done in the proper sequence to be done correctly.

> Calculate how many values the shared variable B can have? (above question).
1234, 3412, 1342, 3124, 1324, 3142

Scanned by CamScanner

Critical Section Problem

> A critical section is a core segment that access shared variable and has to be executed as an atomic action. It means that, in a group of cooperating processes, at a given point of time only one process must be executing it's critical section.

> If any other process also wants to execute its critical section, it must wait until first one finishes.

Note Processes can be of 2 types :

① Independent process

② Co-operative Process

> There are the things that processes can share
(i) Shared variable memory, code segment, resources (CPU, Printer, scanner)

```
main()
{
entry section | gets lock (Process is granting permission to enter in its critical section)
Critical section - (Process is executing in its critical section)
exit section | releasing lock (Process is leaving critical section, so that other processes can execute critical section)
Remainder section
}
```

↳ Rest all code segments come in remainder section

Scanned by CamScanner

Synchronisation Mechanism:

Here is a sol^{n} to the critical section problem that must satisfy following request.

(1) Mutual Exclusion (Mutex): Out of a group of cooperating processes only one process can be in its critical section at a given point of time. It is a code segment that prevent simultaneous access to a shared resource. (Primary condi^{n}s)

(2) Progress: If no process is in its critical section and if one or more process wants to execute their critical section, then anyone of these threads or processes must be allowed to get into critical section.

(3) Bounded Waiting: ~~It's~~ After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section because this process request is granted, so after the limit is reached, system must grant other processes to get in its critical section. (Secondary condi^{n}s)

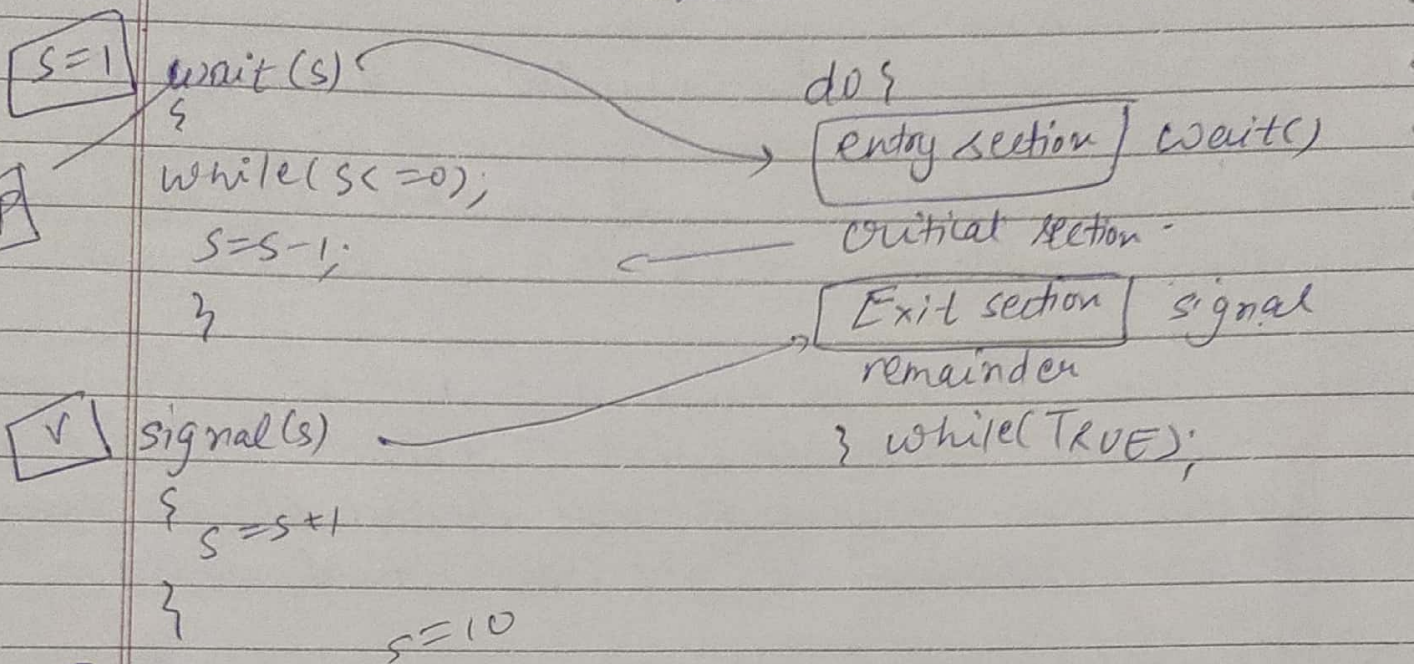
(4) No Assumption related to hardware Speed: There is no fixed parameters for executing any mechanism or algorithm to execute on a particular hardware config.

> Semaphor (It is used to prevent race condition):

To impose mutual exclusion Dijkstra proposed a very significant technique for managing concurrent processes by using the value of a simple integer variable to synchronise the progress of the interacting/cooperating processes. ∴ the integer variable is called semaphor, ∴ it is a

synchronising tool.

Operations on Semaphor



s=10

Q If we have two process 6P and 4V. Find value s?

Ans 6P = 10 - 6 = 4 4V = 4 + 4 = 8

Q If the value of s is 17 and system want to perform 5P, 3P and 1P. What'll be the current updated value of s?

