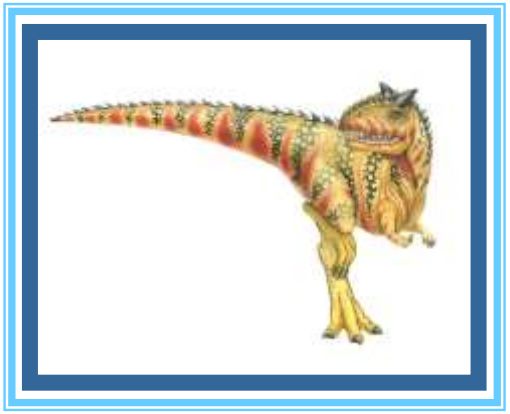
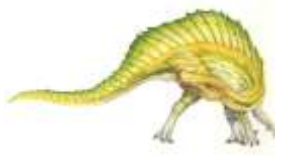
Practice Problems



Silberschatz, Galvin and Gagne ©2013

Process Synchronization



Problem 1

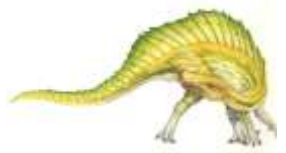
- The enter_CS( ) and leave_CS( ) functions to implement critical section of a process are realized using **test-and-set** instruction as follows:

```
void enter_CS(X)
{
   while (test-and-set(X));
}

void leave_CS(X)
{
   X = 0;
}
```

In the above solution, X is a memory location associated with the CS and is initialized to 0. Now, consider the following statements:

A. The above solution to CS problem is deadlock-free
B. The solution is starvation free
C. The processes enter CS in FIFO order
D. More than one process can enter CS at the same time.



Problem 1: Solution

```
boolean test_and_set (boolean *target)
          {
               boolean rv = *target;
               *target = TRUE;
               return rv:
          }
```

Clearly, the given mechanism is test-and-set lock which has the following characteristics

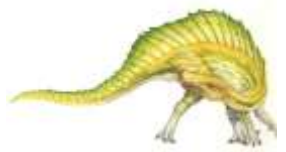
a) It ensures **mutual exclusion**.

b) It ensures **freedom from deadlock**.

c) It may cause the process to **starve** for the CPU.

d) It does not guarantee that processes will always execute in a **FIFO** order otherwise there would be **no starvation**.

Thus, Option (A) is correct.



Problem 2

- Fetch_And_Add(X,i) is an atomic Read-Modify-Write instruction that reads the value of memory location X, increments it by the value *i* and, returns the old value of X.

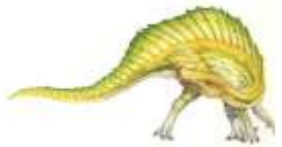
- It is used in the pseudo code shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available.

```
AcquireLock(L)
{
  while (Fetch_And_Add(L,1))
        L = 1;
}

ReleaseLock(L)
{
  L = 0;
}
```

This implementation

A. Fails as L can overflow
B. Fails as L can take on a non-zero value when the lock is actually available
C. Works correctly but may starve some processes
D. Works correctly without starvation



Problem 2: Solution

The given synchronization mechanism has been implemented as-

```
while ( Fetch_And_Add(L, 1))
L = 1;
```

Entry Section

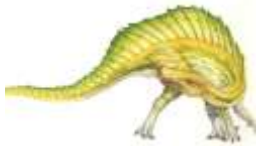
Critical Section

```
Lock = 0
```

Exit Section

Working-

This synchronization mechanism works as explained in the following scenes-



Problem 2: Solution

Scene-01:

- A process P_1 arrives.
- It executes the Fetch_And_Add(L, 1) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 0+1=1.
- The returned value 0 breaks the while loop condition.

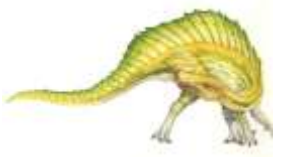
Scene-02:

- Another process P_2 arrives.
- It executes the Fetch_And_Add(L, 1) instruction.
- Since lock value is now set to 1, so it returns value 1 to the while loop and sets the lock value to 1+1=2.
- The returned value 1 does not break the while loop condition.
- Process P_2 executes the next instruction L=1 and sets the lock value to 1 and again checks the condition.
- The lock value keeps changing from 1 to 2 and then 2 to 1.
- The process P_2 is trapped inside an infinite while loop.
- The while loop keeps the process P_2 busy until the lock value becomes 0 and its condition breaks.

