Operating Systems and Systems Programming (15B11CI412)

Tutorial - 5

Process Synchronization

Q1. Define the meaning of a race condition? Use an execution sequence to illustrate your answer.

A race condition is a problem that can occur in a multithreaded program. Suppose that a thread takes a sequence of actions in which one action can depend on the result of a previous action. A race condition occurs if it's possible for another thread to change or invalidate the result of the previous action, before the first thread completes the sequence. For example, there is a race condition in the simple assignment statement count = count+1 because it is executed as a sequence of steps. The old value of count is read, one is added to that value, and the new value is stored back into count. A race condition occurs if it is possible for another thread to increment the value of count between the time when the first thread reads the old value and the time when it stores the new value. In this case, the race condition can result in an incorrect value for count -- count might be increased by one when it was supposed to be increased by two. Another example occurs in the if statement

```
if ( ! list.isEmpty() )
        return list.removeFirst();
```

There is a race condition if it is possible for another thread to empty the list between the time when the first thread tests whether the list is empty and the time when the first thread tries to remove an element from the list. In this case, the race condition can result in an exception when the first thread tries to remove an item from an empty list.

Q2. A good solution to the critical section problem must satisfy three conditions: mutual exclusion, progress and bounded waiting. Explain the meaning of the progress condition. Does starvation violate the progress condition?

If no process is executing in its critical section and some processes are waiting to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next (i.e., only those wanted to enter can join the competition and none of the other processes can have any influence), and this selection cannot be postponed indefinitely. The progress condition only guarantees the decision of selecting one process to enter a critical section will not be postponed indefinitely. It does not mean a waiting process will enter its critical section eventually. Therefore, starvation is not a violation of the progress condition. Instead, starvation violates the bounded waiting condition.

Q3. Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.

Throughput in the readers-writers problem is increased by favouring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favouring readers could result in starvation for writers. The starvation in the readers-writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration. When a

reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

Q4. What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

Q5. Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user level programs.

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

Q6. Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

Q7. Demonstrate that monitors and semaphores are equivalent in so far as they can be used to implement the same types of synchronization problems

A semaphore can be implemented using the following monitor code:

```
monitor semaphore {
    int value = 0;
    condition c;

    semaphore_increment() {
       value++;
       c.signal();
    }

    semaphore_decrement() {
       while (value == 0)
       c.wait();
       value--;
    }
}
```

A monitor could be implemented using a semaphore in the following manner. Each condition variable is represented by a queue of threads waiting for the condition. Each thread has a semaphore associated with its queue entry. When a thread performs await operation, it creates a new semaphore (initialized to zero), appends the semaphore to the queue associated with the condition variable, and performs a blocking semaphore decrement operation on the newly created semaphore. When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.

Q9. Write a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock, which invokes a procedure *tick* in your monitor at regular intervals.

```
monitor AlarmClock

int now=0;
condition wakeup;
wakeme (int n)

{

        int alarm;
        alarm = now + n;
        while (now < alarm) wakeup.wait (alarm);
        wakeup.signal;

}

tick()

{

        now = now + 1;
        wakeup.signal;

}
```

Q10. Consider three concurrently executing threads in the same process using two semaphores s1 and s2. Assume s1 has been initialized to 1, while s2 has been initialized to 0.
What are the possible values of the global variable x, initialized to 0, after all three threads have terminated?

```
/* thread A */
P(&s2);
P(&s1);
x = x*2;
V(&s1);
/* thread B */
P(&s1);
x = x*x;
V(&s1);
/* thread C */
P(&s1);
x = x+3;
V(&s2);
V(&s1);
```

The possible sequences are B, C, A (x = 6) or C, A, B (x = 36) or C, B, A (x = 18).

Q 11. The following pair of processes share a common variable X:

Process A	Process B
int Y;	int Z;
A1: Y = X*2;	B1: Z = X+1;
A2: X = Y;	B2: X = Z;

X is set to 5 before either process begins execution. As usual, statements within a process are executed sequentially, but statements in process A may execute in any order with respect to statements in process B.

A. How many different values of X are possible after both processes finish executing?

Here are the possible ways in which statements from A and B can be interleaved.
A1 A2 B1 B2: X = 11
A1 B1 A2 B2: X = 6
A1 B1 B2 A2: X = 10
B1 A1 B2 A2: X = 10
B1 A1 A2 B2: X = 6
B1 B2 A1 A2: X = 12.

B. Finally, suppose the programs are modified as follows to use a shared binary semaphore T:

Process A	Process B
int Y;	int Z;
A1: Y = X*2;	B1: wait(T);
A2: X = Y;	B2: Z = X+1;
signal(T);	X = Z;

T is set to 0 before either process begins execution and, as before, X is set to 5. Now, how many different values of X are possible after both processes finish executing?

The semaphore T ensures that all the statements from A finish execution before B begins. So now there is only one way in which statements from A and B can be interleaved:
A1 A2 B1 B2: X = 11.