Critical Section

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

Critical Section

- The three key requirements are as follows:
  1. **Mutual Exclusion**. No more than one process can be executing in its critical section.
  2. **Progress**. The selection of the processes that will enter the critical section next cannot be postponed indefinitely.
  3. **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

1 (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.

<u>Process P0</u>

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

<u>Process P1</u>

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

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

Critical Section (1 (b))

- This solution does not guarantee mutual exclusion. Consider the following sequence:

1. P1: turn=0;
2. P1: while(flag and turn==0);
3. P1: critical-section
4. **Context switch from P1 to P0**
5. P0: flag=true;
6. P0: while(turn==1);
7. critical-section

→ **P0 and P1 are both in the critical section** ☹

Critical Section (1 (b))

- This solution does not guarantee progress. Consider the following sequence:

1. **P0: executes the while loop once and turn becomes 1**
2. P0: flag=true;
3. P0: while(turn==1); // loop forever

→ **Progress is not satisfied for process P0 ☹**

Critical Section (1 (b))

- This solution satisfies bounded waiting.

1. **P0: waiting @ while(turn==1);**

In this case, PO will enter the moment P1 executes the first statement inside its while loop. So P1 can enter its critical section 0 times after P0 has requested access.

2. **P1: waiting @ while(flag and turn==0);**

In this case, P1 will enter the moment P0 executes turn=1 statement after its critical section. So P0 can enter its critical section 1 time after P1 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.

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).

→ False.

Justification: Race condition can also occur in the producer-consumer example of the lecture if the increment to **counter** is done using a **temp** variable

temp = counter;

temp = temp+1;

counter = temp;

In this case, race condition occurs irrespective of how these high-level instructions are translated.

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

→True.

Justification: By disabling interrupts, no other process, including the operating system, will be able to run during the critical section.

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

→False.

Justification: A solution that always favors a process P0 over another process P1 can satisfy progress without satisfying bounded waiting. In this case, the bounded waiting requirement for process P1 will be violated. Consider the following, with **flag** initialized to **false** and **timeout** some large value.

Process P0	Process P1
while(flag);	while(flag) {wait(timeout);}
flag=true;	flag=true;
critical section;	critical section;
flag=false;	flag=false;

TestAndSet

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.

swap (boolean lock, register s);

while (1) {

entry section ?

critical section

exit section ?

remainder section

}

Swap

- boolean **lock** is a shared memory variable and initialized to false.

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

exit section:
    lock: = *false*;