Problem 2: Solution

Scene-03:

- Process P_1 comes out of the critical section and sets the lock value to 0.
- The while loop condition breaks.
- Now, process P_2 waiting for the critical section enters the critical section.



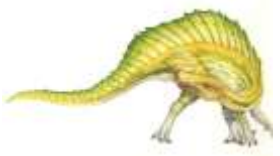
Problem 2: Solution

Failure of the Mechanism-

- The mechanism fails to provide the synchronization among the processes.
- This is explained below-

Explanation-

The occurrence of following scenes may lead to two processes present inside the critical section-



Problem 2: Solution

Scene-01:

- A process P_1 arrives.
- It executes the Fetch_And_Add(L, 1) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 0+1=1.
- The returned value 0 breaks the while loop condition.
- Process P_1 enters the critical section and executes.

Scene-02:

- Another process P_2 arrives.
- It executes the Fetch_And_Add(L, 1) instruction.
- Since lock value is now set to 1, so it returns value 1 to the while loop and sets the lock value to 1+1=2.
- The returned value 1 does not break the while loop condition.
- Now, as process P_2 is about to enter the body of while loop, it gets preempted.

Problem 2: Solution

Scene-03:

- Process P_1 comes out of the critical section and sets the lock value to 0.

Scene-04:

- Process P_2 gets scheduled again.
- It resumes its execution.
- Before preemption, it had already satisfied the while loop condition.
- Now, it begins execution from next instruction.
- It sets the lock value to 1 and here the blunder happens.
- This is because the lock is actually available and lock value = 0 but P_2 itself sets the lock value to 1.
- Then, it checks the condition and now there is no one who can set the lock value to zero.
- Thus, Process P_2 gets trapped inside the infinite while loop forever.
- All the future processes too will be trapped inside the infinite while loop forever.

Problem 2: Solution

Thus,

- Although mutual exclusion could be guaranteed but still the mechanism fails.
- This is because lock value got set to a non-zero value even when the lock was available.

Also,

> This synchronization mechanism leads to overflow of value 'L'.

Explanation-

- When every process gets preempt after executing the while loop condition, the value of lock will keep increasing with every process.
- When first process arrives, it returns the value 0 and sets the lock value to 0+1=1 and gets preempted.
- When second process arrives, it returns the value 1 and sets the lock value to 1+1=2 and gets preempted.
- When third process arrives, it returns the value 2 and sets the lock value to 2+1=3 and gets preempted.
- Thus, for very large number of processes preempting in the above manner, L will overflow.

gne ©2013

Problem 2: Solution

Also,

> This synchronization mechanism does not guarantee bounded waiting and may cause starvation.

Explanation-

- There might exist an unlucky process which when arrives to execute the critical section finds it busy.
- So, it keeps waiting in the while loop and eventually gets preempted.
- When it gets rescheduled and comes to execute the critical section, it finds another process executing the critical section.
- So, again, it keeps waiting in the while loop and eventually gets preempted.
- This may happen several times which causes that unlucky process to starve for the CPU.

Thus, Options (A) and (B) are correct.

Problem 3

- Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared Boolean variables S1 and S2 are randomly assigned.

Method used by P_1	Method used by P_2
while (S_1 == S_2);	while (S_1 != S_2);
Critical Section	Critical Section
$S_1 = S_2$	$S_2 = !S_1$

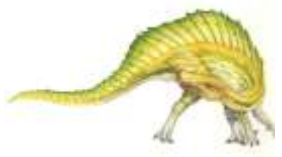
Which one of the following statements describes the properties achieved?