Ans 8

Properties of Semaphore

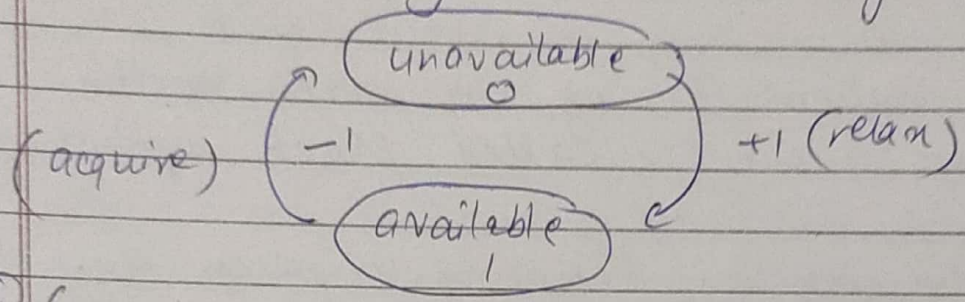
1. It is simple to implement.
2. Works with many processes.
3. It can have many different critical sections with different semaphores.
4. Each critical section has unique access semaphor values.
5. Can permit multiple processes into the critical section at once if desirable.

Scanned by CamScanner

Page .....
Date .....

Types of Semaphore

(1) **Binary**: It is a special form of semaphore used for implementing mutual exclusion. Binary semaphore is initialised to one and only takes the value of (0 or 1) during the execution of a program.



(2) **Counting Semaphore**: It takes more than two values. They can have any value that you want.

Applications of Semaphore

> To solve the critical section problem.
> For ordering of the events.
> For resource management.

Limitations of Semaphore

> With improper use, a process may block indefinately, such a situation is called Deadlock.

Busy Waiting

> It is a condition in which if one process is executing in its critical section and any other process wants to enter his critical section, then that process needs to check some condition in its entry section in continuous loop. (spin lock).

Scanned by CamScanner

Semaphor Queue

- It ensures no process go into busy waiting.
- The process waiting to execute in its critical section is moved to the semaphore queue till it get a chance to enter in its critical section without CPU allotment and this saves lot of CPU time.

(1) Block Operation: Moving the process to the semaphore queue, is called block operation.

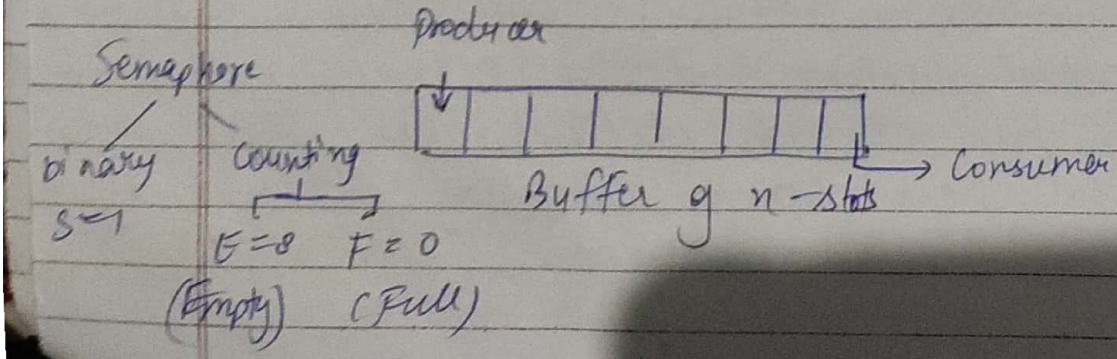
(2) Wake up operation: Removing a process from the semaphore queue and placing it in the ready queue is called wake up operation.

Note: Both the operations are performed by the OS as a basic system call.

Classical Problems of Synchronisation

(1) Bounded Buffer Problem:

Prblm Stmt: This problem is generalised in terms of Producer-Consumer Problem where a finite buffer pool is used to exchange message b/w producer and consumer processes. Because a buffer pool has a max size, therefore this problem is known as bounded buffer problem.



Scanned by CamScanner

→ A producer tries to put the data into an empty slot of the buffer.
→ A consumer tries to remove the data from a filled slot in the buffer.
→ These two processes wouldn't produce the expected o/p if they are being executed concurrently. There needs to be a way to make the producer and consumer in an independent manner.

Step1 : Creating 2 counting semaphores, one is full, another is empty — to keep track of the current no. of full and empty buffers respectively

Producer —

```
do
{  // wait until empty > 0
wait(E); // acquire lock
wait(S); // acquire lock
         // put the value
signal(S);      // release lock
signal(F);      // increment full
} while(True);
```

Consumer

```
do
{  // wait until F > 0
wait(F); // acquire lock
wait(S); // acquire lock
// remove the value
signal(S);   // release lock
signal(E);   // increment empty
} while(True);
```

Scanned by CamScanner

② Dining Philosopher's Problem

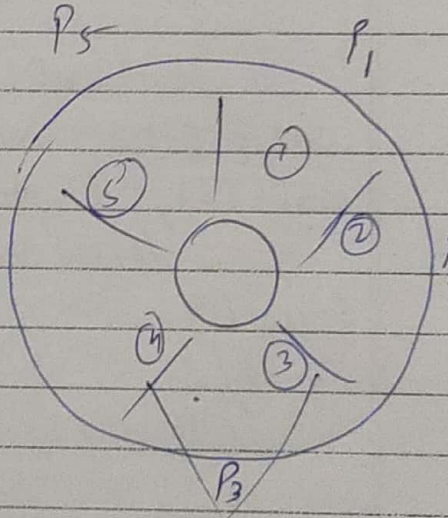
Page
Date

~~Dynamic~~ Mutex locks

> In this approach, in the ~~empty~~ entry section of code, a lock is acquired over the critical resources and used inside the critical section. After that the lock will be released when the proces exit from the critical section. As the resource is locked while a proces executes it's critical section hence, no other process can access it.

