Synchronization

1) For correct and efficient operation using shared data, a synchronization solution must ensure: **[MSQ]**
A) No two processes can be inside their critical sections at same time.
B) No assumptions needed about relative process speeds or number of CPUs.
C) No process stopped outside its critical section can block other processes.
D) No process can be indefinitely postponed from entering its critical section.

Solution: (A)(B)(C)(D)
Mutual exclusion: No two processes can be inside their critical sections at same time.
Bounded waiting: No assumptions needed about relative process speeds or number of CPUs.
Progress: No process stopped outside its critical section can block other processes.
Progress: No process can be indefinitely postponed from entering its critical section.

2) _____ is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work, indefinitely.
A) Spin lock
B) Deadlock
C) Livelock
D) None of these.

Solution: (C)

Livelock is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work, indefinitely. This is similar to deadlock in that no progress is made, but differs in that neither process is blocked or waiting for anything.

3) Which of the following statements is/are correct about the Test-And-Set lock (TSL) synchronization mechanism? **[MSQ]**
A) TSL requires hardware support for its implementation.
B) TSL does not guarantee bounded waiting.
C) TSL may suffer from priority inversion problems.
D) TSL may lead to deadlock.

Solution: (A) (B) (C)
TSL is hardware implementation, which does not ensure bounded waiting. TSL may sometimes suffer from priority inversion problems, leading to livelocks.

4) Which of the following is a busy wait or uses spin lock for synchronization? **[MSQ]**
A) Test-And-Set lock (TSL)
B) Peterson's solution
C) Compare-and-Swap instruction (CAS)
D) Semaphores

Solution: (A) (B) (C)
Semaphores do not involve busy waiting or spin locks. If the lock or semaphore is not

available then the process is blocked (sleep). Whereas, TSL,CAS, and Petersons solutions are busy waiting mechanisms.

5) Consider the following code for process synchronization using a busy waiting mechanism (spin lock). The system employs Round robin scheduling with a quantum of 1 cpu clock cycle (cc), and each high level instruction takes 1 clock cycle for execution. Assume process P0 is scheduled first and the shared variable turn is initialized to 1. The time taken to complete the execution of both the processes is _____ clock cycles. **[NAT]**

Process P0	Process P1

while(turn==1);
a=a+b;
turn=1;
	while(turn==0);
	a=a+b;
	turn=0;

Solution: 9

Time	P0	P1	CC
0	while(turn==1); [true: loop]		1
1		while(turn==0); [false]	1
2	while(turn==1); [true: loop]		1
3		a=a+b;	1
4	while(turn==1); [true: loop]		1
5		turn=0; [completed]	1
6	while(turn==1); [false]		1
7	a=a+b;		1
8	turn=1; [completed]		1
		Total clock cycles:	9

The above problem demonstrates the overhead of using spin locks (busy waiting) solutions. It can be noted that only 2 clock cycles (a=a+b) is useful work, rest 7 clock cycles are overhead.

6) Consider the following partially filled table obtained as a result of operation by two processes P1 and P2, on two counting semaphores S1 and S2, both initialized to 0.

Time	Process	Action	Blocked	S1	S2
0	-	-	-	0	0
1	P1	V(S1)		1	0
2	P1	P(S2)	P1	1	***X***
3	P2	V(S2)		1	0
4	P2	***Y***		0	0

The correct options for the place holders X and Y are ?
A) X = -1, Y = P(S1)
B) X = 1, Y = P(S2)
C) X = 0, Y = V(S1)
D) X = -1, Y = V(S2)

Solution: (A)

Time	Process	Action	Blocked	S1	S2
0	-	-	-	0	0
1	P1	V(S1)		1	0
2	P1	P(S2)	P1	1	-1
3	P2	V(S2)		1	0
4	P2	P(S1)		0	0

7) Consider the following synchronization mechanism for Producer/Consumer problem, where,
Producer:

- creates data and adds to the buffer
- do not want to overflow the buffer

Consumer:

- removes data from buffer (consumes it)
- does not want to get ahead of producer

