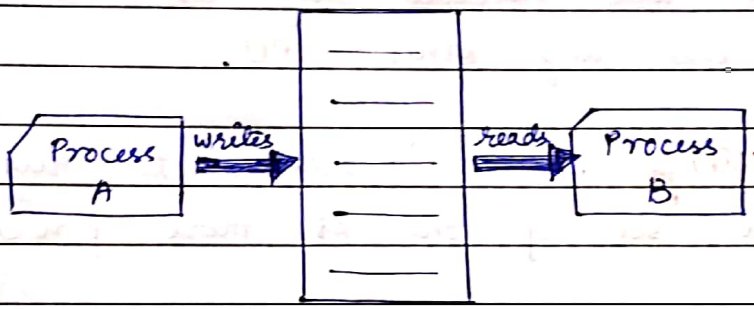
MODULE - II

Date / /
Page No.
Shivalal

⇒ PROCESS SYNCHRONIZATION

Process Synchronization means sharing system resources by processes in such a way that concurrent access to shared data is handled by minimizing the chance of inconsistent data.

Several processes run in an OS. Some of them share resources due to which problems like data inconsistency may arise.

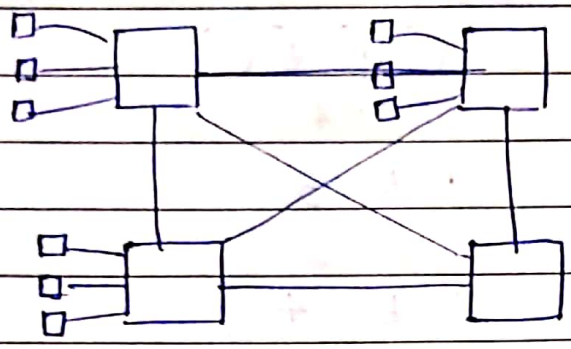


One process changes the data in a memory location whereas another process is trying to read the data from the same memory location. It is possible that the data read by the second process might be incorrect or erroneous. So, we have to follow synchronization, i.e., one process will finish writing then only another process will start reading.

Process Management in OS can be divided into 3 parts:

1. Multiprogramming (Multitasking)
2. Multiprocessing
3. Distributive Processing

⇒ DISTRIBUTIVE PROCESSING



It involves multiple processes on multiple systems. All of these involve co-operation, competition & communication between processes that either run simultaneously.

⇒ CONCURRENCY

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

⇒ DIFFICULTIES & ISSUES IN CONCURRENCY

1. Sharing global resources safely is difficult.
2. Optimal allocation of resources is difficult.
3. Locating programming errors can be difficult.

Ques -

```
P1()                    P2()
{                       {
(1) C = B - 1;          (3) D = 2 * B;
(2) B = 2 * C;          (4) B = D - 1;
}                       }
```

B is a shared variable and its initial value is 2.

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

③ ① ④ ②

D = 4

C = 1

B = 3

B = 2

⇒ RACE CONDITION

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

Ques - Calculate (for the above ques) how many values the shared variable B can have.

1234, 3412, 1324, 1342, 3124, 3142.

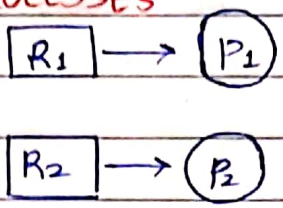
The value of B can be 2, 3 or 4.

⇒ CRITICAL SECTION PROBLEM

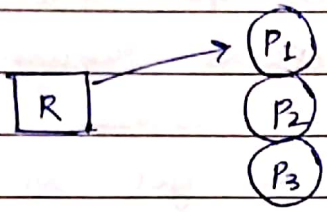
A critical section is a core segment that access shared variables & has to be executed as an atomic action. It means that in a group of co-operating processes at a given point of time, only one process

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

⇒ INDEPENDENT PROCESSES



⇒ COOPERATING PROCESSES



Processes can share variables, memory, core segment, resources (CPU, printer, scanner)

eg:
```
~~do~~ 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 segment comes in this.
```
}
```

⇒ SYNCHRONIZATION MECHANISM

Here is a solution to the critical section problem that must satisfy following requirements.

(1) MUTUAL EXCLUSION (MUTEX):

Out of a group of co-operating processes, only one process can be included in its critical section at a given point of time. It is a core segment that prevents simultaneous access to a shared resource.

Mutex & Progress → Primary conditions

Date / /
Page No.
Shivalal

(2.) PROGRESS :

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

(3.) BOUNDED WAITING :

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

(4.) NO ASSUMPTION RELATED TO HARDWARE SPEED :

There is no fixed parameter for execution of any mechanism or algorithm to execute on a particular hardware configuration.

Semaphore : 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 simple integer variable to synchronize the progress of the interacting or cooperative processes. The integer variable is called semaphore. Therefore, it is a synchronizing tool.

→ OPERATIONS IN SEMAPHORE

```
do
{
entry section    wait()    P
critical section
exit section     signal()  V
remainder section
} while (TRUE);
```

eg:

```
wait (s)                    signal(s)
{                           {
  while (S<=0);               s = s+1;
    s = s-1;                }
}
```

wait() will work as 'down' operator and signal() will work as 'up' operator.

Ques- S = 10, 6P, 4V then what will be the updated value of S?

P ⇒ 10 − 6 = 4

V ⇒ 4 + 4 = 8

Value of S = 8

This is counting semaphore.

Ques- S = 17, 5P, 3V and 1P. What will be the current updated value?

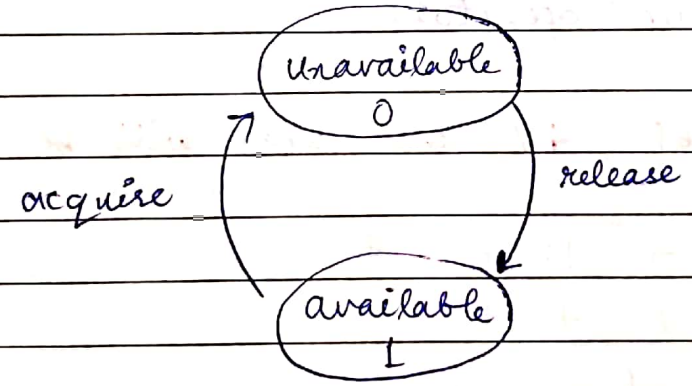
S = 17 − 5 + 3 − 1 = 14

⇒ PROPERTIES OF SEMAPHORE

- It is simpler to implement.
- It works with many processes.
- It can have many different critical sections with different semaphores. Each critical section has unique accessed semaphore.
- It can permit multiple processes into the critical section at once if desirable.

⇒ TYPES OF SEMAPHORE

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



(2.) **Counting**: It can take more than 2 values. They can have any value you want.

⇒ APPLICATIONS OF SEMAPHORE

1. To solve the critical section problems.
2. For ordering of the events.
3. For resource management

→ LIMITATIONS OF SEMAPHORE

1. With improper use, a process may block indefinitely. Such a situation is called deadlock.
2. **Busy Waiting :** It is a condition in which if one process is executing in its critical section & any other process wants to enter its critical section then that process needs to check some condition in its entry section in continuous loop. (Spin Lock)

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

(i.) **Block Operation :-** Moving the process to the semaphore queue is called block operation.

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

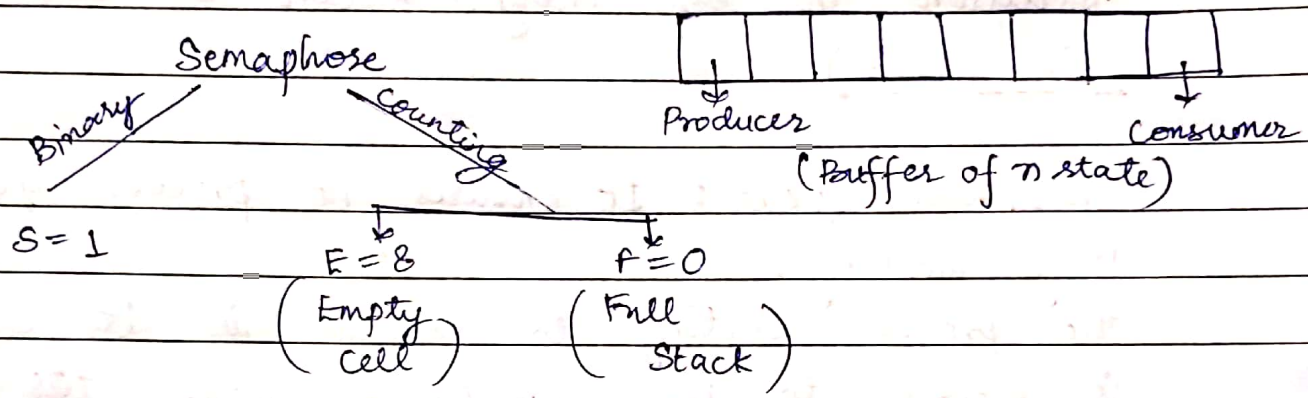
Note: Both the operations are performed by the OS as the basic system call.

→ CLASSICAL PROBLEMS OF SYNCHRONIZATION

(1.) **BOUNDED BUFFER PROBLEM :**
Producer Consumer Problem

→ Problem Statement :

This problem is generalised in terms of producer-consumer problem where a finite buffer pool is used to exchange messages b/w producer & consumer processes, because the buffer pool has maximum size. Therefore, this problem is known as bounded buffer problem.



A producer tries to put the data into an empty slot of the buffer. A consumer tries to remove the data from the filled slot in the buffer.

These two processes will not produce the expected O/P if they are being executed concurrently. ~~Their~~ There needs to be a way to make the producer & consumer in an independent manner.

→ Solution : Creating 2 counting semaphore, one is full another one is empty to keep the track of the current no. of full & empty buffers respectively.

Producer:

```
do {                        // wait until Empty >0
    wait (E);               // acquire lock
    wait (S);               // put or write the value & acquire lock
    . . . . .
    signal (S);             // release the lock
    Signal (F);             // increment the full
} while (True);
```

Consumer:

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

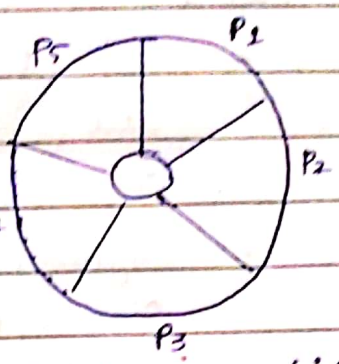
⇒ MUTEX LOCKS

In this approach, in the entry section of code, a lock is acquired over the critical resources & used inside the critical section after that the lock will be released when the process exit from the critical section. As the resource is locked, while a process execute its critical section, hence, no other process can exit it.

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

(2.) DINING PHILOSOPHER'S PROBLEM :

Problem Statement :



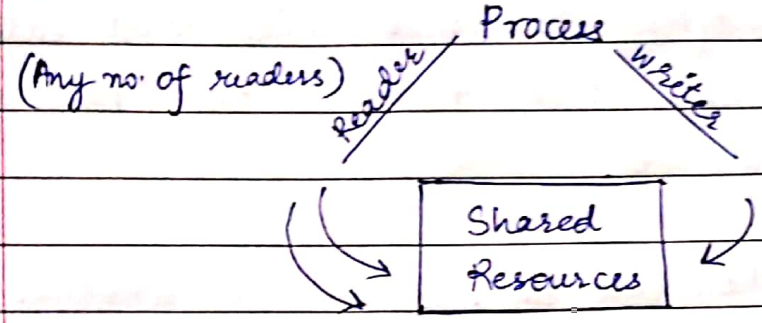
Consider there are 5 philosophers sitting around the circular dining table. The dining table has 5 chopsticks and a bowl of food in the middle. At any instant, a philosopher is either eating or thinking. When a philosopher wants to eat, he/she uses 2 chopsticks, one from their left & one from their right. When a philosopher wants to think, he/she keeps down both the chopsticks at their original place. The problem is no philosopher will starve.

Each philosopher can forever continue to think & eat alternatively. It is assumed that no philosopher can know when other wants to eat or think.

Solution :

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

3.) READER WRITER 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 can't write to the resource if there are non-zero number 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

