Tutorial 3 **CS3103 Operating Systems**

Student Name: _____

Student No. : | | | | | | | | |

Day: ☐ Monday ☐ Tuesday

Time: ☐ 10:00 - 10:50 ☐ 11:00 - 11:50 ☐ 12:00 - 12:50 ☐ 14:00 – 14:50 ☐ 18:00 - 18:50

The Threads

Introduction

Topics to be covered in this tutorial include:

- How different thread interleavings either cause or avoid race conditions.

Acknowledgement

This tutorial was adapted from OSTEP book written by Remzi and Andrea Arpaci-Dusseau at University of Wisconsin. This free OS book is available at http://www.ostep.org.

Getting Started

1. Logging in to the Linux server

- Start the SSH client, e.g., MobaXterm or Xshell.
- Log in to the Linux server using the following details:

  Host Name: gateway.cs.cityu.edu.hk

  User Name: your EID (e.g., cctom2)

  Password: your password

Your password will not be shown on the screen as you type it, not even as a row of stars (******).

NOTE: The shell will always give you a prompt if it is ready to accept commands. The shell prompt normally ends in a `$` sign as we use in this tutorial. Some shell prompts end in `%` or `>` instead. Never copy/type the shell prompt used in this tutorial.

NOTE: Please don't forget to log out (use the `exit` command) after you finish your work.

2. Getting the sample multi-threaded programs and `x86.py` simulator

The two sample multi-threaded programs used in tutorial slides are provided. A simulation program, called `x86.py`, is also given to see how threads interleave. It can mimic the execution of assembly programs (short assembly sequences in .s) by multiple threads. All these codes, simulator and assembly programs are in the directory */public/cs3103/tutorial3*.

Start by copying them to a directory in which you plan to do your work. For example, to copy tutorial3 directory and its contents (7 files) to your current directory and change to it, enter:

```
$ cp -rf /public/cs3103/tutorial3 .
$ cd tutorial3
```

The last dot/period (.) indicates the current directory as destination.

Running the Sample Multi-threaded Programs

In this tutorial, we provide a simple set of programs that use threads: `t0.c` creates two threads, each of which does some independent work, in this case printing "A" or "B". `t1.c` creates two threads, each of which wishes to update a global shared variable, i.e., add a number to the shared variable `counter`, and repeat a "specific" number of times, determined by the user (It's you).

To view the content of the provided codes, use `cat` command:

```
$ cat t0.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}
(content removed for brevity)
```

Or use `nano` editor:

```
$ nano t0.c
```

We now compile and run the programs, to see how they behave. To build them, type "`make`" to the shell.

```
$ make
```

To run program `t0`, type "`./t0`":

```
$ ./t0
main: begin
A
B
main: end
```

To run program `t1`, type "`./t1 <loopcount>`". For example, let's set `loopcount` to 10 million times (10000000). Thus, the desired result is: 20,000,000.

Sometimes, everything works how we might expect:

```
$ ./t1 10000000
main: begin [counter = 0] [0x60108c]
A: begin [addr of i: 0x7fbf228c5f3c]
B: begin [addr of i: 0x7fbf220c4f3c]
B: done
```

```
A: done
main: done
 [counter: 20000000]
 [should: 20000000]
```

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result. Sometimes, we get:

```
$ ./t1 10000000
main: begin [counter = 0] [0x60108c]
A: begin [addr of i: 0x7f8edc567f3c]
B: begin [addr of i: 0x7f8edbd66f3c]
A: done
B: done
main: done
 [counter: 13247747]
 [should: 20000000]
```

Let's try it more times, just to see if we've gone crazy. After all, aren't computers supposed to produce **deterministic** results, as you have been taught?! Not only is each run wrong, but also yields a different result! A big question remains: why does this happen?

To understand why this happens, we must understand the code sequence that the compiler generates for the update to `counter`. Let's run a thread simulator and look at some detailed execution traces to understand the problem better.

Introduction to `x86.py` simulator

The simulator, `x86.py`, will help you to gain familiarity with threads by seeing how they interleave. It mimics the execution of short assembly sequences by multiple threads. Note that the OS code that would run (for example, to perform a context switch) is **not** shown; thus, all you see is the interleaving of the user code.