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

Prblm Stmt



Consider there are 5 philosophers sitting around a circular dining table. The dining table has 5 chopsticks and a bowl of food in the middle. At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he/she uses 2 chopsticks one from their left, one from their right. When a philosopher wants to think, he/she keeps down both the chopsticks at their original place. The prblm is no philosopher will starve. Each philosopher can forever continue to eat and think alternatively. It is assumed that no philosopher can know when others went to eat or think.

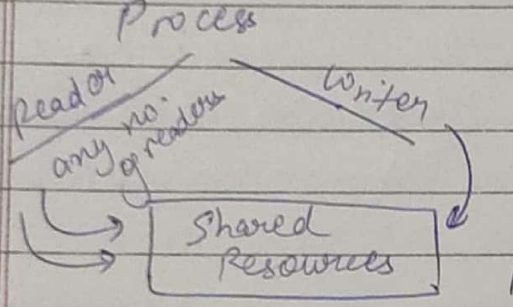
Scanned by CamScanner

```
do {
wait(stick[i]);
wait(stick[(i+1)%5]);
/* eat */
signal(stick[i]);
signal(stick[(i+1)%5]);
}
```

① If the philosopher P_1 & P_2 will start eating.
② Might be philosopher P_2 & P_3 will start eating.
③ From above two (mutex cases) ① & ② the philosopher P_5 will ever starve or P_5 will never get a chance to eat.
To solve the case ③ we are having another sol^n:

```
// ever starve                    i=4
while(True){
wait(T);
wait(stick[i]);
wait(stick[(i+1)%5]);
// CS (eating)
signal(stick[(i+1)%5]);
signal(stick[(i)]);
signal(T);
}
```

③ Reader/Writer Problem



Problem: There is a shared resource which should be accessed by multiple processes. When a writer is writing data to resource no other process can access the resource. A writer could not write to the resource if there are non-zero no. of readers accessing the same resource. At the time of writing no reading is allowed and at the time of reading, no writing is allowed.

Writer (wrt=1)

```
wait(wrt);
// critical section
signal(wrt);
```

Reader (updating the read counter in mutual exclusive manner; mutex=1)

```
wait(mutex);          // wrt=1, rc=0
rc++;                 // mutex=0
if(rc==1)             // rc=1
wait(wrt);            // wrt=0
signal(mutex);        // mutex=1
// Reading
wait(mutex);          // mutex=0
rc--;
if(rc==0)
signal(wrt);          // wrt=1
signal(mutex);        // mutex=1
}
```

This mutex is utilised for decreasing the read counter by 1 so that the system can identify who is the last reader.

Scanned by CamScanner

Two Process Sol"

Page .....
Date .....

Peterson's Algorithm

> The critical section problem sol" can be 2 process sol"
> It is designed to impose mutual exclusion in the execution of critical section for only 2 processes that's why it is known as 2 process sol".
> There is a 2 process sol" that has been given by Peterson.
> There is a simple algo that can be run by 2 processes to ensure mutual exclusion for one resource. It doesn't require any other hardware. It is restricted to two processes that alternate executions b/w their critical sections and remainder section.

P_0

```
do {
flag[P0] = T;
Turn = P1;
while (Turn == P1 && flag[P1] == T)
  {
  }
// CS
flag[P0] = F
remainder section
}
```

P_1

```
do {
flag[P1] = T;
Turn = P0;
while (Turn == P0 && flag[P1] == T)
      {
      }
```

Scanned by CamScanner

// CS

flag[P_i] = F_i

remainder section

}

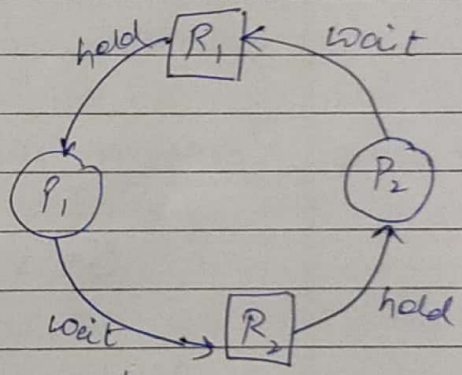
Dekker's Algo

Will discuss later

Deadlock

9/10/2019

> It is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource resulting in both programs ceasing to function.



How to avoid deadlock

> Deadlocks can be avoided by avoiding atleast one of 4 condi'ns because all these four conditions are required simultaneously to cause deadlock.

① Mutual Exclusion

> Resources shared such as read-only files do not lead to deadlock but resources such as printers require exclusive access by a single process. Then only 1 process can use at a time, no resource is going to share.

② Hold & Wait:

The processes must be prevented from holding one or more resources while simultaneously.

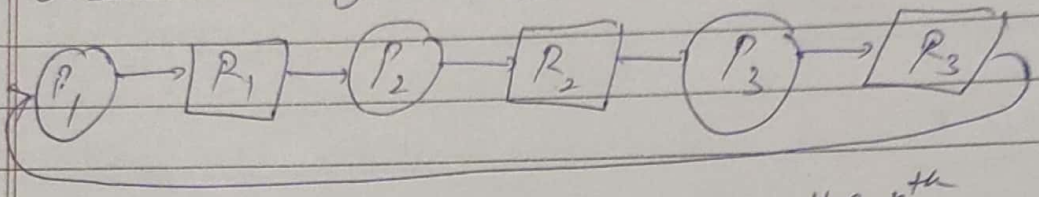
Scanned by CamScanner

waiting for one or more others.