```
wait (wrt);
critical section   (Writing is started)
signal (wrt);
```

reader

```
wait (mutex);
rc++;
if (rc == 1)
{wait (wrt);}
signal (mutex);
// reading ⇒ Critical Section
wait (mutex);
rc--;
if(rc == 0)
signal (wrt);
signal (mutex);
```

mutex = 1
wrt = 1
rc = 0 (read counter)

- mutex is being used for updating the read counter in each section again & again, in mutual exclusion manner.
- This mutex is being used to decrease the rc so that system can identify the last reader.

→ DINING PHILOSOPHERS ANOTHER SOLUTION :

(I) If the philosopher P_1 & P_3 will start eating
(II) Might be philosopher P_2 & P_4 will start eating
(III) From above 2 cases : (I) & (II), the philosopher P_5 will ever starve.

To solve solve the case III, we have another solution

```
while (True) {                                  T=4
    wait (T);
    wait (stick[i]);
    wait (stick [(i+1)%5 ]);
  // Critical section ⇒ eating
    signal (stick [(i+1)%5 ]);
    signal(T);
}
```

The critical section problem solution can be two-process solution.

⇒ PETERSON'S ALGORITHM

It is designed to impose mutual exclusion in the execution of critical section for only two processes that's why it is known as two process solution.
There is a two process solution that has been given by Peterson.
There is a simple algorithm that can be run by two processes to ensure mutual exclusion for one resource. It doesn't require any special hardware. It is restricted to two processes that alternate execution b/w their critical section &

remainder section & vice-versa.

P_0	P_1

do
{
flag[P_0] = T;
Turn = P_1 ;
while (Turn == P_1 && flag[P_1] == T);
{
}
// CS
flag[P_0] = F;
remainder section
}
	do
	{
	flag[P_1] = T;
	Turn = P_0;
	while(Turn == P_0 && flag[P_0] == T);
	{
	}
	// CS
	flag[P_1] = F;
	remainder section
	}

→ GENERALIZED ALGORITHM:

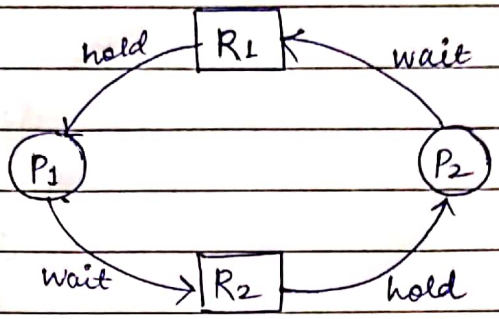
{ P_i P_j [1 − i]

flag [i] = T;
Turn = j;
while (flag [j] == T && Turn == j);
// CS
flag [i] = F;
remainder section
}

NOTE: In while condition, if we use OR operator then it will break the mutual exclusion.

Dekker's Algorithm
Code for busy waiting problem soln } H.W.

⇒ DEADLOCK



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

Imp:

→ HOW TO AVOID DEADLOCK :

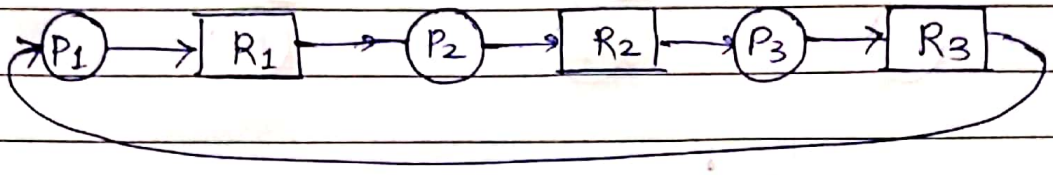
Deadlocks can be avoided by avoiding at least one of four conditions because all these 4 conditions are required simultaneously to cause deadlock.

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 one 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 waiting for one or more others.

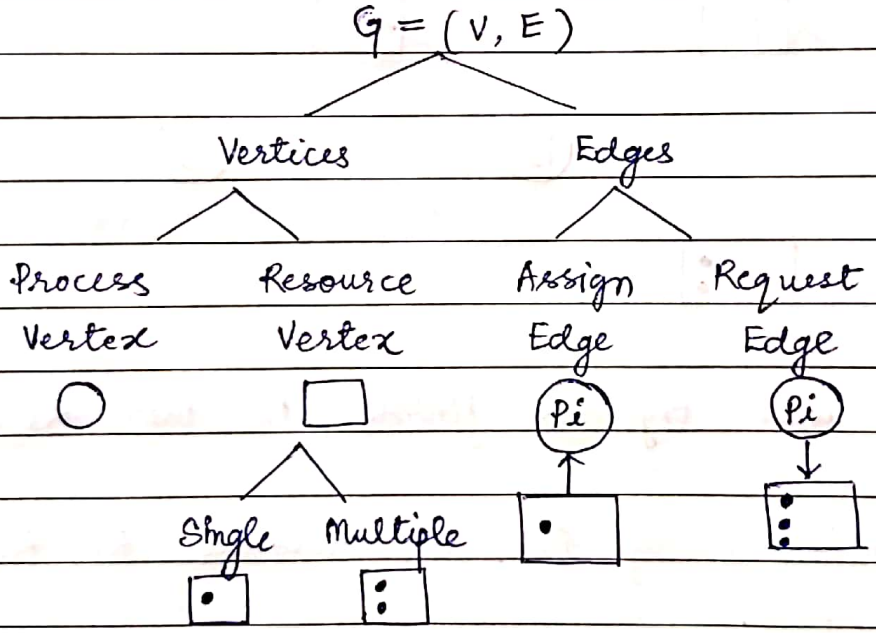
3.) No Preemption : Preemption of process resource allocation can avoid the condition of deadlock wherever possible. (Round Robin preferred)

4.) Circular wait :



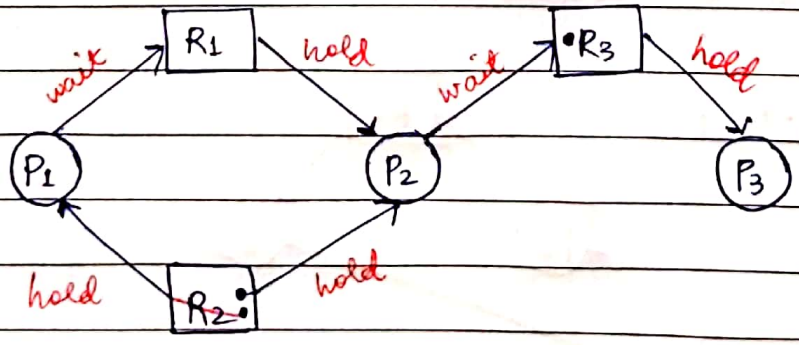
Circular wait can be avoided if ~~all~~ nth resources be acquired by the Po process so we have to break the cycle & we can follow the processes request resources in strictly increasing or decreasing order.

⇒ RESOURCE ALLOCATION GRAPH

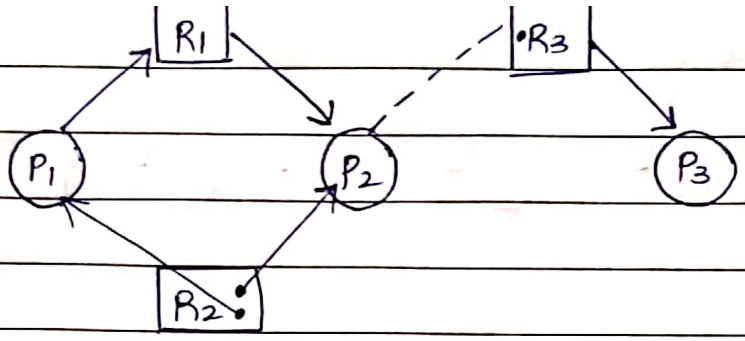


– Graph Presentation –

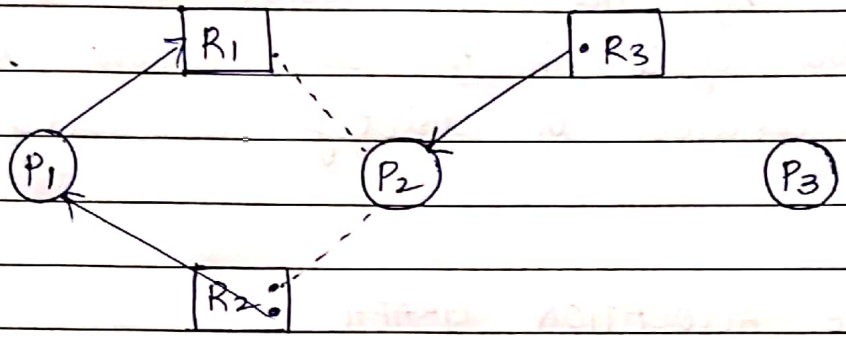
Imp: E.g:



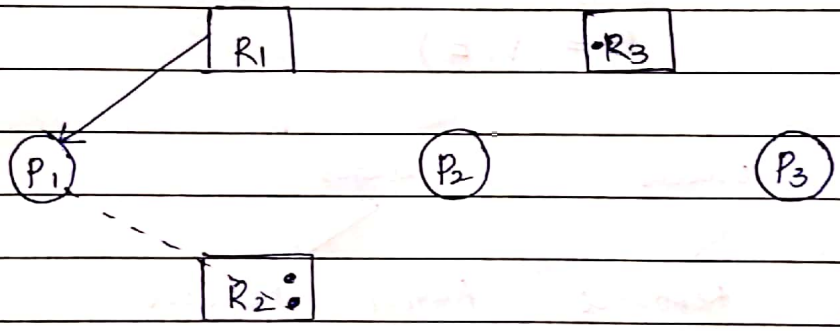
(I)



(II)



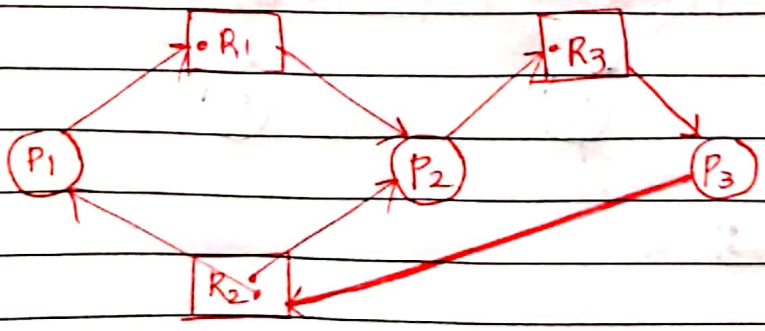
(III)



- First we will try to finish P_3 then R_3 will be free.
- After completion of P_2, the resource R_1, R_2, R_3 will get free.
- After that, P_1 will also get completed.

Then there is no deadlock.

Ques -



Is there any deadlock? Yes because hold & wait cond^n.

⇒ METHODS OF HANDLING DEADLOCK

1. **DEADLOCK IGNORANCE :** Restarting the machine if any problem occurs.

2. **DEADLOCK PREVENTION :** Try to dissatisfy one or more of the following necessary conditions.

- Mutual Exclusion :- Make some of the resources shareable. e.g : file in read-only manner.
- Hold & Wait :- When a process requires 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.