1. Mutual exclusion but not progress
2. Progress but not mutual exclusion
3. Neither mutual exclusion nor progress
4. Both mutual exclusion and progress

Problem 3: Solution

The initial values of shared Boolean variables S_1 and S_2 are randomly assigned. The assigned values may be-

- $S_1 = 0$ and $S_2 = 0$
- $S_1 = 0$ and $S_2 = 1$
- $S_1 = 1$ and $S_2 = 0$
- $S_1 = 1$ and $S_2 = 1$



Problem 3: Solution

Case-01: If $S_1 = 0$ and $S_2 = 0$-

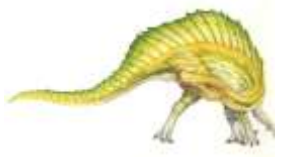
In this case,

- Process P_1 will be trapped inside an infinite while loop.
- However, process P_2 gets the chance to execute.
- Process P_2 breaks the while loop condition, executes the critical section and then sets $S_2 = 1$.
- Now, $S_1 = 0$ and $S_2 = 1$.
- Now, process P_2 can not enter the critical section again but process P_1 can enter the critical section.
- Process P_1 breaks the while loop condition, executes the critical section and then sets $S_1 = 1$.
- Now, $S_1 = 1$ and $S_2 = 1$.
- Now, process P_1 can not enter the critical section again but process P_2 can enter the critical section.

Problem 3: Solution

Thus,

- Processes P1 and P2 executes the critical section alternately starting with process P_2.
- Mutual exclusion is guaranteed.
- Progress is not guaranteed because if one process does not execute, then other process would never be able to execute again.
- Processes have to necessarily execute the critical section in strict alteration.



Problem 3: Solution

Case-02: If $S_1 = 0$ and $S_2 = 1$-

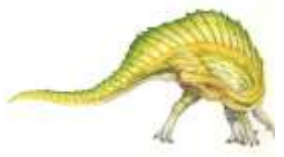
In this case,

- Process P_2 will be trapped inside an infinite while loop.
- However, process P_1 gets the chance to execute.
- Process P_1 breaks the while loop condition, executes the critical section and then sets $S_1 = 1$.
- Now, $S_1 = 1$ and $S_2 = 1$.
- Now, process P_1 can not enter the critical section again but process P_2 can enter the critical section.
- Process P_2 breaks the while loop condition, executes the critical section and then sets $S_2 = 0$.
- Now, $S_1 = 1$ and $S_2 = 0$.
- Now, process P_2 can not enter the critical section again but process P_1 can enter the critical section.

Problem 3: Solution

Thus,

- Processes P1 and P2 executes the critical section alternately starting with process P_1.
- Mutual exclusion is guaranteed.
- Progress is not guaranteed because if one process does not execute, then other process would never be able to execute again.
- Processes have to necessarily execute the critical section in strict alteration.



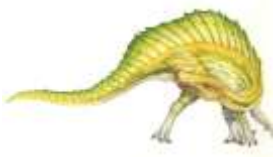
Problem 3: Solution

Case-03: If $S_1 = 1$ and $S_2 = 0$-

- This case is same as case-02.

Case-04: If $S_1 = 1$ and $S_2 = 1$-

- This case is same as case-01.

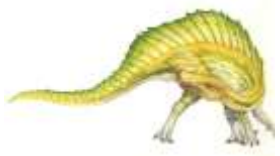


Problem 3: Solution

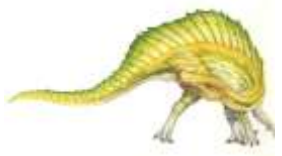
Thus, Overall we can conclude-

- Processes P1 and P2 executes the critical section alternatively.
- Mutual exclusion is guaranteed.
- Progress is not guaranteed because if one process does not execute, then other process would never be able to execute again.
- Processes have to necessarily execute the critical section in strict alteration.

Thus, Option (A) is correct.

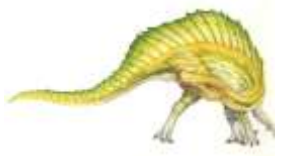


Counting Semaphores



Problem 1

- A counting semaphore S is initialized to 10. Then, 6 P operations and 4 V operations are performed on S. What is the final value of S?



Problem 1: Solution

- P operation also called as wait operation decrements the value of semaphore variable by 1.
- V operation also called as signal operation increments the value of semaphore variable by 1.

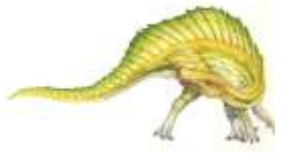
Thus,

Final value of semaphore variable S

= 10 – (6 x 1) + (4 x 1)

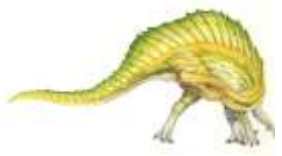
= 10 – 6 + 4

= 8



Problem 2

- A counting semaphore S is initialized to 7. Then, 20 P operations and 15 V operations are performed on S. What is the final value of S?



