Tutorial 3 Answer Sheet

CS3103 Operating Systems

Student Name: LIU Hengche
Student No. : 5 7 8 5 4 3 2 9

Day: ☐Tuesday ☑Wednesday ☐Friday
Time: ☐ 10:00 - 10:50 ☐ 11:00 - 11:50 ☐ 16:00 - 16:50 ☑ 18:00 - 18:50

The Threads

Submission:

- Deadline: Sunday, March 3, 2024, 23:59 pm HKT.
- Answers are allowed in text only. Any form of image/snapshot is not allowed.
- Submit this answer sheet via Canvas->Assignments->Tutorial 3.

Questions

Question 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.

Answer:

%dx would be -1 during the run. This is because since we did not set the initial value of the dx register, then the default value is 0, after the instruction "sub $1, %dx", which subtracts the immediate value 1 from the value in the %dx register, the value stored in dx would be -1. Since 0 is greater than -1, then we will not jump to the .top but halt the program. Then the register dx's value would be -1 through out the instructions.

Below is the output:

```
   dx      Thread 0

    0

   -1      1000 sub    $1,%dx

   -1      1001 test $0,%dx

   -1      1002 jgte .top

   -1      1003 halt
```

Question 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?

Answer:

The output is as follows:

```
   dx       Thread 0                  Thread 1
    3
    2     1000 sub    $1,%dx
    2     1001 test $0,%dx
    2     1002 jgte .top
    1     1000 sub    $1,%dx
    1     1001 test $0,%dx
    1     1002 jgte .top
    0     1000 sub    $1,%dx
    0     1001 test $0,%dx
    0     1002 jgte .top
   -1     1000 sub    $1,%dx
   -1     1001 test $0,%dx
   -1     1002 jgte .top
   -1     1003 halt
    3     ----- Halt;Switch -----   ----- Halt;Switch -----
    2                               1000 sub    $1,%dx
    2                               1001 test $0,%dx
    2                               1002 jgte .top
    1                               1000 sub    $1,%dx
    1                               1001 test $0,%dx
    1                           1002 jgte .top

    0                           1000 sub    $1,%dx

    0                           1001 test $0,%dx

    0                           1002 jgte .top

   -1                           1000 sub    $1,%dx

   -1                           1001 test $0,%dx

   -1                           1002 jgte .top

   -1                           1003 halt
```

The presence of multiple threads does not effect our calculation because there's no race between the two threads as the interruption frequency is set to 100 which is sufficient for the thread to finish before context switching. The outputs of dx is exactly the same for the two threads because they are both initially set to 3 in the beginning.

Question 3: 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.

Answer:

The output is as follows:

```
 2000      Thread 0

    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
```

After the addition, register ax has the value of 1, which is then moved to the memory location of 2000. Because register bx is not set in the beginning, then it's the default value of 0. After subtracting the value will be -1 which is already smaller than 0 so there would not be any extra loops for memory 2000 to increase its value. Therefore, memory 2000 will have the value of 1 throughout the instructions after moving ax into it.

Question 4: 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?

Answer:

The final value is determined by the timing of the interleaving of thread. If register ax's value is changed by the next time of addition, then memory 2000's final value would be 2 (seed 0 and 2), otherwise it's 1 (seed 1). The timing of interrupt matters. If the interrupt occurs at the critical section, the output may be wrong.

Critical Section:

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

Question 5: 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?

Answer:

The critical section has the length of three instructions. So when the interruption frequency is less than 3, i.e. when i = 1 or 2, the value is 1, which is the wrong output. When the interrupt is 3, i.e. when i=3, the program give the "correct" answer of 2

Question 6: 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)?

Answer:

When the interrupt interval is a multiple of 3(the length of instructions of the critical section), or other values that avoid the interruption in the critical section, the outcome is correct. The other intervals are surprising (i = 1,2,4,5...)