3. **DEADLOCK AVOIDANCE :** For avoiding deadlock, an additional information is required about how resources are to be requested.

e.g: In a computer system with one tape drive, one printer; the system might need to know the process P will request first the tape drive & then the printer before releasing both the resources whereas process Q will request first the printer and then the tape drive.

3. **DEADLOCK AVOIDANCE :** For avoiding deadlock, an additional information is required about how resources are to be requested. (Safe State)

e.g: In a computer system with one tape drive, one printer; the system might need to know the process P will request first the tape drive & then the printer before releasing both the resources whereas process Q will request first the printer and then the tape drive.

→ **Safe State:** A state is safe if the system can allocate resources to each process upto the maximum 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 $<P_1, P_2, \ldots P_i>$ is a

safe sequence for the current allocation state if for each P_i, the resource request that P_i can still make be satisfied by the currently available resources + the resources held by all.

→ Banker's Algorithm

- MAX – how much of each resource each process could possibly request.
- 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. Process may need more instances of resources.

NEED = MAX – ALLOCATION

Ques –

Process	Allocation A B C	Max A B C	Available A B C	Need A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2	5 3 2	1 2 2
P_2	3 0 2	9 0 2	7 4 3	6 0 0
P_3	2 1 1	2 2 2	7 4 5	0 1 1
P_4	0 0 2	4 3 3	7 5 5	4 3 1
			10, 5, 7	

(Available: 3 3 2 + 2 0 0 = 5 3 2; + 2 1 1 = 7 4 3; + 0 0 2 = 7 4 5; + 0 1 0 = 7 5 5; + 3 0 2 = 10, 5, 7)

⇒ Safe Sequence

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

Safe sequence can be more than one.

Ques –

Process	Allocation E F G	Max E F G	Available E F G	Need E F G
P_0	1 0 1	4 3 1	3 3 0	3 3 0
P_1	1 1 2	2 1 4	4 3 1	1 0 2
P_2	1 0 3	1 3 3	5 4 3	0 3 0
P_3	2 0 0	5 4 1	6 4 6	3 4 1
			8, 4, 6	

(Available: 3 3 0 + 1 0 1 = 4 3 1; + 1 1 2 = 5 4 3; + 1 0 3 = 6 4 6; + 2 0 0 = 8, 4, 6)

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

Ques

Processes	Allocation			MAX			Available			NEED		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
P_0	0	0	1	8	4	3	3	2	2	8	4	2
P_1	3	2	0	6	2	0				3	0	0
P_2	2	1	1	3	3	3				1	2	2

Considers the following independent requests for additional resources in the current on going state:

		X	Y	Z
REQ1 :	P_0	0	0	2
REQ2 :	P_1	2	0	0

Can these be granted safely or not?

(a.) REQ 1
(b.) REQ 2
(c.) Both (a.) & (b.)
(d.) None of these

From resource request algorithm, the new allocated resources for P_0 are : 001 + 002 = 003

Soln: When REQ 1 is assigned : Available

322 − 002 = 320

320 + 320 = 640

640 + 211 = 851

322 − 003 = 319

319 + 320 = 639

639 + 211 = 850

Safe Sequence:

~~$< P_1, P_2, P_0 >$~~

~~$< P_1 >$~~

$< P_1, P_2, P_0 >$

When REQ 2 is assigned : Available

322 − 200 = 122

122 + 211 = 333

333 + 320 = 653

322 + 320 = 642

642 + 211 = 853

∴ REQ1 & REQ2 can't be assigned.

Ques-

Processes	~~Allocation~~ Need			Allocation ~~MAX~~			Available			~~Need~~		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
P_0	4	1	2	1	0	2	2	2	0			
P_1	1	5	1	0	3	1						
P_2	1	2	3	1	0	2						

REQ1:

	X	Y	Z
P_0	0	1	0

Req1 is assigned :

$$\begin{array}{r} 220 \\ -010 \\ \hline 210 \end{array}$$

Safe seq : < >

System is in deadlock so req1 can't be granted.

Ques-

Processes	Allocation				MAX				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0	0	0	0	0
P_1	1	0	0	0	1	7	5	0					0	7	5	0
P_2	1	3	5	4	2	3	5	6					1	0	0	2
P_3	0	6	3	2	0	6	5	2								
P_4	0	0	1	4	0	6	5	6								

REQ1 :

	A	B	C	D
P_1	0	3	2	0

Imp ?

Ques- Consider a system with 3 processes that share 4 instances of the same resource type. Each process can request a maximum of 'k' instances. Resource instance can be requested and only one at a time. The largest value of k that will always avoid deadlock is ?

(a.) 2
(b) 3
(c) 4
(d.) 5

$P = 3, \quad R = 4$