Problem 2: Solution

We know-

- P operation also called as wait operation decrements the value of semaphore variable by 1.
- V operation also called as signal operation increments the value of semaphore variable by 1.

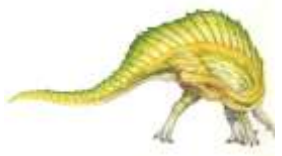
Thus,

Final value of semaphore variable S

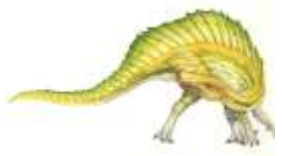
= 7 – (20 x 1) + (15 x 1)

= 7 – 20 + 15

= 2



Binary Semaphores



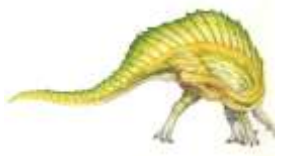
Problem 1

Each process P_i, i = 1, 2, …, 9 is coded as follows-

```
1.  repeat
2.     P(mutex)
3.     { Critical Section }
4.     V(mutex)
5.  forever
```

The code for P_{10} is identical except that it uses V(mutex) in place of P(mutex). What is the largest number of processes that can be inside the critical section at any moment?

1. 1
2. 2
3. 3
4. None of these



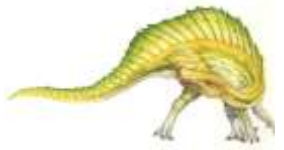
Problem 1: Solution

Code for Processes $P_1, P_2, \ldots, P_9$

```
1.  repeat
2.     P(mutex)
3.     { Critical Section }
4.     V(mutex)
5.  forever
```

Code for Process P_{10}

```
1.  repeat
2.     V(mutex)
3.     { Critical Section }
4.     V(mutex)
5.  forever
```

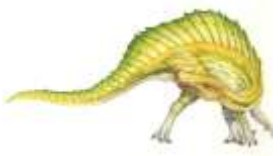


Problem 1: Solution

Consider mutex is initialized with 1.

Then-

- All the 10 processes may be present inside the critical section at the same time.
- If there would have been more number of processes, then they could also be present inside the critical section at the same time.



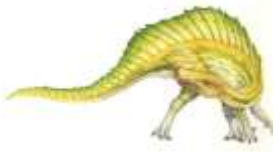
Problem 1: Solution

Scene-01:

- Initially, process P_1 arrives.
- It performs the wait operation on mutex and sets its value to 0.
- It then enters the critical section.

Scene-02:

- Consider the processes P_2, P_3, … , P_9 arrives when process P_1 is executing the critical section.
- All the other processes get blocked and are put to sleep in the waiting list.



Problem 1: Solution

Scene-03:

- Process P_{10} arrives.
- It performs the signal operation on mutex.
- It selects one process from the waiting list say process P_2 and wakes it up.
- Process P_2 can now enter the critical section (P_1 is already present).
- Now, process P_{10} enters the critical section (P_1 and P_2 are already present).
- After executing critical section, during exit, it again performs the signal operation on mutex.
- It selects one process from the waiting list say process P_3 and wakes it up.
- Process P_3 can now enter the critical section (P_1 and P_2 are already present).
- Process P_{10} may keep on executing repeatedly.
- It wakes up 2 processes each time during its course of execution.
- In this manner, all the processes blocked in the waiting list may get entry inside the critical section.

Thus, Option (D) is correct.

Problem 2

Let m[0]…m[4] be mutexes (binary semaphores) and P[0]…P[4] be processes. Suppose each process P[i] executes the following:

```
wait(m[i]);

wait(m[(i+1) mod 4]);

……

release(m[i]);

release(m[(i+1) mod 4]);
```

This could cause-

1. Thrashing
2. Deadlock
3. Starvation but not deadlock
4. None of the above

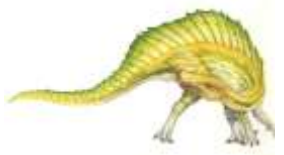
Problem 2: Solution

Given processes have to execute the wait operations as-

- Process P_0 : wait(m[0]); wait(m[1]);
- Process P_1 : wait(m[1]); wait(m[2]);
- Process P_2 : wait(m[2]); wait(m[3]);
- Process P_3 : wait(m[3]); wait(m[4]);
- Process P_4 : wait(m[4]); wait(m[1]);

Now,

- A deadlock may be caused when different processes execute the wait operations on mutexes.
- The occurrence of following scenes will give birth to a deadlock.