Information common to both processes:
empty := n // where n is the number of buffer slots.
full := 0

mutex := 1

Producer Process	Consumer Process

repeat
produce an item in nextp
 	
wait(empty);
wait(mutex);
 	
add nextp to buffer
 	
signal(mutex);
signal(full);
until false;
	repeat
	
	wait(full);
	wait(mutex);
	
	remove an item from buffer to nextc
	
	signal(mutex);
	signal(empty);
	
	consume the item in nextc
	
	until false;

Which of the following are correct about the above mechanism?**[MSQ]**

A) The above mechanism is the correct implementation for the Producer Consumer problem.
B) The mutex semaphore is used to ensure mutually exclusive access to the buffer.
C) The empty and full semaphores are used for process synchronization.
D) If we change the code in the consumer process from: wait(full), wait(mutex)
to wait(mutex), wait(full), then a deadlock may occur.

Solution: (A)(B)(C)(D)
The empty and full semaphores are used for process synchronization. The mutex semaphore is used to ensure mutually exclusive access to the buffer.
If we change the code in the consumer process from:
wait(full)
wait(mutex)
to
wait(mutex)
wait(full),
then we could reach a deadlock where Process 1 is waiting for Process 2 and Process 2 is waiting for Process 1.

8) Identify the correct statement from the following :

A) The Test-and-Set Lock hardware synchronization mechanism suffers from spin-lock when a round-robin scheduling algorithm is used.
B) Peterson’s solution to the mutual-exclusion problem works for non-preemptive scheduling.
C) An operating system that can disable interrupts can implement semaphores.
D) None of these.

Solution (C)

With round-robin scheduling TSL works. Sooner or later all the processes will run, and eventually leave their critical regions.
Peterson's solution certainly works with preemptive scheduling. In fact, it was designed for that case. When scheduling is non preemptive, it might fail. Consider the case in which turn is initially 0 but process 1 runs first. It will just loop forever and never release the CPU.
To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore. If it is doing a down and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If it is doing an up , it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.

9) Consider the following synchronization mechanism using semaphores S1 and S2 :

PROCESS 1	PROCESS 2

While (1) {
// Do something
S1.Wait();
Print("1");
S2.signal();
//Do something
}
	While (1) {
	// Do something
	S2.Wait();
	Print("2");
	S1.signal();
	//Do something
	}

The initial value of S1 and S2 so that we can obtain the sequence 1,2,1,2,1,2..... are
A. S1=0, S2=1
B. S1=1, S2=0
C. S1=0, S2=0
D. S1=1, S2=1

Solution: (B)
"1" has to be printed first, hence Process 1 should be executed first. So we make S1=1 and S2=0. Process 1 uses S2.signal() to notify process 2 and then Process 2 prints "2". This goes on….

10) Select the correct options among the following: **[MSQ]**

A) Starvation-freedom imply deadlock-freedom.
B) Starvation-freedom imply bounded-waiting.
C) Bounded-waiting imply starvation-freedom
D) We need Progress and Bounded-Waiting both to imply Deadlock-Freedom.

Solution: (A) (D)
I) Does starvation-freedom imply deadlock-freedom?

Yes! If every process can eventually enter its critical section, although waiting time may vary, it means the decision time of selecting a process is finite. Otherwise, all processes would wait in the entry section.

II) Does starvation-freedom imply bounded-waiting?
No! This is because the waiting time may not be bounded even though each process can enter its critical section.

III) Does bounded-waiting imply starvation-freedom?
No! Bounded-Waiting does not say if a process can actually enter. It only says there is a bound. For example, all processes are locked up in the entry section (i.e., failure of Progress).

IV) We need Progress and Bounded-Waiting both to imply Deadlock-Freedom
Yes! progress along with bounded waiting makes starvation-free, and starvation-free implies deadlock freedom.