The assembly code that is run is based on x86, but somewhat simplified. In this instruction set, there are four general-purpose registers (`%ax`, `%bx`, `%cx`, `%dx`), a program counter (PC), and a small set of instructions which will be enough for our purposes.

Here is an example code snippet that we will be able to run:

```
.main
mov 2000, %ax   # get the value at the address
add $1, %ax     # increment it
mov %ax, 2000   # store it back
halt
```

The code is easy to understand. The first instruction, an x86 "`mov`", simply loads a value from the address specified by `2000` into the register `%ax`. Addresses, in this subset of x86, can take some of the following forms:

- `2000`: the number (2000) is the address
- `(%cx)`: contents of register (in parentheses) forms the address
- `1000(%dx)`: the number + contents of the register form the address
- `10(%ax,%bx)`: the number + reg1 + reg2 forms the address

To store a value, the same "`mov`" instruction is used, but this time with the arguments reversed, e.g.:

```
mov %ax, 2000
```

The "`add`" instruction, from the sequence above, should be clear: it adds an immediate value (specified by `$1`) to the register specified in the second argument (i.e., `%ax = %ax + 1`).

Thus, we now can understand the code sequence above: it loads the value at address `2000`, adds `1` to it, and then stores the value back into address `2000`.

The fake-ish "`halt`" instruction just stops running this thread.

To run the program and get its options, do this:

```
$ ./x86.py -h
```

If this doesn't work, type "`python2`" before the command, like this:

```
$ python2 x86.py -h
```

Let's run the simulator and see how this all works! Assume the above code sequence is in the file "simple-race.s".

```
$ ./x86.py -p simple-race.s -t 1
(content removed for brevity, the same hereinafter.)
       Thread 0
1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000
1003 halt
```

The arguments used here specify the program (`-p`), the number of threads (-t 1), and the interrupt interval, which is how often a scheduler will be woken and run to switch to a different task. Because there is only one thread in this example, this interval does not matter.

The output is easy to read: the simulator prints the program counter (here shown from 1000 to 1003) and the instruction that gets executed. Note that we assume (unrealistically) that all instructions just take up a single byte in memory; in x86, instructions are variable-sized and would take up from one to several bytes.

We can use more detailed tracing to get a better sense of how machine state changes during the execution:

```
$ ./x86.py -p simple-race.s -t 1 -M 2000 -R ax,bx
2000      ax    bx          Thread 0
    ?       ?     ?
    ?       ?     ?   1000 mov 2000, %ax
    ?       ?     ?   1001 add $1, %ax
    ?       ?     ?   1002 mov %ax, 2000
    ?       ?     ?   1003 halt
```

Oops! Forgot the `-c` flag (which actually computes the answers for you).

```
$ ./x86.py -p simple-race.s -t 1 -M 2000 -R ax,bx -c

2000      ax    bx          Thread 0
    0       0     0
    0       0     0   1000 mov 2000, %ax
    0       1     0   1001 add $1, %ax
    1       1     0   1002 mov %ax, 2000
    1       1     0   1003 halt
```

By using the `-M` flag, we can trace memory locations (a comma-separated list lets you trace more than one, e.g., 2000,3000); by using the `-R` flag we can track the values inside specific registers.

The values on the left show the memory/register contents **AFTER** the instruction on the right has executed. For example, after the "`add`" instruction, you can see that `%ax` has been incremented to the value 1; after the second "`mov`" instruction (at PC=1002), the memory contents at 2000 are now also incremented.

There are a few more instructions you'll need to know, let's get to them now. Here is a code snippet of a loop:

```
.main
.top
sub  $1,%dx
test $0,%dx
jgte .top
halt
```

A few things have been introduced here. First is the "`test`" instruction. This instruction takes two arguments and compares them; it then sets implicit "condition codes" (kind of like 1-bit registers) which subsequent instructions can act upon.

In this case, the other new instruction is the "`jump`" instruction (in this case, "`jgte`" which stands for "jump if greater than or equal to"). This instruction jumps if the second value is greater than or equal to the first in the test.