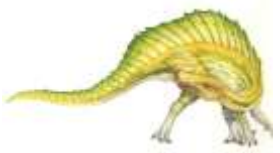
Problem 2: Solution

Scene-01:

- Process P_0 arrives.
- It executes wait(m[0]) and gets preempted.

Scene-02:

- Process P_1 gets scheduled.
- It executes wait(m[1]) and gets preempted.



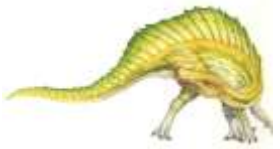
Problem 2: Solution

Scene-03:

- Process P_2 gets scheduled.
- It executes wait(m[2]) and gets preempted.

Scene-04:

- Process P_3 gets scheduled.
- It executes wait(m[3]) and gets preempted.



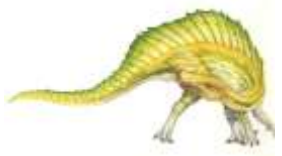
Problem 2: Solution

Scene-05:

- Process P_4 gets scheduled.
- It executes wait(m[4]) and gets preempted.

Now,

- The system is in a deadlock state since no process can proceed its execution.
- Thus, Option (B) is correct.



Problem 3

In the above question, which of the following pairs of processes may be present inside the critical section at the same time?

1. (P_0, P_2)
2. (P_1, P_3)
3. (P_2, P_4)
4. (P_3, P_4)
5. All of these

?

P_3 get M_3 & M_0

P_4 get M_4 & M_1

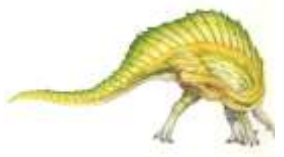
↓ 无 mod

↓ 有 mod

Problem 3: Solution

- All the given pair of processes in options execute wait operation on different mutexes.
- Hence, they can get entry inside the critical section at the same time.
- Thus, Option (E) is correct.

- Mutual exclusion is violated here.
- Maximum number of processes that may be present inside the critical section at the same time = 2.



Problem 4

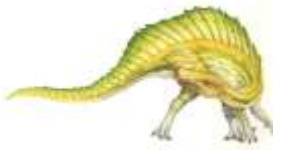
The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0 and S2 = 0.

Process P0	Process P1	Process P2

while (true)
{
wait (S0);
print '0'
release (S1);
release (S2);
}
	wait (S1);
	release (S0);
		wait (S2);
		release (S0);

How many times will process P0 print '0'?

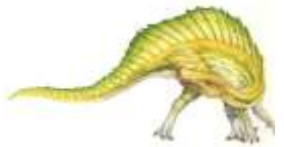
1. At least twice
2. Exactly twice
3. Exactly thrice
4. Exactly once



Problem 4: Solution

Scene-01:

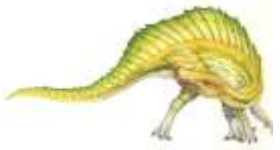
- Process P0 arrives.
- It executes wait operation on semaphore S0 successfully. Now, S0 = 0.
- It then **prints '0'**. (1st time)
- It executes signal operation on semaphore S1. Now, S1 = 1.
- It executes signal operation on semaphore S2. Now, S2 = 1.
- While loop causes process P0 to execute again.
- It executes wait operation on semaphore S0 unsuccessfully and gets blocked.



Problem 4: Solution

Scene-02:

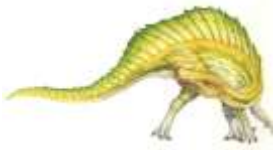
- Process P1 gets scheduled.
- It executes wait operation on semaphore S1 successfully. Now, S1 = 0.
- It executes signal operation on semaphore S0 which wakes up the process P0.
- The execution of process P1 is completed.