(3) No Preemption

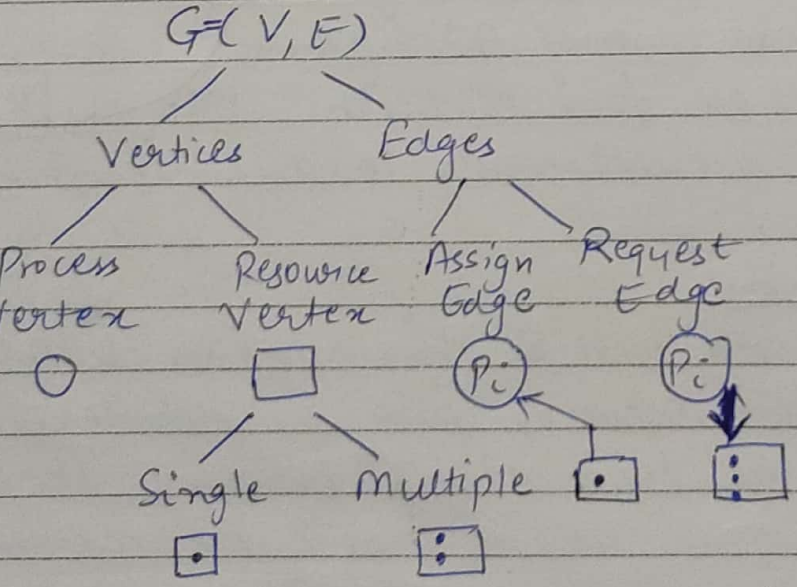
> Preemption of process resource allocation can avoid the condition of deadlock wherever possible.

(4) Circular ~~Weig~~ Wait

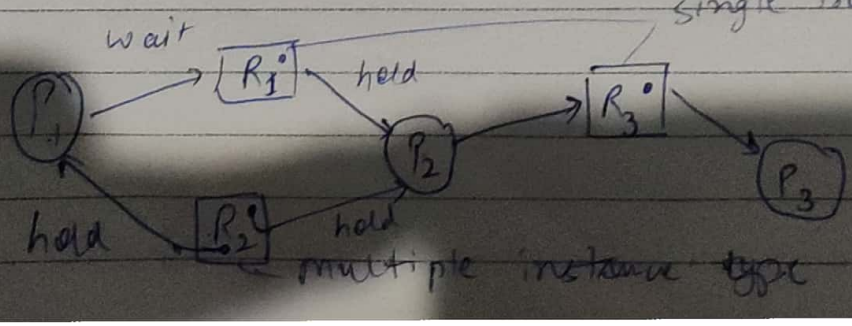


> Circular wait can be avoided if the nth ~~resource~~ be required by the Pn process so we have to break the cycle and we can follow the processes request resources in strictly increasing or decreasing order.

10/09/19 Resource Allocation Graphs

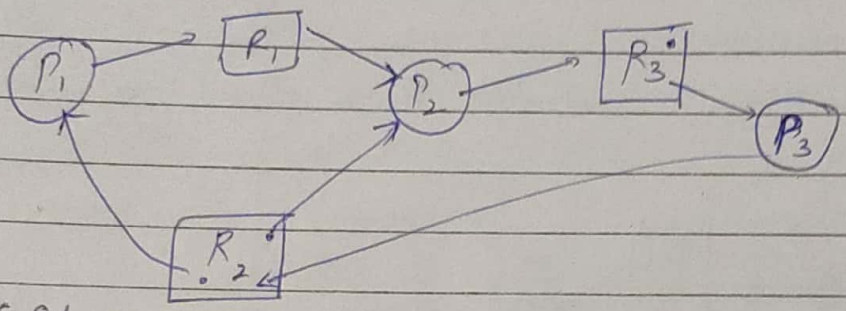


For example:



Scanned by CamScanner

Step 1: First we will try to finish the process P_3, then R_3 resource will be free.
Step 2: After completion of P_3, the resource R_1, R_2, R_3 will get free.
Step 3: After that P_1 can also be completed.
Then there's no deadlock.

Q. 

Whether there's deadlock or not?

Ans There is a deadlock.

Methods of handling Deadlock

Method (1) Deadlock Ignorance

Method (2) Deadlock Prevention – Try to dissatisfy one or more of the following necessary conditions.

→ Mutual exclusion - make some of the resources shareable for e.g. file in read-only manner.

→ Hold and Wait - When a process acquires a resource, it doesn't hold any other resource.

→ No Preemption - We will try to preempt some of the processes from a particular resource if possible

→ Circular Wait → Once a process has some resources allocated to it, it can allocate a new resource only if no. of all its allocated resources is less than no. of assigned to the requested resource.

Refer to previous page

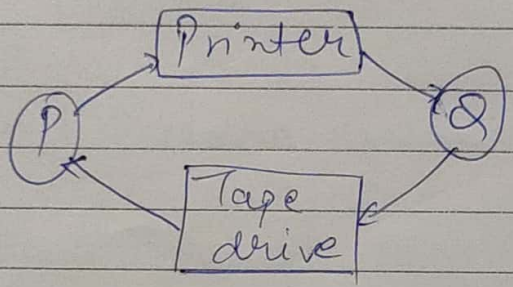
Scanned by CamScanner

Date .....