$\boxed{R \geq P(N-1) + 1}$

$4 \geq 3(k-1) + 1$

$3 \geq 3(k-1)$

$1 \geq k - 1$

$k \leq 2$

Ques - $P = 5, \quad R = 4$

$4 \geq 5(N-1) + 1$

$3 \geq 5(N-1)$

$N - 1 \leq 3/5$

$N \leq \frac{3}{5} + 1 \Rightarrow N \leq \frac{8}{5} \Rightarrow N \leq 1.6$

Max value : 1

⇒ BANKER'S ALGORITHM STEPS

(1.) RESOURCE REQUEST ALGORITHM

Request$_i$ → Request vector for process P_i

Step 1: If Request$_i \leq$ Need$_i$ then go to the Step 2 otherwise error.

Step 2: $Request_i \leq Available \Rightarrow$ go to Step 3.
Otherwise Pi will be in waiting state

Step 3: $Available = Available - Request_i$

$Allocation = Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

If the resulting resource allocation state is safe.
If there is any unsafe state for process Pi then Pi must wait for $Request_i$.

(2.) SAFETY ALGORITHM

An algorithm for finding out whether a system is in safe state or not.

Step 1: $Work = Available$
$Finish[i] = False$ for $i = 0, 1, \ldots n-1$

Step 2: $Need_i \leq Work$ then follow Step 3, 4:

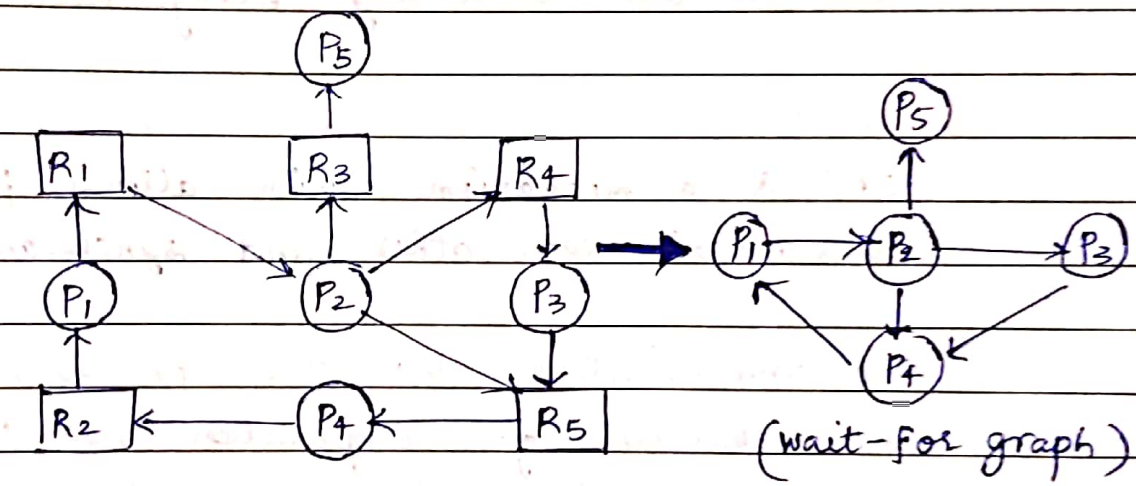
Step 3: $Work = Work + Allocation$

Step 4: if $Finish[i] = True$ for all i then system is in safe state.

4. DEADLOCK DETECTION & RECOVERY :

If deadlock prevention & avoidance are not done properly, as deadlock may occur & the last thing is just to detect the deadlock & recover from it. In each resource category there is a single instance when we can use a variation of the resource allocation graph known as a "wait-for-graph".

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



(Resource-Allocation - Graph)

Cycles in the wait-for-graph indicate deadlock. This algorithm must maintain the wait-for-graph & periodically search for closed loops.

→ RECOVERY FROM DEADLOCK :

Another possibility is to system recover from the deadlock automatically.

(i.) Process Termination : Terminate all the processes involved in the deadlock. Terminate processes one by one until the

deadlock is broken.

(ii.) Resource Preemption : When preempting resources to relieve deadlock

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

(b.) Rollback : Ideally one would like to rollback a preempted process to a safe state prior to the point at which that resource was originally allocated to the process.

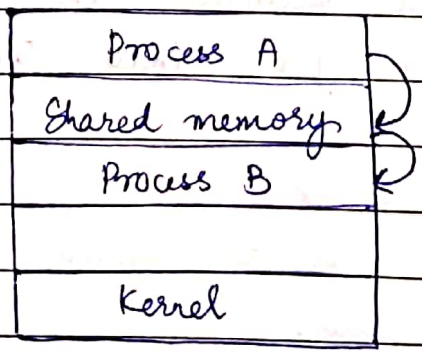
(c.) Starvation : A process would not starve because its resources are constantly being preempted.

⇒ INTER PROCESS COMMUNICATION (IPC)

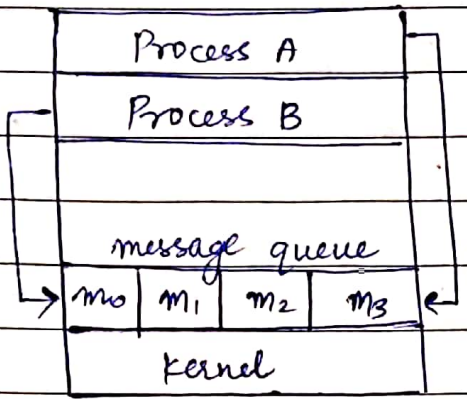
IPC is a mechanism which allows processes to communicate each other and synchronize their actions.

The communication b/w these processes can be recognized as a method of co-operation b/w them. Process can communicate with each other using two methods: Shared memory & message passing.

(i.) SHARED MEMORY e.g: Producer Consumer Problem



(2) MESSAGE PASSING :



In this mechanism, there are two types of primitives which can be used for exchanging no. of messages:

- send (message, destination)
- receive (message, host)

— Homework —

Ques -

Processes	Allocation			MAX			Available			Need		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
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

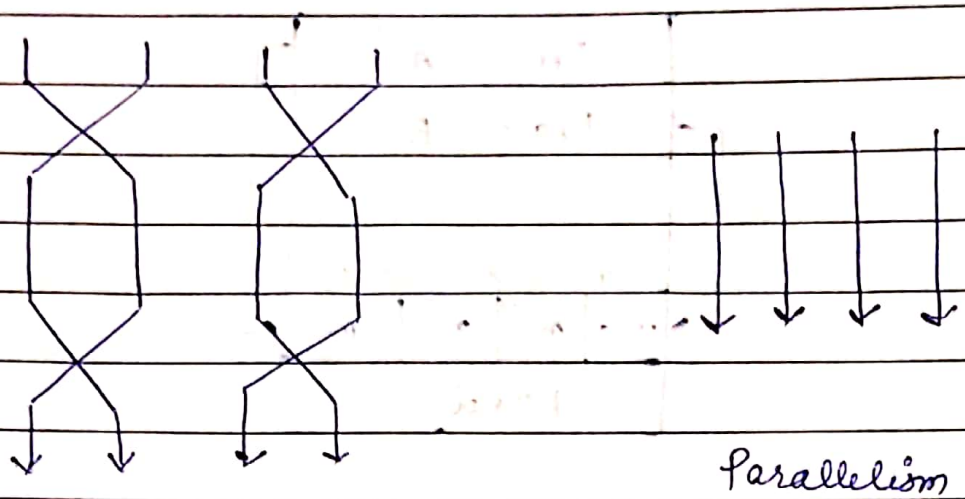
	X	Y	Z
P_0 :	0	1	0

Ques -

Processes	Allocation			Max			Available			Need		
	X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
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						

	X	Y	Z
P_1 :	1	0	2

⇒ CONCURRENT PROCESSES



Concurrency

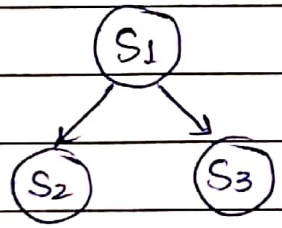
Parallelism

For representing concurrent processes, we may utilise two types of presentations :

1. Parallel begin (parbegin) & Parallel end (parend)
2. Fork & Join Construct

→ TYPE 1 : PARBEGIN & PAREND REPRESENTATION :

Q1-



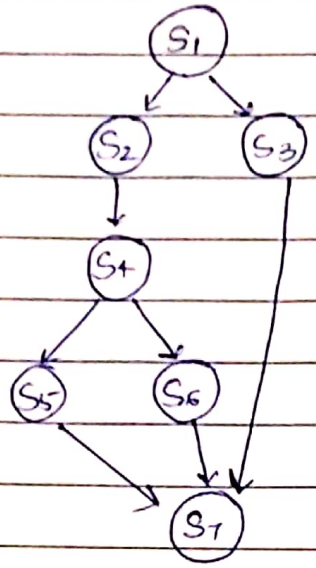
S_1 ⇒ statement ⇒ $x = a + b$

S_2 ⇒ $y = x + z$

S_3 ⇒ $w = x + a$

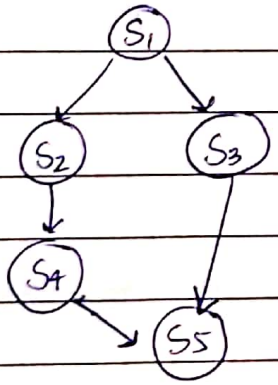
```
begin
S1;
parbegin
S2;
S3;
parend
end
```

(Q2-)



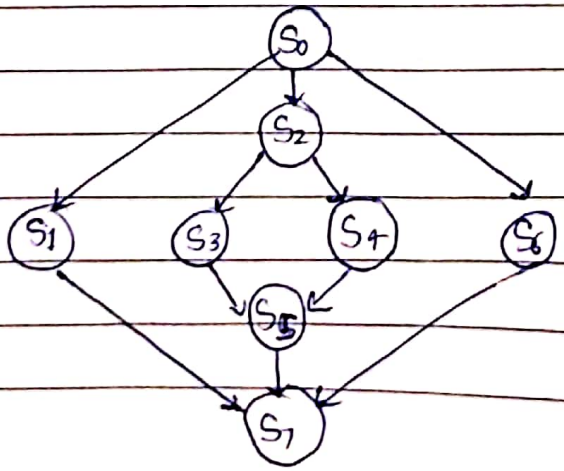