Problem 4: Solution

Scene-03:

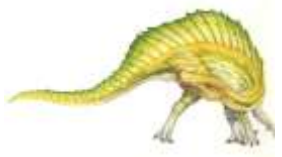
- Process P0 gets scheduled again.
- It **prints '0'**. (2nd time)
- It executes signal operation on semaphore S1. Now, S1 = 1.
- It executes signal operation on semaphore S2. Now, S2 = 1.
- While loop causes process P0 to execute again.
- It executes wait operation on semaphore S0 unsuccessfully and gets blocked.



Problem 4: Solution

Scene-04:

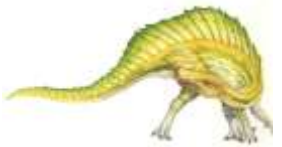
- Process P2 gets scheduled.
- It executes wait operation on semaphore S2 successfully. Now, S2 = 0.
- It executes signal operation on semaphore S0 which wakes up the process P0.
- The execution of process P2 is completed.



Problem 4: Solution

Scene-05:

- Process P0 gets scheduled again.
- It **prints '0'**. (3rd time)
- It executes signal operation on semaphore S1. Now, S1 = 1.
- It executes signal operation on semaphore S2. Now, S2 = 1.
- While loop causes process P0 to execute again.
- It executes wait operation on semaphore S0 unsuccessfully and gets blocked.

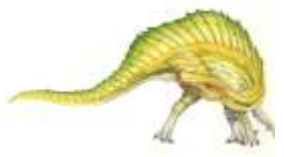


Problem 4: Solution

Now,

- The execution of processes P1 and P2 is already completed.
- There is no other process in the system which can perform signal operation on semaphore S0.
- Thus, process P0 can not execute any more.

Thus, maximum number of times process P0 can print ‘0’ = 3 times.



Problem 5

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S1 and S2. The code for the processes P and Q is shown below-

Process P:

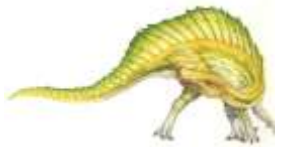
```
1.  while(1)
2.  {
3.      P(S1);
4.      P(S2);
5.      Critical Section
6.      V(S1);
7.      V(S2);
8.  }
```

Process Q:

```
1.  while(1)
2.  {
3.      P(S1);
4.      P(S2);
5.      Critical Section
6.      V(S1);
7.      V(S2);
8.  }
```

This ensures-

1. Mutual Exclusion
2. Deadlock
3. Starvation but not deadlock
4. None of these

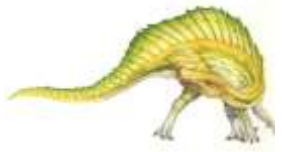


Problem 5: Solution

- The process which so ever arrives first performs the wait operation on the semaphores.
- If the other process arrives in the middle, it will get blocked and put to sleep in the waiting list.
- When the former process performs the signal operation on semaphores, it wakes up the latter process.
- This ensures mutual exclusion.
- There is no possibility of deadlock or starvation.

Thus, Option (A) is correct.

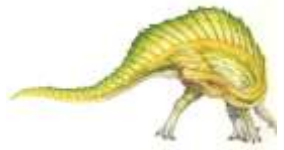
- In the code of above processes, there is no need of using two semaphores.
- Mutual exclusion could be achieved by using only one semaphore too.



