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.

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

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

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

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

Question 2

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

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?

Question 4

This is Peterson's Solution:

```
#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 guarantees that no two processes can enter their critical sections at the same time.

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