```
begin
S1;
parbegin
S2;
begin
S2;
S4;
parbegin
S5;
S6;
parend
end
parend
S7;
end
```

(Q3-)



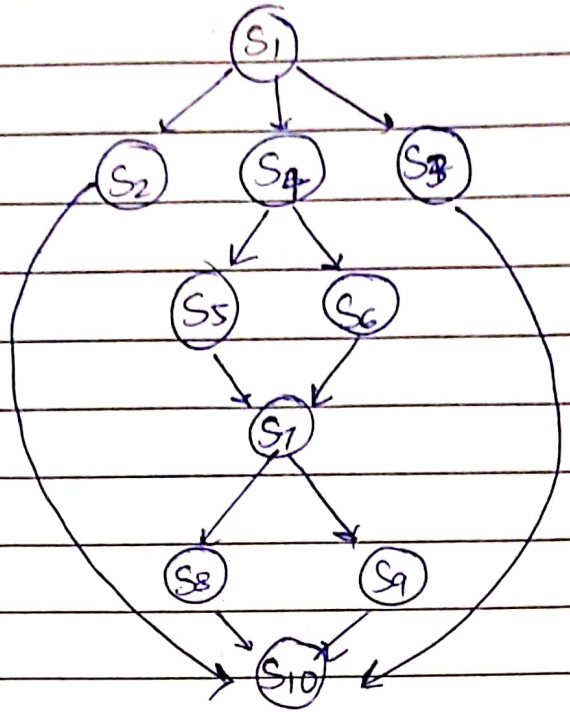