Method ③ Deadlock Avoidance:

For avoiding deadlock an addi^n al info is required about how resources are to be executed.

for e.g. In a computer system with one tape drive, one printer, the system might need to know, the process P ~~need to~~ will request first the tape drive, then the printer. Before releasing both the resources whereas process Q will request first the printer then the tape drive.



15/10/19 Safe state: A state is safe if a system can allocate resources to each process upto the max. in some order to avoid a deadlock condition.

> A system is in safe state only if there exists a safe sequence.

> A sequence of processes $P_1, P_2, P_3 \cdots P_i$ is a safe state if for each P_i the resource request that P_i can still be satisfied by the currently available resources plus the resources held by all processes

Banker's Algorithm

MAX → How much of each resource each process could possibly request.

Scanned by CamScanner

ALLOCATED: How much of each resource each process is currently holding.

AVAILABLE: How much of each resource, the system currently has available.

NEED: How much the remaining need of each process may need more instances of resources.

Q1)

Processes	Allocation (A B C)	Max (A B C)	Available (A B C)	Need (A B C) (Max − Allocation)
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2	2 0 0	1 2 2
P_2	3 0 2	9 0 2	5 3 2	6 0 0
P_3	2 1 1	2 2 2	7 4 3	0 1 1
P_4	0 0 2	4 3 3	7 5 3	4 3 1
			3 0 2	
			10 5 5	

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

Check if Available is greater than need if yes then put that process in safe sequence, proceed further if a process's need is satisfied by available then put it in sequence and after adding that process's allocation to available and proceed as same

Q2)

Processes	Allocation	Max	Available	Need
P_0	1 0 1	4 3 1	3 3 0	3 3 0
P_1	1 1 2	2 1 4	1 0 1	1 0 2
P_2	1 0 3	1 3 3	4 3 1	0 3 0
P_3	2 0 0	5 4 1	1 1 2	3 4 1
			5 4 3	
			1 0 3	
			6 4 6	

Safe Sequence $> P_0, P_1, P_2, P_3$

Scanned by CamScanner

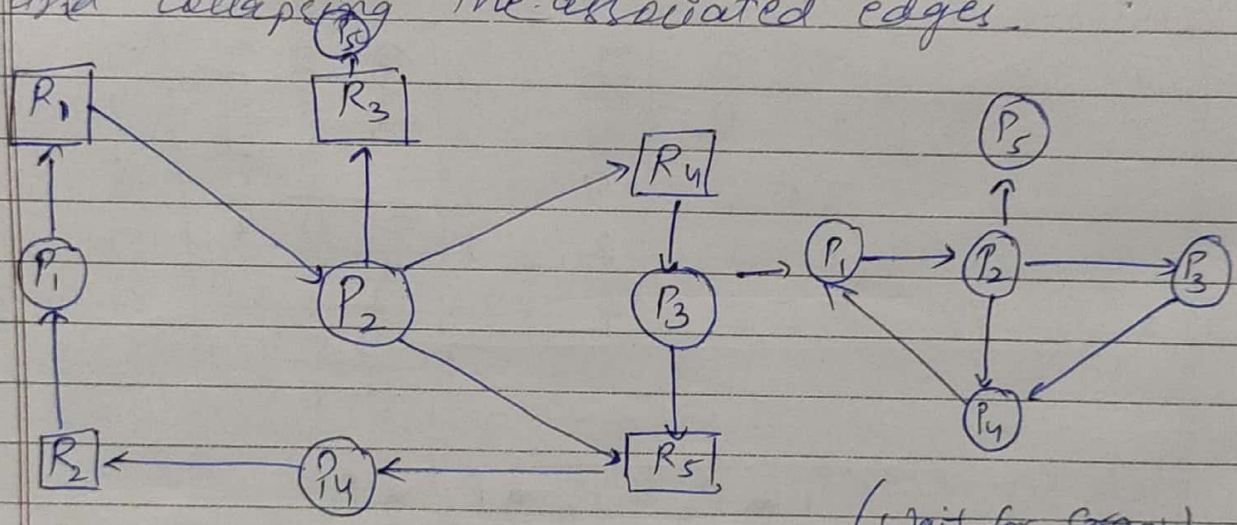
Date 1.7.10.19

Deadlock Detection and Recovery

> If deadlock prevention and avoidance are not done properly, as deadlock may occur and the last thing is just to detect the deadlock and recover from it.

> In each resource category there is a single instance, then we can use a variation of resource allocation graph known as a wait for graph.

> A wait for graph can be constructed from a resource allocation graph by eliminating the no. of resources and collapsing the associated edges.



(Resource allocation graph) (Wait for Graph)

Cycles in the wait for graph indicate deadlock.

> This algorithm must maintain the wait for graph and periodically search for closed loops.

Recovery of Deadlock → another possibility is to system recover from the deadlock automatically

(1) Process Termination : Terminate all the processes involved in the deadlock. Terminate process one by one until the deadlock is broken.

Scanned by CamScanner

Page .....
Date .....

