Tutorial-6

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:

   I. The above solution to CS problem is deadlock-free
   II. The solution is starvation free.
   III. The processes enter CS in FIFO order.
   IV More than one process can enter CS at the same time.

   Which of the above statements is TRUE?
   (a) I only
   (b) I and II
   (c) II and III
   (d) IV only

   Ans: option(a)
   Explanation:
   The test-and-set instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation. Since it is an atomic instruction it guarantees mutual exclusion.

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

(a) At least twice (b) Exactly twice
(c) Exactly thrice (d) Exactly once

Ans: option (a)
Explanation:
P0 will execute first because only S0=1. Hence it will print 0 (for the first time). Also P0 releases S1 and S2. Since S1=1 and S2=1, therefore P1 or P2, any one of them can be executed.
Let us assume that P1 executes and releases S0 (Now value of S0 = 1). Note that P1 process is completed.
Now S0=1 and S2=1, hence either P0 can execute or P2 can execute. Let us check both the conditions:-

Let us assume that P2 executes, and releases S0 and completes its execution. Now P0 executes; S0=0 and prints 0 (i.e. second 0). And then releases S1 and S2. But note that P1 and P2 processes has already finished their execution. Again if P0 tries to execute it goes into sleep condition because S0=0. Therefore, minimum number of times '0' gets printed is 2.

Now, let us assume that P0 executes. Hence S0=0, (due to wait(S0)), and it will print 0 (second 0) and releases S1 and S2. Now only P2 can execute, because P1 has already completed its execution and P0 cannot execute because S0 = 0. Now P2 executes and releases S0 (i.e. S0=1) and finishes its execution. Now P0 starts its execution and again prints 0 (thrid 0) and releases S1 and S2 (Note that now S0=0). P1 and P2 has already completed its execution therefore again P1 takes its turn, but since S0=0, it goes into sleep condition. And the processes P1 and P2 which could wakeup P0 has already finished their execution.Therefore, maximum number of times '0' gets printed is 2.

3. 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 pseudocode 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

Solution:

Ans: option (b)
Explanation:
Assume that L=0, that means the lock is now available. A process P1 wants to acquire the lock by executing the AcquireLock() function. We can see that the while loop fails because the Fetch_And_Add instruction returns the previous value of L, which was 0. (Note that after the execution of the atomic instruction, the present value of L is now 1). Since the returned value was 0, the while loop fails and P1 process comes out of the while loop and hence acquires the lock.
Now scheduler schedules another process P2 and context switching takes place. P2 tries to acquire the lock by executing the AcquireLock() function. But it goes on executing the while loop infinitely (because the atomic instruction always returns a non-zero value and the while loop condition is always true).
After some time, scheduler again schedules P1. We assume that P2 was stopped soon after the Fetch_And_Add() instruction returned the value. Hence L has some non-zero value. Now P1 releases the lock L. That means now L=0. Assume that again context switch takes place and P2 arrives.
We have assumed that P2 was switched out when it executed the Fetch_And_Add instruction and a non-zero value has been returned. Since the returned value was non-zero the condition becomes true and the statement L=1 is executed. Hence now L=1. Again the while loop is executed, Fetch_And_Add instruction will return value 1 and hence again the while loop enters into infinite loop. Now none of the process is able to acquire the lock. Therefore the above implementation fails.

4. Two processes, P1 and P2, need to access a critical section of code. Consider the following synchronization construct used by the processes:

/* P1 */	/* P2 */
while (true) {	while (true) {
wants1 = true;	wants2 = true;
while (wants2 == true);	while (wants1==true);
/* Critical	/* Critical
Section */	Section */
wants1=false;	wants2 = false;
}	}
/* Remainder section */	/* Remainder section */

Here, wants1 and wants2 are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct?
(a) It does not ensure mutual exclusion.
(b) It does not ensure bounded waiting.
(c) It requires that processes enter the critical section in strict alternation.
(d) It does not prevent deadlocks but ensures mutual exclusion.

Solution:

Ans: option (d)
Explanation:
The code ensures the condition of mutual exclusion: Assume P1 is initiated. It sets wants1=true. Now since wants2 = false, P1 exists from its while loop and enters its

critical section. Now suppose context switch takes place and P2 gets executed. Now it sets wants2=true, and now enters the while loop and remains busy till P1 comes out of the critical section and sets wants1=false, because wants1= true (as set by P1). So we can see that the mutual exclusion condition is satisfied.

The code does not prevent deadlock: Assume that P1 starts its execution. It sets wants1=true and then gets preempted. Now P2 starts its execution. P2 sets wants2=true and suddenly gets preempted. Now P1 starts execution; it enters the while loop and finds that wants2=true and remains busy in the while loop. Now P1 gets preempted. P2 enters execution; it enters the while loop and finds that wants1=true remains busy in the while loop. Hence both P1 and P2 remains busy forever.

5. The atomic fetch-and-set x, y instruction unconditionally sets the memory location x to 1 and fetches the old value of x in y without allowing any intervening access to the memory location x. consider the following implementation of P and V functions on a binary semaphore.

```
void P (binary_semaphore *s) {
 unsigned y;
 unsigned *x = &(s->value);
do {
        fetch-and-set x, y;
} while (y);
}
void V (binary_semaphore *s) {
 S->value = 0;
}
```

Which one of the following is true?
(a) The implementation may not work if context switching is disabled in P.
(b) Instead of using fetch-and-set, a pair of normal load/store can be used
(c) The implementation of V is wrong
(d) The code does not implement a binary semaphore

Solution:

Ans: option (a)
Explanation:
Assume that P1 process enters the P function. Initially if the value of semaphore is 1, according to the definition of the fetch-and-set instruction, the value of y will be 1 when the fetch-and-set instruction is executed. Hence the while loop goes on executing until other processes execute the V function. But if context switching is disabled in P while loop will keep on executing and other processes are not allowed to execute.