11) Consider two processes A and B that uses binary semaphore to implement synchronization mechanism on three resources associated with binary semaphores a,b and c. The process A access resources as follows :
A : P(a); P(b); P(c);

Which of the following access order by process B will not lead to deadlock ?
I) P(a); P(b); P(c);
II) P(a); P(c); P(b);
III) P(c); P(b); P(a);
IV) P(b); P(a); P(c);

A) I only
B) I and II only
C) I,II and III only
D) All I, II , III and IV.

Solution (B)
A : P(a); P(b); P(c); B : P(c); P(b); P(a); Deadlock is possible on b and c.
A : P(a); P(b); P(c); B : P(b); P(a); P(c); Deadlock is possible on b and a.
I and II access on semaphore a will block the other process , hence deadlock free.

12) Consider the following pseudo-code for a process Pi, where “shared boolean flag[2]” is a variable declared in shared memory, initialized as: flag[0] = flag[1] = FALSE;

```
// i=0 for P0, i=1 for P1
P (int i) {
 while (TRUE) {
   flag[i] = TRUE;
    while (flag[(i+1) % 2) == TRUE);
          < Critical Section >
    flag[i] = FALSE;
 < Remainder Section >
 }
}
```

Which of the following is FALSE?
A) Mutual exclusion is achieved.
B) Progress is achieved.
C) The above code uses the spin-lock mechanism.
D) All of them.

Solution : (B)
This solution may result in failure of progress, where neither P0 or P1 can proceed to enter their critical section. This will happen when P0 makes flag[0] = TRUE, and then there is a context switch. P1 makes the flag[1] = TRUE. Now both P0 and P1 will be waiting indefinitely in the while loop before entry to the critical section.
Rest of the statements are true, as the mutual exclusion is achieved. The code uses spin-lock mechanism to check the lock availability. (while (flag[(i+1) % 2) == TRUE);)

13) Consider the following code in which item=0, A and B are shared variables. Pick the correct choice from the given options.

Process A	Process B

1. int A = 1;
2. while(1){
3. if (B == 0) {
4. if (item == 0) {
5. item=item+1;
6. }
7. }
8. A = 0;
9. }
	1. int B = 1;
	2. While (1) {
	3. if (A == 0) {
	4. if (item == 0) {
	5. item=item+1;
	6. }
	7. }
	8. B = 0;
	9. }

A) The above code ensures mutual exclusion on shared variable item.
B) The above code does not ensure mutual exclusion on shared variable item.
C) The above code ensures bounded waiting.
D) None of the above.

Solution : (B)
The mutual exclusion is violated as both the processes enter the critical section and race condition can occur on shared variable item. One sequence is shown below:
Process A: A=1, while(1) { //preempt.
Process B: B=1, while(1) { //preempt.
Process A: if(B==0) [it evaluates to false] //preempt.
Process B: if(A==0) [it evaluates to false], Jumps to line 8 makes B=0; while(1){ //preempt.
Process A: Jumps to line 8 makes A=0; while(1){ if(B==0); if(item==0){ [it evaluates to true] Enters line5, //preempt.
Process B: if(B==0); if(item==0){ [it evaluates to true] enters line 5.
Hence, both Processes A and B are about execute line 5 leading to race condition on variable item.

Bounded waiting also fails as any of the process can repeatedly enter the critical section.

14) Consider the following code for process synchronization using two binary semaphores S1 and S2, both initialized to 1.

Process P1	Process P2
P(S1); P(S2); //Critical Section ……	P(S2); P(S1); //Critical Section ……..

The probability that the above synchronization code leads to a deadlock is _____. **[NAT]**

Solution: 0.66-0.67
The above two processes can access the semaphores in 6 different possible sequences.
(1) P1: P(S1),P(S2) ...| P2: P(S2),….
(2) P2: P(S2),P(S1) ...| P1: P(S1)…..
(3) P1: P(S1) | P2:P(S2) | P1:P(S2) (Blocked) | P2: P(S1) (Blocked)
(4) P2: P(S2) | P1:P(S1) | P2:P(S1) (Blocked) | P1: P(S2) (Blocked)
(5) P1: P(S1) | P2:P(S2) | P2: P(S1) (Blocked) | P1:P(S2) (Blocked)
(6) P2: P(S2) | P1:P(S1) | P1: P(S2) (Blocked) | P2:P(S1) (Blocked)
Hence, out of six possible sequences, four sequences lead to deadlock = 4/6 =0.66