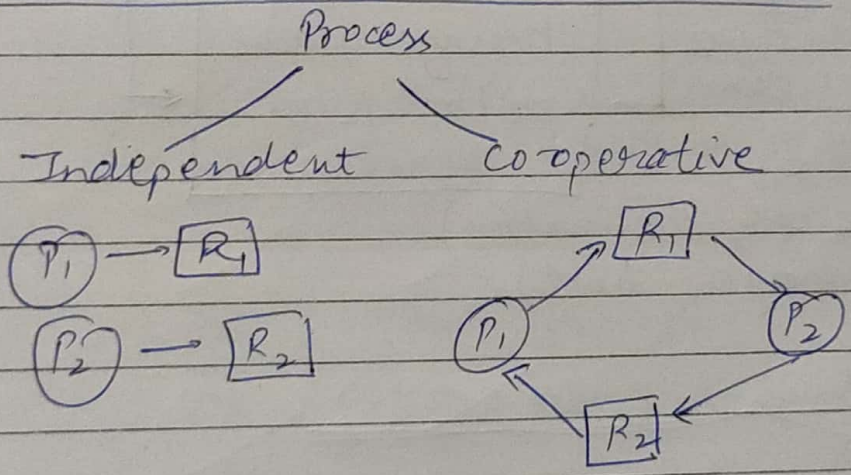
(2) Resource Preemption → When preempting resources to relieve deadlock.

2a) Selecting the victim: deciding which resource to preempt from which processes.

2b) Roll back: ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process.

2c) Starvation: A process couldn't starve because its resources are constantly being preempted.

Inter Process Communication (IPC)



IPC → IPC is a mechanism which allows processes to communicate each other and synchronise their actions.

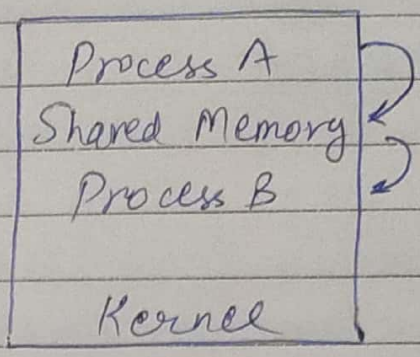
> The communication b/w these processes can be designed as a method of cooperation b/w them.

> Processes can communicate with each other using two methods: one is shared memory and other is message passing.

Scanned by CamScanner

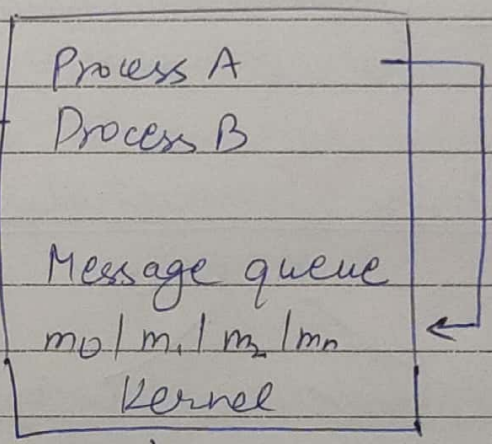
① Shared memory

e.g. Producer-Consumer problem.
Refer to previous



② Message passing

> In the message passing mechanism there are 2 types of primitive types of primitive can be used for exchanging no. of msg:



(a) Send(message, destination)
(b) Recieve(message, host)

Deadlock Prevention (Hold & wait)

> Conservative approach: Process is allowed to start execution if and only if it has acquired all the resources (less efficient, non-implementable, easy, deadlock independent)

> Do not hold: Process will acquire only desired resources but before making any fresh request it must release all the resources that it currently holds (efficient, implementable)

> Wait Time Outs: We place a max. time upto which a process can wait after which process must release all the holding resources.

Scanned by CamScanner

Sunil
Page
Date

No Preemption

- Forceful preemption: We allow a process to forcefully preempt the resource holding by other processes
- The method may be used by high priority process or system process.
- The process which are in waiting state must be selected as a victim instead of process in the running state.

Circular Wait

- Circular wait can be eliminated by just giving a natural no. of every resource. $f: N \to R$
- Allow every process to either only in the increasing or decreasing order of the resource no.
- If a process require a less no. (in case of increasing order) then it must first resolve all the resources larger than required no.

Q

Process	Allocation	MAX	Available	Need
P_0	0 0 3	8 4 3	3 2 2	8 4 0 8 4 2
P_1	3 2 0	6 2 0	1 2 2	3 0 0 1 0 0
	5 2 0		5 2 0	
P_2	2 1 1	3 3 3	6 4 2	1 2 2 1 2 2
			2 1 1	
			8 5 3	

(a) REQ1 $P_0 \to 002$
REQ2 $P_1 \to 200$

	Step 1	Step 2
	3 2 2	3 2 2
	− 0 0 2	2 0 0
		+ 1 2 2
Available	3 2 0	

From resource-request algorithm, the new allocated resources for P_0 are $001 + 002 \to 003$

Calc the safe sequence

Ans $\langle P_1, P_2, P_0 \rangle$

Scanned by CamScanner

Q

Process	Allocation	MAx	Available	Need
P_0	1 0 2		2 2 0	4 1 2
P_1	0 3 1			1 5 1
P_2	1 0 2			1 2 3

x y z

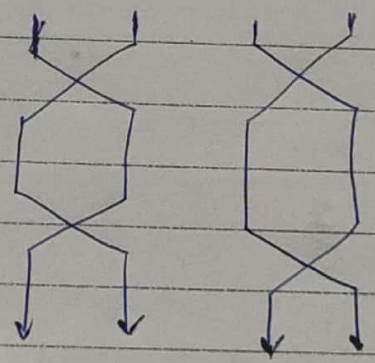
P_0 : 0 1 0

Q

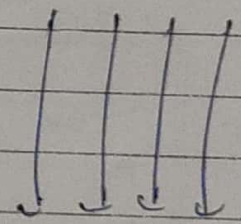
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

P_1 : 1 0 2

Concurrent Processes



(Concurrency)



parallelism

? For representing concurrent processes we can utilise 2 types of presentations, one is

(1) parallel begin or parbegin and parallel end or parend

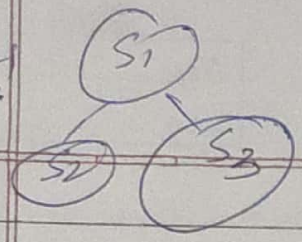
(2) fork and join construct.