```
begin
S1;
parbegin
S3;
begin
S2;
S4;
end
parend
S5;
end
```

(Q4-)




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

Q5-)




```
begin
S1;
parbegin
S2;
S3;
begin
S4;
parbegin
S5;
S6;
parend
S7;
parbegin
S8;
S9;
parend
end;
parend
S10;
end
```

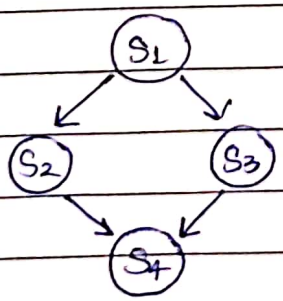
→ **TYPE 2 : FORK & JOIN CONSTRUCT :**

fork is called by a thread (parent) to create a new thread (child process) of concurrency.

Join recombines two concurrent computations into 1. It is called by both the parent & child process.

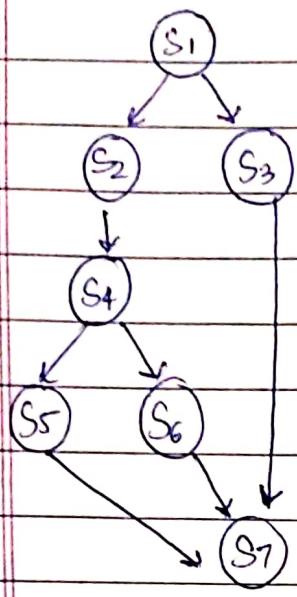
fork increases the concurrency & join decreases the concurrency.

e.g:



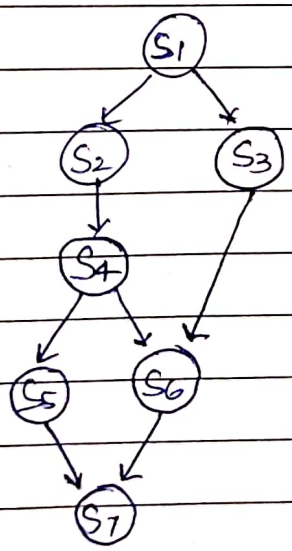