15) When a process does not get access to the resource, it loops continually for the resource and wastes CPU cycles. It is known as

(A) deadlock
(B) spinlock
(C) livelock
(D) none

Solution (B) Spinlock is a mechanism to check the availability of the lock, where the process

loops continually for the lock and wastes CPU cycles

16) The operations that cannot be overlapped or interleaved with execution of any other operations are known as

(A) atomic operations
(B) messages
(C) system calls
(D) none

Solution (A)

16) In a _____, the process that locks the Critical Section will only unlock it.

(A) binary semaphore
(B) mutex
(C) counting semaphore
(D) none

Solution (B)
There is one more type of binary semaphore known as mutex, that is, in a binary semaphore, the CS locked by a process may be unlocked by any other process. However, in mutex, only the process that locks the CS can unlock it.

17) The following function (foo) is called by multiple processes (potentially concurrently). Identify the critical section(s) that require(s) mutual exclusion.

```
Line 1:   int i;

Line 2:   void foo()
Line 3:   {
Line 4:      int j;

Line 5:      /* Code Section A*/

Line 6:      i = i + 1;
Line 7:      j = j + 1;

Line 8:      /* Code Section B */
Line 9:   }
```

A) Line 6 and Line 7
B) Line 6 only
C) Line 7 only
D) Line 5,6, 7 and 8 .

Solution B)
There is no race condition on j, since it is a local variable. However, i is a variable shared between processes. Thus i = i + 1 (Line 6) would form a critical section .
18) Consider the following synchronization code using binary semaphores:

semaphore *mutex =1, *data = 1;	
Process P1	Process P2
P(mutex); /* do something */ P(data); /* do something else */ V(mutex); /* clean up */ V(data);	P(data) P(mutex); /* do something */ V(data); V(mutex);

The above code :

A) Does not ensure mutual exclusion

B) Free from deadlock

C) None of these.

D) The above code is not deadlock free.

Solution D)

The numbers inserted on the left indicate an execution order that results in deadlock.

semaphore *mutex =1, *data = 1;	
Process P1	Process P2
1: P(mutex); /* do something */ 4: P(data); /* Deadlocked here */ V(mutex); /* clean up */ V(data);	 2: P(data) 3: P(mutex); /* Deadlocked here */ V(data); V(mutex);

19) Consider the following synchronization code using binary semaphores:

semaphore *mutex =1, *data = 1;	
Process P1	Process P2
P(mutex); /* do something */ P(data); /* do something else */ V(mutex); /* clean up */ V(data);	P(mutex); P(data) /* do something */ V(data); V(mutex);

The above code :

A) Does not ensure mutual exclusion

B) Free from deadlock

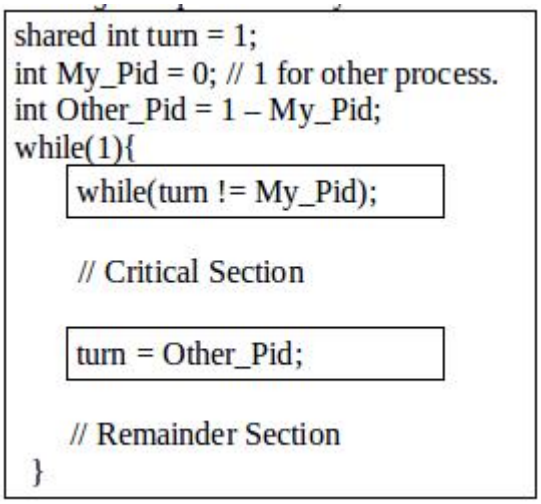
C) None of these

D) The above code is not deadlock free.

Solution B)

The above code ensures mutual exclusion and prevents deadlocks as the code ensures that the semaphores are acquired in the same order.