Type 1 Parallel begin & parallel end Representation

Scanned by CamScanner

Sunil
Page .....
Date .....

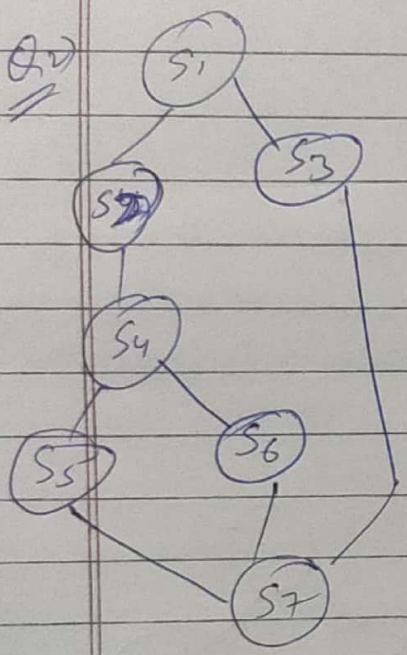
Q1 =>



$S_1 \Rightarrow x = a + b$
$S_2 \Rightarrow y = x + 2$
$S_3 \Rightarrow w = x + a$

```
begin
S1;
parbegin
S2;
S3;
parend
end
```

Q2 =>




```
begin
S1;
parbegin
S3;
begin
S2;
S4;
parbegin
S5;
S6;
parend
end
parend
S7;
end
```

Concurrent Process

Type II Using fork and Join construct

→ Fork is called by a thread (parent). To create a new thread (child process) & concurrency.

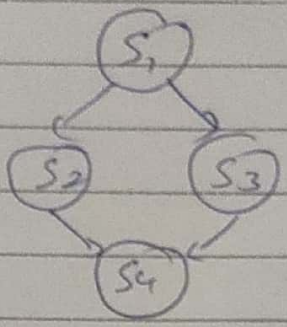
→ Join recombine 2 concurrent combination into one.

→ It is called by both parent and child process

Scanned by CamScanner

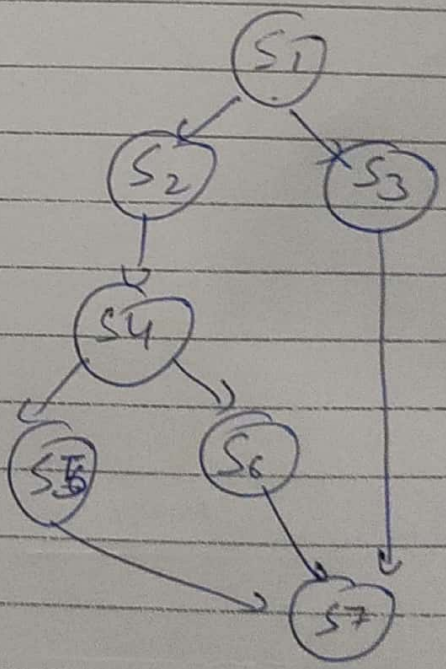
→ Fork increases the concurrency, join decreases the concurrency

```
count = 2
S1;
fork L1;
S2;
goto L2;
L1 : S3;
  L2 : Join count;
S4;
```



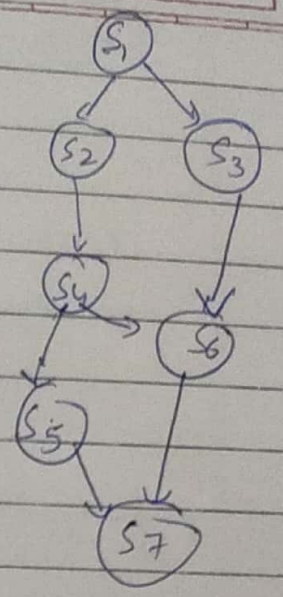
L1 → Virtual memory
Join → reduce ^n of concurrency
Fork → increase

```
count = 3;
  S1;
  Fork L1;
  S2 ;
  S4;
  Fork L2;
  S5;
  goto L3;
  L2 : S6;
goto L3;
  L1 : S3;
    L3 : Join count;
S7;
```



Scanned by CamScanner

Count1 = 2
S1;
fork L1;
S2;
S4;
Count2 = 2;
fork L3;
S5;
goto L2;
L1: S3;
L2: Join Count1
L3: S6;
L4: Join Count2
S7; goto L4;



Producer - Consumer → Bounded Buffer
Deadlock avoidance
4 necessary condi^n s of deadlock

deadlock v/s Starvation
Parallelism v/s Concurrency

Semaphore Queue

```
wait(semaphore *s)
{
   s->value--;
   if(s -> value < 0)
     {
       add this process to s -> list
     block();
     }
}
```

