SC2005 Operating Systems

**TUTORIAL FOUR**

**Process Synchronization- Part I**

1a)
1. Mutual Exclusion: When a process is executing in its critical section, no other process can be executing in its critical section at the same time
2. Progress: When no process is executing in its critical section and no other process is waiting to access the critical section, selection of the next process to enter the critical section cannot be postponed indefinitely
3. Bounded waiting: After a process has requested to enter its critical section and before that request is granted, other processes are allowed to enter their critical section only a bounded number of times

1.

a) Describe the three key requirements that must be satisfied by any solution to the critical section problem.

b) Consider the following user-level solution to the critical section problem, where **flag** and **turn** are shared variables initialized to **false** and **0**, respectively.

while(1): wait
while(0): move on

Bounded waiting. When flag is false and turn is zero, while loop of P0 not executed, so context switch to P1, which executes entirely and goes back to allow P0 to execute. However, the turn is not updated and is still = 0. P1 is then executed again after a context switch.

Process P0

```
while(1){
   flag=true;
   while(turn==1);
   critical-section
   turn=1;
   remainder-section
}
```

If P0 executes its while loop and turn becomes 1, the while loop will loop forever and progress is not satisfied

Process P1

```
while(1){
   turn=0;
   while(flag and turn==0);
   critical-section
   flag=false;
   remainder-section
}
```

context switch from P1 to P0, both P0 and P1 in critical section

Determine which of the three requirements in part (a) are not satisfied. Justify your answer.

1. P0: waiting @ while(turn==1); P0 will enter the moment P1 executes the first statement inside its while loop. P1 can enter its critical section 0 times after P0 has requested access

2. Indicate whether the following statements are true or false. Justify your answers.

a) Race condition only occurs because a single high-level C instruction (e.g., counter++;) is translated into multiple low-level assembly instructions (e.g., register=counter; register=register+1; counter=register).

b) Mutual exclusion can be achieved by disabling interrupts during the critical section.

c) If a solution to a critical section problem satisfies progress, then it also satisfies bounded waiting.

P1: waiting @ while(flag and turn ==0); P1 will enter the moment P0 executes turn =1 statement after its critical section. So P0 can enter its criitical section 1 time after P1 has requested access.

3. Consider a computer that does not have a *TestAndSet* instruction, but has an instruction to ***swap*** the contents of a register and memory word in a single atomic command. Show how it can be used to implement the *entry section* and *exit section* which are before and after the critical section.

2a)
False. If there are two threads reading counter value and updating it, the value of one thread might overwrite the other

Race condition also occurs in producer consumer example if increment to counter is done using a temp variable

2b) True. By disabling interrupts, no other process can execute the critical section while the current process is executing its critical section.

```
while(1){
while(register == memory word){
critical section; }
swap(register, mem word);
remainder section
}
```

-ensures that the mem is updated after every critical section

boolean lock is a shared memory variable and initialized to false

favours process P0 over process P1,flag false

2c) False. This is seen in TestAndSet which uses a shared boolean lock. If lock is false, the first process that executes TestAndSet(&lock) will immediately enter critical section, showing progess. However, if the first process context switches at the critical section, it still holds the lock such that the second process cannot enter its critical section.

```
P0
while(flag);
flag = true;
critical section;
flag = false;
```

```
P1
while(flag){wait(timeout);}
flag = true;
critical section;
flag = false;
```

```
entry section:
      register = true;
      while(register){
            swap(lock,register);
      }

exit section:
      lock: = false;
```