20) Consider the following two processes synchronization code



Which of the following is TRUE?

A) The above code fails to achieve the progress requirement.
B) The above code fails to achieve Mutual Exclusion.
C) The above code fails to achieve Bounded Waiting.
D) The above code may lead to Deadlock.

Solution (A)
Mutual exclusion is achieved as turn variable can ultimately take one value (0 or 1).
Deadlock is not possible as hold and wait is not present.
Bounded waiting is achieved as the processes enter critical section in alternate fashion.
However, the above code fails progress requirement. A process exiting after critical section may not allow other process to enter critical section as the processes enter critical section in strict alternative fashion. Progress can only be achieved if both the processes enters the critical section equal number of times.

21) Consider the following two processes synchronization code :

```
shared boolean wants[2] = false;
int My_Pid = 0; // 1 for other process.
int Other_Pid = 1 – My_Pid;

while(1){
    wants[My_Pid] = true;
    while(wants[Other_Pid]);

    // Critical Section

    wants[My_Pid] = false;

    // Remainder Section
}
```

Which of the following is/are TRUE? [MSQ]

A) The above code achieves Mutual Exclusion.
B) The above code achieves Progress.
C) The above code employs Spin lock.
D) The above code may lead to starvation.

Solution (A) (C) (D)

Mutual exclusion is achieved as while(wants[Other_Pid]) can ultimately take one value (true or false).
Progress is not achieved as the wants[My_Pid]=True can be set by both the processes, and both gets stuck in while loop leaving the CS free. (Indefinite postponement of selection is failure of progress).
Hence, starvation can occur.
Spin lock is busy waiting mechanism that is employed here.

22) Identify the correct statement from the following :
A) The Test-and-Set Lock hardware synchronization mechanism suffers from spin-lock when a round-robin scheduling algorithm is used.
B) Peterson's solution to the mutual-exclusion problem works for non-preemptive scheduling.
C) An operating system that can disable interrupts can implement semaphores.
D) None of these.

Solution (C)
With round-robin scheduling TSL works. Sooner or later all the processes will run, and eventually leave their critical regions.
Peterson's solution certainly works with preemptive scheduling. In fact, it was designed for that case. When scheduling is non preemptive, it might fail. Consider the case in which turn is initially 0 but process 1 runs first. It will just loop forever and never release the CPU.
To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore. If it is doing a down and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore. If it is doing an up , it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.

23) You are designing a data structure for efficient dictionary lookup in a multithreaded application. The design uses a hash table that consists of an array of pointers each corresponding to a hash bin. The array has 1001 elements, and a hash function takes an item to be searched and computes an entry between 0 and 1000. The pointer at the computed entry is either null, in which case the item is not found, or it points to a doubly linked list of items that you would search sequentially to see if any of them matches the item you are searching for. There are three functions defined on the hash table: Insertion (if an item is not there already), Lookup (to see if an item is there), and deletion (to remove an item from the table). Considering the need for synchronization, then choose the appropriate solution:

A) Use a mutex over the entire table
B) Use a mutex over each hash bin
C) Use a mutex over each hash bin and a mutex over each element in the doubly linked list
D) None of these

Solution: (B)
A mutex over the entire table (solution 1) is undesirable since it would unnecessarily restrict concurrency. Such a design would only permit a single insert, lookup or delete operation to be outstanding at any given time, even if they are to different hash bins. A mutex over each element in the doubly linked list (solution 2) would permit the greatest concurrency, but a correct, deadlock-free implementation has to ensure that all elements involved in a delete or insert operation, namely, up to three elements for an delete, or two elements and the hash bin for inserts/some deletes, are acquired in a well-defined order. A mutex over each hash bin is a compromise between these two solutions it permits more concurrency than solution 1, and is easier to implement correctly than solution 2.

24) Which of the following are TRUE regarding the priority inversion problem? **[MSQ]**

A) Priority inversion may occur from resource synchronization among processes of different priorities.
B) Test-andSet lock suffers from priority inversion problems.
C) Priority inversion may lead to starvation.
D) Priority inversion problem may occur in any preemptive scheduling algorithms.