Scanned by CamScanner

```
signal (semaphore *s)
  s -> value ++;
if (s-> value <= 0)
  {
  remove a process P from s -> list;
  wakeup();
  }
}
```

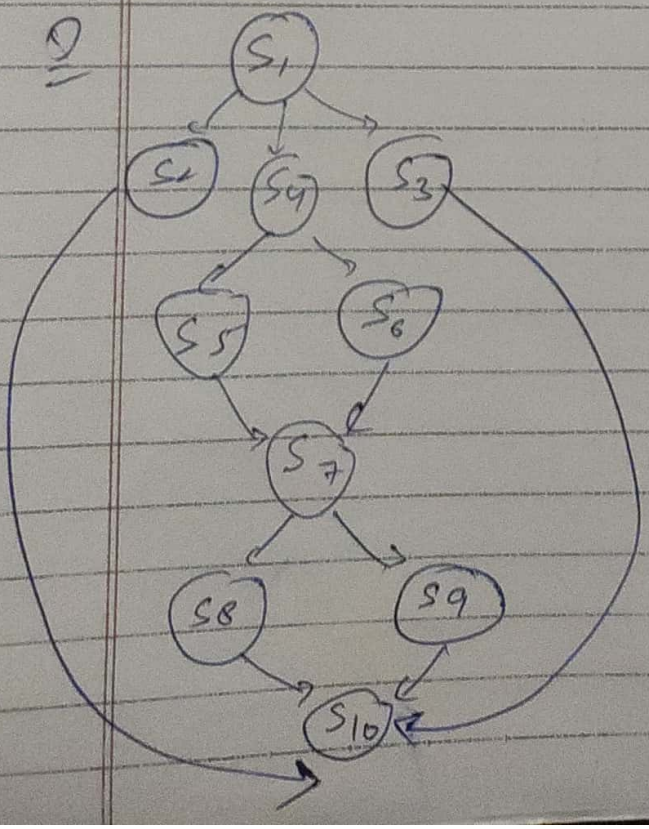
// refer to previous lecture

Concurrency conditions

* When 2 stmts in a program be executed concurrently then ~~they~~, ~~execute~~ still they produce the same result. The following 3 condition must be applied → Bernstein's conditions.

$R(S_1) \cap W(S_2) = \{\}$
$W(S_1) \cap R(S_2) = \{\}$
$W(S_1) \cap W(S_2) = \{\}$

2.

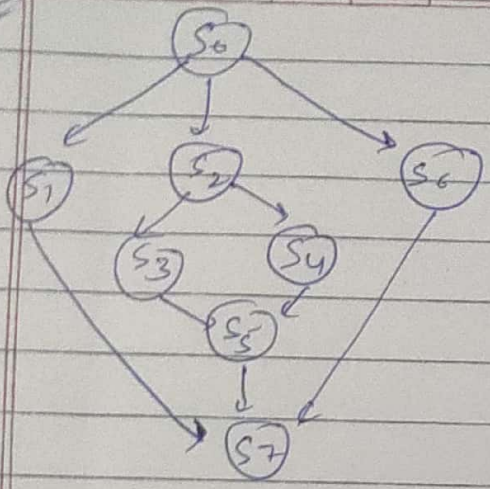



```
begin
S1;
parbegin
  S2;
  S3;
  begin
    S4;
    parbegin
      S5;
      S6;
    parend
    S7;
    parbegin
      S8;
      S9;
```

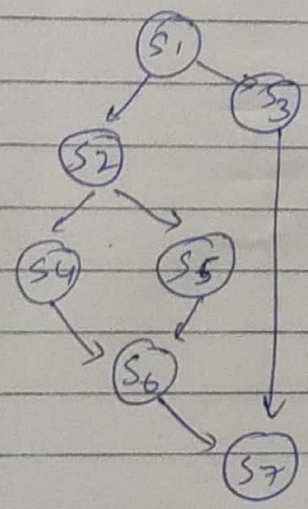
```
    parend
  end
parend
S10;
end
```

Scanned by CamScanner

Q2)




```
begin
S0;
parbegin
  S1;
  begin
    S2;
    parbegin
      S3;
      S4;
    parend
    S5;
  end
  S6;
parend
S7
end
```



Q3)

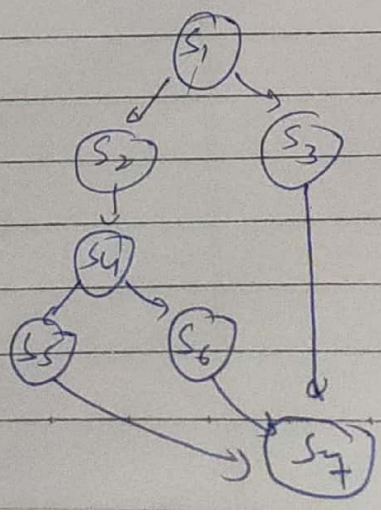
```
Begin
  S1;
  fork L1
  S2;
  L3: Join
  S6;
  L1: Join
  S7
  end;
```

```
fork L2
S4
goto L3
S6
count: 2
L4: Join
S7
end
```

Q4)

```
S1
count1 = 2
fork L1
S2;
S4;
count 2 = 2
fork L3
S5;
goto L4
L1: S3
L2: Join count 2
```

```
L3 : S6
L4 : Join count 1
     goto L2
S7;
```



Scanned by CamScanner