One last point: to really make this code work, `dx` must be initialized to 1 or greater.

Thus, we run the program like this:

```
$ ./x86.py -p loop.s -t 1 -a dx=3 -R dx -C -c
   dx    >= >  <= <  != ==        Thread 0
    3    0  0  0  0  0  0
    2    0  0  0  0  0  0  1000 sub  $1,%dx
    2    1  1  0  0  1  0  1001 test $0,%dx
    2    1  1  0  0  1  0  1002 jgte .top
    1    1  1  0  0  1  0  1000 sub  $1,%dx
    1    1  1  0  0  1  0  1001 test $0,%dx
    1    1  1  0  0  1  0  1002 jgte .top
    0    1  1  0  0  1  0  1000 sub  $1,%dx
    0    1  0  1  0  0  1  1001 test $0,%dx
    0    1  0  1  0  0  1  1002 jgte .top
   -1    1  0  1  0  0  1  1000 sub  $1,%dx
   -1    0  0  1  1  1  0  1001 test $0,%dx
   -1    0  0  1  1  1  0  1002 jgte .top
   -1    0  0  1  1  1  0  1003 halt
```

The "`-R dx`" flag traces the value of `%dx`; the "`-C`" flag traces the values of the condition codes that get set by a test instruction. Finally, the "`-a dx=3`" flag sets the `%dx` register to the value 3 to start with.

As you can see from the trace, the "`sub`" instruction slowly lowers the value of `%dx`. The first few times "`test`" is called, only the "`>=`", "`>`", and "`!=`" conditions get set. However, the last "`test`" in the trace finds `%dx` and `0` to be equal, and thus the subsequent jump does **NOT** take place, and the program finally halts.

Now, finally, we get to a more interesting case, i.e., a race condition with multiple threads. Let's look at the code first:

```
.main
.top
# critical section
mov 2000, %ax  # get 'value' at address 2000
add $1, %ax    # increment it
mov %ax, 2000  # store it back

# see if we're still looping
sub  $1, %bx
```

```
test $0, %bx
jgt .top

halt
```

The code has a critical section which loads the value of a variable (at address 2000), then adds 1 to the value, then stores it back. The code after just decrements a loop counter (in `%bx`), tests if it is greater than zero, and if so, jumps back to the top to the critical section again.

```
$ ./x86.py -p looping-race-nolock.s -t 2 -a bx=1 -M 2000 -c
2000          Thread 0                Thread 1
    0
    0   1000 mov 2000, %ax
    0   1001 add $1, %ax
    1   1002 mov %ax, 2000
    1   1003 sub  $1, %bx
    1   1004 test $0, %bx
    1   1005 jgt .top
    1   1006 halt
    1   ----- Halt;Switch -----  ----- Halt;Switch -----
    1                           1000 mov 2000, %ax
    1                           1001 add $1, %ax
    2                           1002 mov %ax, 2000
    2                           1003 sub  $1, %bx
    2                           1004 test $0, %bx
    2                           1005 jgt .top
    2                           1006 halt
```

Here you can see each thread ran once, and each updated the shared variable at address 2000 once, thus resulting in a count of two there.

The "`Halt;Switch`" line is inserted whenever a thread halts and another thread must be run.

One last example: run the same thing above, but with a smaller interrupt frequency. Here is what that will look like:

```
$ ./x86.py -p looping-race-nolock.s -t 2 -a bx=1 -M 2000 -i 2
2000          Thread 0                Thread 1
    ?
    ?   1000 mov 2000, %ax
    ?   1001 add $1, %ax
    ?   ------ Interrupt ------  ------ Interrupt ------
    ?                           1000 mov 2000, %ax
```

```
    ?                           1001 add $1, %ax
    ?   ------ Interrupt ------  ------ Interrupt ------
    ?   1002 mov %ax, 2000
    ?   1003 sub  $1, %bx
    ?   ------ Interrupt ------  ------ Interrupt ------
    ?                           1002 mov %ax, 2000
    ?                           1003 sub  $1, %bx
    ?   ------ Interrupt ------  ------ Interrupt ------
    ?   1004 test $0, %bx
    ?   1005 jgt .top
    ?   ------ Interrupt ------  ------ Interrupt ------
    ?                           1004 test $0, %bx
    ?                           1005 jgt .top
    ?   ------ Interrupt ------  ------ Interrupt ------
    ?   1006 halt
    ?   ----- Halt;Switch -----  ----- Halt;Switch -----
    ?                           1006 halt
```

As you can see, each thread is interrupt every 2 instructions, as we specify via the "`-i 2`" flag. What is the value of memory[2000] throughout this run? What should it have been?