Solution: (A) (B) (C)
Priority inversion may occur from resource synchronization among processes of different priorities in a preemptive priority scheduling. Priority inversion may lead to starvation of a high priority process. Example of synchronization mechanism that may suffer from this issues is TSL.

25) Consider the following code using mutex M1=1,M2=1, and M3=1.

Process 1	Process 2	Process 3

while (1) {
NonCriticalSection()
Mutex_lock(&M1);
Mutex_lock(&M2);
CriticalSection();
Mutex_unlock(&M2);
Mutex_unlock(&M1);
}
	while (1) {
	NonCriticalSection()
	Mutex_lock(&M2);
	Mutex_lock(&M3);
	CriticalSection();
	Mutex_unlock(&M3);
	Mutex_unlock(&M2);
	}
		while (1) {
		NonCriticalSection()
		Mutex_lock(&M3);
		Mutex_lock(&M1);
		CriticalSection();
		Mutex_unlock(&M1);
		Mutex_unlock(&M3);
		}

Which of the following are correct ? **[MSQ]**

A) Mutual exclusion is satisfied.
B) Deadlock is possible only when all three resources (M1, M2 and M3) are held by processes.
C) Deadlock is not possible.
D) Deadlock is possible if only two resources out of M1, M2 and M3 are held by processes.

Solution: (A) (B)

Mutual exclusion is satisfied as one mutex is common between any two processes. Deadlock involves all three resources and three resources.

26) Which of the following are TRUE? **[MSQ]**

A) Lock variable synchronization mechanism ensures mutual exclusion.
B) Strict alternation synchronization mechanism ensures progress.
C) Peterson's Solution synchronization mechanism ensures bounded waiting.
D) Test-and-Set lock synchronization mechanism may suffer from deadlock.

Solution: (C)
Lock variable : Mutual exclusion is not satisfied.
Strict alternation: Satisfies bounded waiting but no progress.
Peterson's solution: Is correct implementation, which satisfies all the necessary properties.
Test-and-Set lock synchronization mechanism may suffer from livelock but no deadlock.

27) Consider the following code, which is shared by four concurrent processes.

```
int count = 0, n=10; //Shared among all processes

int main()
{
  int i; // local to each process
  for(i=1;i<=n;i++){
    count=count+1;
  }
return 0;
}
```

The minimum value of count after completion of execution is
A) 1
B) 2
C) 10
D) 40

Solution: (B)
Given count=0, n=10.
Let us assume the assembly code for count=count+1 as

Read R0,count
Add R0,1
Write R0,Count

Note: The registers are local to each process. During context switch (preemption) the register contents are saved and reloaded. So, race condition on registers are not possible.

Process P1: Read R0,count
Add R0,1 // Preempt. So R0 contains 0.

Process P2: execute loop 10 times, end. Count value in memory is 10. [Complete]
Process P3: execute loop 10 times, end. Count value in memory is 20. [Complete]
Process P4: execute loop 9 times. Count value in memory is 29. Preempted.
Process P1: Write R0,Count, so count value in memory becomes 1. Preempted.
Process P4: Enters last iteration execute Read R0,count, so R0 has value 1. Preempted.
Process P1: execute loop 9 times. Count value in memory is 10.[Complete]
Process P4: execute Add R0,1 and Write R0,Count. So, the Count value in memory is 2.

[Complete].

28) Consider the Compare-and-Swap synchronization mechanism to implement mutual exclusion on a critical section. Compare_and_Swap(&lock, oldval, newval) is atomic and writes newval into var and returns true if the old value of var is oldval. If the old value of var is not oldval, Compare-and-Swap returns false and does not change the value of the variable. Assume that the lock is initialized to 0.

```
int Compare-and-Swap(int *var,int oldval,int newval) {
    int temp = *var;
    if(*var == oldvalue)
       *var = newvalue
    return temp;
}
```

```
Entry{
  P:__________
}

//Critical Section

Exit{
 lock=0;
}
```

Choose the correct predicate P in the Entry section, to achieve mutual exclusion.
A) while(Compare-and-Swap(&lock, 1, 1)!=0);
B) while(Compare-and-Swap(&lock, 1, 0)!=0);
C) while(Compare-and-Swap(&lock, 0, 1)!=0);
D) None of these.