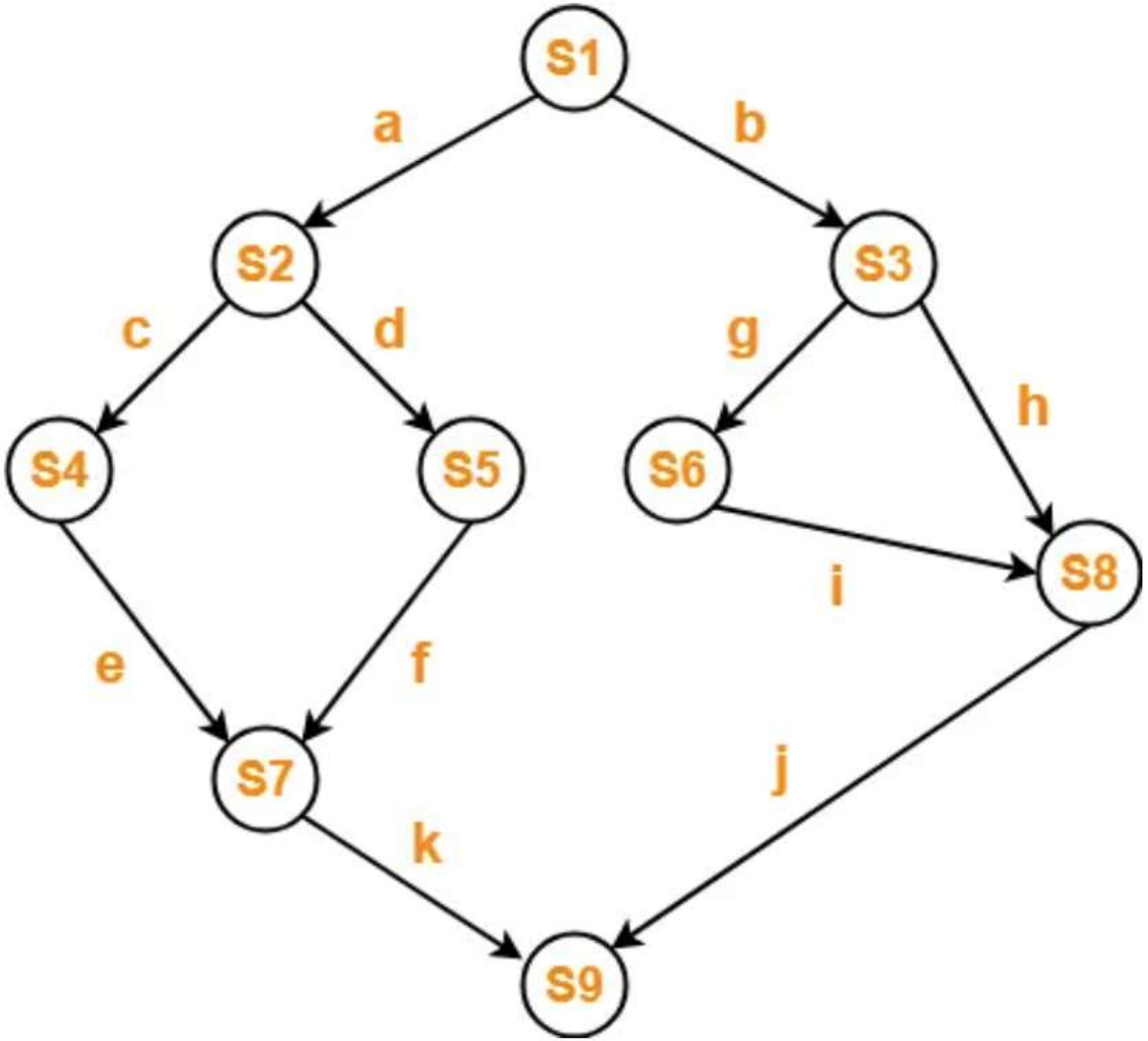
Problem 6

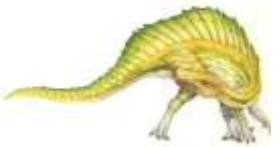
Consider the following program segment for concurrent processing using semaphore operators P and V for synchronization. Draw the precedence graph for the statements S_1 to S_9.

```
var
a, b, c, d, e, f, g, h, i, j, k : semaphore;
begin
cobegin
    begin S1; V(a); V(b) end;
    begin P(a); S2; V(c); V(d) end;
    begin P(c); S4; V(e) end;
    begin P(d); S5; V(f) end;
    begin P(e); P(f); S7; V(k) end
    begin P(b); S3; V(g); V(h) end;
    begin P(g); S6; V(i) end;
    begin P(h); P(i); S8; V(j) end;
    begin P(j); P(k); S9 end;
coend
end;
```

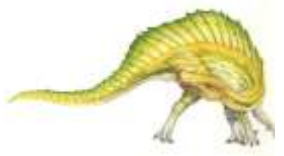


Problem 6: Solution





Deadlock

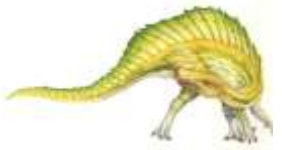


Problem 1

Consider there are n processes in the system P_1, P_2, P_3, …… , P_n where-

- Process P_1 requires x_1 units of resource R
- Process P_2 requires x_2 units of resource R
- Process P_3 requires x_3 units of resource R and so on.

