CG2271 Real Time Operating Systems
Tutorial 8

Question 1

Describe what a race condition is.

A race condition is a condition where the final outcome of several parallel threads updating shared memory depends on which thread completes updating first. This generally leads to incorrect solutions, i.e. solutions that are different from one where the threads run sequentially.

Suppose two identical tasks update a shared variable *tmp* using this instruction:

```
tmp++;
```

Can a race condition occur? Why or why not?

Yes. The reason is that this update is typically broken up into multiple machine instructions, e.g.:

```
LD R1, tmp      ; Load tmp into R1
ADD R1, R1, 1   ; Increment R1
SW R1, tmp      ; Store incremented R1 into tmp
```

If a thread gets pre-empted before it completely writes back to tmp, race conditions will occur. Even if tmp++ is implemented using one single instruction, e.g.:

```
ADD tmp, tmp, 1         ; Add 1 to tmp, directly in memory.
```

In a multiple CPU system, another CPU may gain control of the shared memory bus first and incorrectly update tmp.

Question 2

Using Google, the textbook or any other resource, describe what "atomicity" means. The kernel often disables interrupts to guarantee atomicity. Explain how this works. Explain further why it's a bad idea to allow user processes to achieve atomicity in the same way.

Atomicity means that the steps taken in an instruction, or the steps taken in a group of instructions, are executed as one complete unit without possibility of interruption. In our example in Q4:

```
LD R1, tmp      ; Load tmp into R1
ADD R1, R1, 1   ; Increment R1
SW R1, tmp      ; Store incremented R1 into tmp
```

Atomicity means that all 3 instructions are executed completely without possibility of the thread being interrupted in between.

Single-processor operating systems may disable interrupts to guarantee atomicity because task/process/thread switching does not happen without interrupts. However, OSes are written and tested thoroughly before being released (e.g. uCOS/II is tested to Do178b compliance, which means it is safe to use this OS on passenger aicraft), so there is less chance of an OS disabling interrupts and causing the entire system to shut down completely.

On the other hand user programs may not be as thoroughly tested, and allowing user programs to disable interrupts could lead to the entire system being crippled, as no further task switches would be possible.

Question 3

For each of the following mechanisms:

i) Is the mechanism is effective in enforcing mutal exclusions, and why or why not.

ii) If the mechanism is able to enforce mutual exclusion, list and explain the advantages and disadvantages of the mechanism.

1) Disabling Interrupts.
2) Lock variables.
3) Strict alternation.
4) Peterson's Solution.
5) Sleep/wakeup.

SEE TABLE AT END

Question 4

The condition variables in monitors work in a very similar way to "sleep" and "wake".

a. Explain how condition variables are the same and how they are different from "sleep" and "wake".

Same: Both "sleep" and "wait" allow a process to voluntarily give up CPU time. Both "wake" and "signal" allow another process to wake the sleeping process.

Different: "wake" requires a specific process ID to wake up, whereas "signal" wakes up any process that is sleeping on a condition variable (note: it is possible that only 1-of-n processes is woken, or all n). "wait" and "signal" are used only within the mutex provided by a monitor.

b. The "sleep" and "wake" operations are known to cause deadlock, resulting in the producer/consumer problem. If a buffer is implemented using monitors and condition variables, can the producer/consumer problem still occur? Why or why not?

No. The key problem with sleep/wake is that the consumer sees an empty buffer (or producer sees a full buffer), but gets pre-empted before going to sleep. Producer inserts one item into the buffer, and after incrementing, sees that count==1, meaning that previously the buffer was empty and there's possibly a consumer asleep, and sends a wake. However since the consumer hasn't "slept" yet, the wake is lost. When the consumer resumes, it goes to sleep and never consumes anything from the buffer. Eventually the buffer fills completely and the producer also sleeps.

Similar agument applies for the case where a producer sees a fully buffer and sleeps.

This cannot happen in a monitor:

Consumer sees an empty buffer, and gets pre-empted.
Producer starts up, but is unable to enter the monitor since Consumer is still there. Control is handed back to Consumer.
Consumer waits on "empty" condition variable. Control is handed back to producer.
Producer is now able to enter the semaphore, insert an item and send a signal to the waiting consumer.

We can see that in this case because the producer is unable to enter the monitor before the consumer does a wait, the signal does not get lost.

Mechanism	Effective?	Why/Why not?	Advantages	Disadvantages
Disabling interrupts	Y	Without interrupts, there will be no process switching, hence only one process is running. However this will not work in a multi-processor environment since disabling interrupts only affects one CPU.	Simple to implement IF you have kernel privileges to disable interrupts.	Only works in single-processor environments. If a program disables interrupts and hangs, the entire system will no longer work since it cannot switch tasks.
Lock variables	N	Race condition on the lock variable.	-	-
Strict alternation	Y	Processes update the "turn" variable to allow the OTHER process to run. A race condition therefore does not affect the algorithm. Suppose this is a time-slicing OS that gives each process a certain amount of CPU time then switches to another. Process 1 is currently in the critical section (turn=1), and the OS switches to process 0. Towards the end of process 0's turn, this can happen: Process 0: Reads in turn=1, gets pre-empted. Process 1: finishes critical section, sets turn=0. Gets pre-empted or voluntarily gives up CPU time. Process 0: Sees that turn=1, loops, reads in turn=0, enters critical section.	Simple to implement.	One task not in its critical section can block out another task that wants to enter the critical section (see notes). Busy waiting.

		We can see that the worst case that happens in this race condition is that process 0 loops one extra time. However the mutex remains protected.		
Peterson's Solution	Y	Consider process 0 wanting to enter the critical section, with neither process currently in there. **Process 0:** interested[0] is set to 1. turn is set to 0 Loads in interested[1]. Since neither process is in the CS, interested[1]=0, however process 0 is pre-empted. **Process 1:** interested[1] is set to 1. turn is set to 1 Loads in interested[0], sees it is 1, loops until it is pre-empted. **Process 0:** "Knows" from earlier on that interested[1]=0, so enters CS without looping. CS is secure.	Does not require hardware support for mutual exclusion (neither does strict alternation). Does not suffer from problem where a process in a non-critical section is able to lock out another process, since there is no implicit assumption of whose turn it is to go.	Busy waiting. Relatively difficult to understand.

Sleep/wakeup	N	This is a coordination mechanism to allow one task to sleep while a condition is not met, and another to wake it when a condition is met. There is no attempt to see if anyone is in the CS.	-	-