Solution : (C)

```
Entry{
  while( Compare-and-Swap(&lock, 0, 1)!=0);
}

//Critical Section

Exit{
 lock=0;
}
```

CAS is a technique used to obtain synchronization during multiple writes where each write value depends upon the current state of the shared variable. It basically means that a variable will first be compared with a value to see if it has changed. If it has changed, then it means its value has been updated by some other thread and hence swap is not possible. In such a case, the current value of the shared variable is obtained and a new value is calculated from it. If it has not changed, then it means that its value has not been modified by any other thread and so it can be swapped.

29) Which of the following is not a disadvantage of disabling interrupts to serialize access to a critical section

A) User code cannot utilize this technique for serializing access to critical sections

B) Interrupt controllers have a limited number of physical interrupt lines, thereby making it problematic to allocate them exclusively to critical sections.

C) This technique would lock out other hardware interrupts, potentially causing critical events to be missed.

D) This technique could not be used to enforce a critical section on a multiprocessor

Solution (B)

Number of interrupts does not matter as all the interrupts are disabled during a process when it enters critical section.

30) What is the output produced by the following two processes P and Q using a muutex ?

P	Q

While(1){
P(mutex);
print 1;
V(mutex);
}
	While(1){
	P(mutex);
	print 2;
	V(mutex);
	}

A) 1*2*

B) (12)*

C) (21)*

D) (1|2)*

Solution (D)

When mutex = 0, then nothing is printed.

When mutex = 1, then 1 or 2 can be printed in any order.

Combining both cases we get (1|2)*

31) Consider the following two concurrent processes

int d = 0, r = 0;	

void p1 () {
  d = 2;
  r = 1;
}
	int p2 () {
	  x=d;
	  while (!r) { }
	  return x;
	}

What are the possible return values of p2()?
A) 0, only.
B) 2, only
C) 0 or 2.
D) None of these.

Solution (C)
The d= 2 is returned when both the processes are executed serially i.e., P1 followed by P2. Also, 0 can be returned if p2 executes x=d, and then p1 sets r =1, then x contains the old value of d (0) which is then returned.

32) Consider the following synchronization code using binary semaphore between two processes

Int x=1, y=1 Semaphore mx = 1;my=0;	

P(mx);
  x = x+1
V(my);
	P(my);
	  x = y+1
	V(mx);

The value of x and y after execution is
A) infinite, 1
B) 2,1
C) 1,1
D) 3,1

Solution (B) The above is similar to sequential code x=x+1; x=y+1. Therefore, x=2, y=1 after execution. Note that the code is run only once as no loop is given. Synchornization

33)

1) Does starvation-freedom imply deadlock-freedom?

Yes! If every process can eventually enter its critical section, although waiting time may vary, it means the decision time of selecting a process is finite. Otherwise, all processes would wait in the entry section.

2) Does starvation-freedom imply bounded-waiting?

No! This is because the waiting time may not be bounded even though each process can enter its critical section.

3) Does bounded-waiting imply starvation-freedom?

No! Bounded-Waiting does not say if a process can actually enter. It only says there is a bound. For example, all processes are locked up in the entry section (i.e., failure of Progress).

4) We need Progress and Bounded-Waiting both to imply Deadlock-Freedom

Yes! progress along with bounded waiting makes starvation-free, and starvation-free implies deadlock free.

5) Progress does not imply Bounded Waiting:

Yes! Progress says a process can enter with a finite decision time. It does not say which process can enter, and there is no guarantee for bounded waiting.

6) Bounded Waiting does not imply Progress:

Yes! Even though we have a bound, all processes may be locked up in the entry section (failure of Progress).