What is the minimum number of units of resource R that ensures no deadlock?



Problem 1: Solution

In worst case,

> The number of units that each process holds = One less than its maximum demand

So,

- Process P_1 holds $x_1 - 1$ units of resource R
- Process P_2 holds $x_2 - 1$ units of resource R
- Process P_3 holds $x_3 - 1$ units of resource R and so on.

Now,

- Had there been one more unit of resource R in the system, system could be ensured deadlock free.
- This is because that unit would be allocated to one of the processes and it would get execute and then release its units.

Problem 1: Solution

Maximum number of units of resource R that ensures deadlock

$= (x_1\text{-}1) + (x_2\text{-}1) + (x_3\text{-}1) + \ldots. + (x_n\text{-}1)$

$= (x_1 + x_2 + x_3 + \ldots. + x_n) - n$

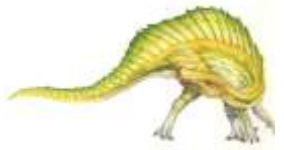
$= \sum x_i - n$

= Sum of max needs of all n processes – n

Minimum number of units of resource R that ensures no deadlock

= One more than maximum number of units of resource R that ensures deadlock

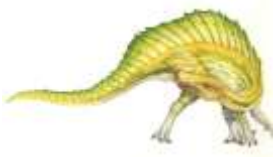
$= (\sum x_i - n) + 1$



Problem 2

A system is having 3 user processes each requiring 2 units of resource R. The minimum number of units of R such that no deadlock will occur-

1. 3
2. 5
3. 4
4. 6



Problem 2: Solution

In worst case,

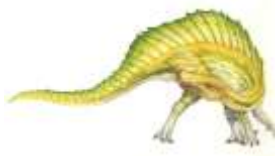
The number of units that each process holds = One less than its maximum demand

So,

- Process P1 holds 1 unit of resource R
- Process P2 holds 1 unit of resource R
- Process P3 holds 1 unit of resource R

Thus,

- Maximum number of units of resource R that ensures deadlock = 1 + 1 + 1 = 3
- Minimum number of units of resource R that ensures no deadlock = 3 + 1 = 4

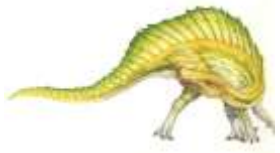


Problem 3

A system is having 10 user processes each requiring 3 units of resource R. The minimum number of units of R such that no deadlock will occur _____?

Solution-

- Maximum number of units of resource R that ensures deadlock = 10 x 2 = 20
- Minimum number of units of resource R that ensures no deadlock = 20 + 1 = 21

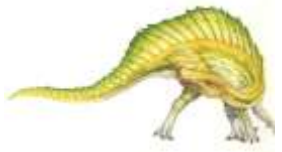


Problem 4

A system is having 3 user processes P1, P2 and P3 where P1 requires 2 units of resource R, P2 requires 3 units of resource R, P3 requires 4 units of resource R. The minimum number of units of R that ensures no deadlock is _____?

Solution-

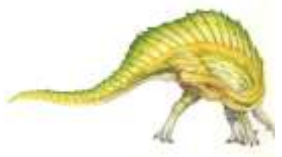
- Maximum number of units of resource R that ensures deadlock = 1 + 2 + 3 = 6
- Minimum number of units of resource R that ensures no deadlock = 6 + 1 = 7



Problem 5

Consider a system having m resources of the same type being shared by n processes. Resources can be requested and released by processes only one at a time. The system is deadlock free if and only if-

1. The sum of all max needs is < m+n
2. The sum of all max needs is > m+n
3. Both of above
4. None of these



Problem 5: Solution

We have derived above-

Maximum number of units of resource R that ensures deadlock = $(\sum x_i - n)$

Thus, For no deadlock occurrence,

Number of units of resource R must be > $(\sum x_i - n)$

i.e. $m > (\sum x_i - n)$

or $\sum x_i < m + n$

Thus, Correct Option is (A).