IT5002 Tutorial 7

Race Conditions

Question 1

We have the following two threads:

```
int x = 5; // Shared variable

void *thread1(void *p)
{
        while(1)
        {
                x++;
                .. Other lines of code ..
        }
}

void *thread2(void *p)
{
        while(1)
        {
                x*=2;
                .. Other lines of code ..
        }
}
```

a. Write down the x++ and x*=2 in assembly on a CPU with load-store architecture (i.e. ALU only operates on registers. You must first load variables in memory into registers. You must then write the registers back to the variables in memory) You can choose any kind of assembly language, even one that doesn't exist, as long as it is a load-store architecture.

Thread 1:

```
loop:
        lw $r0, x
        addi $r0, $r0, 1
        sw $r0, x
        jmp loop
```

Thread 2:

```
loop:
        lw $r0, x
        shl $r0, $r0, 1
        sw $r0, x
        jmp loop
```

b. Based on your answer to a., what are the final values of x after both thread1 and thread2 have run one iteration?

x=5

Thread 1 runs to completion, x=6
Thread 2 runs to completion, x = 12

Thread 2 runs to completion, x = 10
Thread 1 runs to completion, x = 11

Thread 1 loads x and is pre-empted
Thread 2 loads x and writes back 5 x 2 = 10
Thread 1 increments x to 6 and writes it back

Thread 2 loads x and is pre-empted
Thread 1 loads x, updates it to 6, writes back
Thread 2 gets 5 x 2 = 10

Possible values are 12, 11, 6 and 10

c. Research into the term "race condition". How does this program demonstrate race conditions?

A race condition is a condition where the outcome of a computation that is performed by multiple processes/threads depends on which process/thread finishes first.

d. Which of the answers in b. is correct? How do you define correctness in multithreaded programs?

Either 12 or 11 depending on the intended sequence of execution.

e. In general, in multithreaded programs shared variables are often updated correctly, and are sometimes updated wrongly. Why?

Either because the threads are executed in the wrong sequence, or because one thread gets pre-empted before it can save its results and the next thread gets a stale value. The first thread then overwrites the next thread's results.

Question 2

How are local variables created in C (GIYF)? Can local variables be affected by race conditions? Why or why not?

Local variables are created on the process's stack when a function is called. They're popped off and lost when the function exits.

They can never cause race conditions because only the thread or process calling that function can access to the variables. Since no other process or thread has access, race conditions are not possible.

Question 3

Co-operative multitaskers technically cannot suffer from race conditions. Why not? Despite this however, correctness of multithreaded processes in co-operative multitaskers is not guaranteed. Why not?

There are two reasons:

- The sequence of process execution might be wrong. The scheduler might choose to run process A before B, when A depends on a result in B.
- A process may inadvertently give up control of the CPU before completing calculations, causing a dependent process to run and get the wrong results. An example of this is when a process decides to call printf, which will trigger an OS call that might trigger a context switch without the programmer's knowledge.

Question 4

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */

void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 – process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

i) Explain how Peterson's Solution enforces mutual exclusion.

At first glance Peterson's Solution looks like a "lock variable" solution, but with some key differences:

- There are TWO different lock variables, one for each process.

- Each processes only WRITES to the lock variable, which is an atomic operation. The main weakness of the Lock Variable solution is that it involved read-update-write operations on a single shared variable, which does not happen in this case.
- Before entering the critical section, both processes check to see whether THE OTHER process intends to enter. If so it waits.
- Setting interested to TRUE takes place BEFORE this check, so in the worst case, both set interested to true at the same time, leading to deadlock, which brings us to:

ii) Explain how the "turn" variable in Peterson's Solution prevents deadlock.

Without "turn" in "enter_region" function, deadlock will occur.

P0	P1
other=1	
Interested[0]=1	
* pre-empted *	other=0;
	Interested[1]=1;
while(interested[1]==1); // Loops forever	* pre-empted *
…	
* pre-empted *	while(interested[0]==1); // Loops forever

Now both P0 and P1 are stuck in the while loops.

Adding in turn will prevent this, because turn will either be 0 or 1. When turn is 0, process 0 is stuck in the loop and process 1 will execute the critical section, and vice versa.