Now let's give a little more information on what can be simulated with this program. The full set of registers: `%ax`, `%bx`, `%cx`, `%dx`, and the `PC`. In this version, there is no support for a "stack", nor are there call and return instructions.

The full set of instructions simulated are:

```
mov immediate, register     # moves immediate value to register
mov memory, register        # loads from memory into register
mov register, register      # moves value from one register to other
mov register, memory        # stores register contents in memory
mov immediate, memory       # stores immediate value in memory

add immediate, register     # register  = register  + immediate
add register1, register2    # register2 = register2 + register1
sub immediate, register     # register  = register  - immediate
sub register1, register2    # register2 = register2 - register1

test immediate, register    # compare immediate and register (set
condition codes)
test register, immediate    # same but register and immediate
test register, register     # same but register and register
```

```
jne                         # jump if test'd values are not equal
je                          #                        ... equal
jlt                         #     ... second is less than first
jlte                        #                ... less than or equal
jgt                         #            ... is greater than
jgte                        #                ... greater than or equal

xchg register, memory       # atomic exchange:
                            #   put value of register into memory
                            #   return old contents of memory into reg
                            # do both things atomically

nop                         # no op
```

Notes:
- '`immediate`' is something of the form `$number`
- '`memory`' is of the form '`number`' or '`(reg)`' or '`number(reg)`' or '`number(reg,reg)`' (as described above)
- '`register`' is one of `%ax`, `%bx`, `%cx`, `%dx`

Finally, the full set of options to the simulator are available with the `-h` flag:

```
$ ./x86.py -h
```

Most are obvious. Usage of `-r` turns on a random interrupter (from 1 to interrupt frequency as specified by `-i`), which can make for more fun during tutorial problems.

Now you have the basics in place; read the questions to study race condition and related issues in more depth.

Questions

All questions should be answered on the separate answer sheet provided.

1. Let's examine a simple program, "`loop.s`". First, just read and understand it. Then, run it with these arguments (`./x86.py -p loop.s -t 1 -i 100 -R dx`). This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. What will `%dx` be during the run? Use the `-c` flag to check your answers; the answers, on the left, show the value of the register (or memory value) after the instruction on the right has run.

2. Same code, different flags: (`./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx`). This specifies two threads, and initializes each `%dx` to 3. What values will `%dx` see? Run with `-c` to check. Does the presence of multiple threads affect your calculations? Is there a race in this code?

3. Run this: `./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx`. This makes the interrupt interval small/random; use different seeds (`-s 1`, `-s 2`, etc) to see different interleavings. Does the interrupt frequency change anything?

4. Now, a different program, `looping-race-nolock.s`, which accesses a shared variable located at address 2000; we'll call this variable `value`. Run it with a single thread to confirm your understanding: `./x86.py -p looping-race-nolock.s -t 1 -M 2000`. What is value (i.e., at memory address 2000) throughout the run? Use `-c` to check.

5. Run with multiple iterations/threads: `./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000`. Why does each thread loop three times? What is final value of `value`?

6. Run with random interrupt intervals: `./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0` with different seeds (`-s 1`, `-s 2`, etc.) Can you tell by looking at the thread interleaving what the final value of `value` will be? Does the timing of the interrupt matter? Where can it safely occur? Where not? In other words, where is the critical section exactly?

7. Now examine fixed interrupt intervals: `./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1`. What will the final value of the shared variable `value` be? What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the "correct" answer?

8. Run the same for more loops (e.g., set `-a bx=100`). What interrupt intervals (`-i`) lead to a correct outcome? Which intervals are surprising (unexpected)?