```
count = 2;
S1;
fork L1;
S2;
goto L2;
L1 : S3;
L2 : Join count;
S4;
```

Ques-



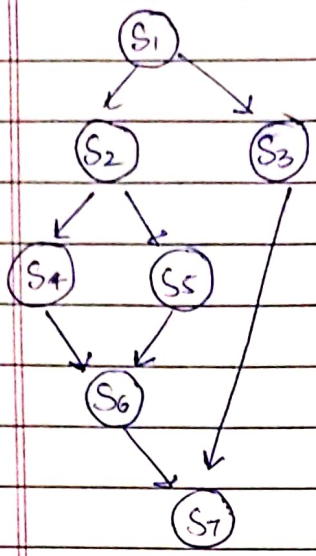

```
count = 3;
S1;
Fork L1;
S2;
S4;
fork L2;
S5;
goto L3;
L2: S6;
goto L3;
L1: S3;
L3: Join count;
S7;
```

Ques-




```
count1 = 2;
S1;
fork L1;
S2;
S4;
count2 = 2;
fork L3;
S5;
goto L2;
L1: S3;
L2: Join count1;
L3: S6;
L4: Join count2;
goto L4;
S7;
```

Ques -




```
begin
S1;
fork L1;          L1 : S3
S2;               goto L4;
fork L2;          L2 : S5;
S4;
L3 : join         goto L3
S6;
L4 : Join
S7;
end
```

⇒ SEMAPHORE QUEUE

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

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

⇒ CONCURRENCY CONDITIONS / BERNSTEIN'S CONDITION

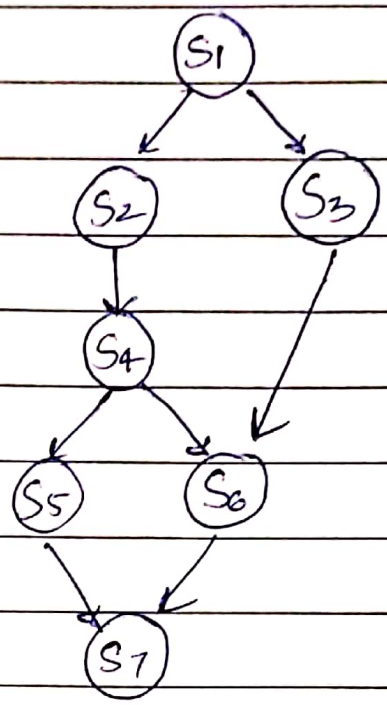
When two statements in a program be executed concurrently & still produce the same result, the following 3 conditions must be satisfied.

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

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

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

```
S1;
Count 1 := 2;
fork L1;
S2;
S4;
count2 = 2;
fork L3;
S5;
goto L4;
L1: S3;
L2: Join count 1;
L3: S6;
L4: Join count 2;
goto L2;
S7;
```




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