#uni/semester3/Betriebssysteme/chapter26/simulator-fragen

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.

```
vladb@VladB ~/G/h/S/B/H/HW26-ThreadsIntro (master)>
./x86.py -p loop.s -t 1 -i 100 -R dx -c
ARG seed 0
ARG numthreads 1
ARG program loop.s
ARG interrupt frequency 100
ARG interrupt randomness False
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace dx
ARG cctrace False
ARG printstats False
ARG verbose False

   dx          Thread 0
    0
   -1    1000 sub  $1,%dx
   -1    1001 test $0,%dx
   -1    1002 jgte .top
   -1    1003 halt
```

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?

```
vladb@VladB ~/G/h/S/B/H/HW26-ThreadsIntro (master)>
./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx -c
ARG seed 0
ARG numthreads 2
ARG program loop.s
ARG interrupt frequency 100
ARG interrupt randomness False
ARG argv dx=3,dx=3
ARG load address 1000
ARG memsize 128
ARG memtrace
ARG regtrace dx
ARG cctrace False
ARG printstats False
ARG verbose False

   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
```

➜ No race in this code, because each thread has its own dx on its stack.

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) to see different interleavings. Does the interrupt frequency change anything?

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

➜ Altering the frequency doesn't really change anything, besides the ordering. It Is because there is no race in the code.

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.

```
vladb@VladB ~/G/h/S/B/H/HW26-ThreadsIntro (master)>
./x86.py -p looping-race-nolock.s -t 1 -M 2000 -c
ARG seed 0
ARG numthreads 1
ARG program looping-race-nolock.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False

 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
```

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?

➜ value = 6, count the positions where mov %ax, 2000

➜ three times because:

➜ bx is init with the value of three

➜ in code line 13 there is jgt, which stands for jump if second value greater than first.

➜ bx is not global but on each stack of the threads. Which means that changing bx in one thread, won't change it in the other.

```
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   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   1000 mov 2000, %ax
   2   1001 add $1, %ax
   3   1002 mov %ax, 2000
   3   1003 sub  $1, %bx
   3   1004 test $0, %bx
   3   1005 jgt .top
   3   1006 halt
   3   ----- Halt;Switch -----  ----- Halt;Switch -----
   3                            1000 mov 2000, %ax
   3                            1001 add $1, %ax
   4                            1002 mov %ax, 2000
   4                            1003 sub  $1, %bx
   4                            1004 test $0, %bx
   4                            1005 jgt .top
   4                            1000 mov 2000, %ax
   4                            1001 add $1, %ax
   5                            1002 mov %ax, 2000
   5                            1003 sub  $1, %bx
   5                            1004 test $0, %bx
   5                            1005 jgt .top
   5                            1000 mov 2000, %ax
   5                            1001 add $1, %ax
   6                            1002 mov %ax, 2000
   6                            1003 sub  $1, %bx
   6                            1004 test $0, %bx
   6                            1005 jgt .top
   6                            1006 halt
```

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?

➜ Yes, look at where the value is loaded from address, added then saved back.
➜ The timing of the interrupt is essential, because there is a difference if an interrupt occurs right after the value was loaded from the address or if it happens right after the value is saved back to the address. In the former case the other thread would also start working on that value and so synchronisation will be damaged.
➜ The interrupt can safely occur after the value was loaded, incremented and stored back. If it is done before one of these operations, there won't be a sync with the other threads.
➜ Critical section is PC 1000, 1001, 1002. Or the operations from above.

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?

➜ -i1

```
vladb@VladB ~/G/h/S/B/H/HW26-ThreadsIntro (master)>
./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 -c

 2000          Thread 0                Thread 1

    0
    0   1000 mov 2000, %ax
    0   ------ Interrupt ------  ------ Interrupt ------
    0                           1000 mov 2000, %ax
    0   ------ Interrupt ------  ------ Interrupt ------
    0   1001 add $1, %ax
    0   ------ Interrupt ------  ------ Interrupt ------
    0                           1001 add $1, %ax
    0   ------ Interrupt ------  ------ Interrupt ------
    1   1002 mov %ax, 2000
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1002 mov %ax, 2000
    1   ------ Interrupt ------  ------ Interrupt ------
    1   1003 sub  $1, %bx
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1003 sub  $1, %bx
    1   ------ Interrupt ------  ------ Interrupt ------
    1   1004 test $0, %bx
```

```
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1004 test $0, %bx
    1   ------ Interrupt ------  ------ Interrupt ------
    1   1005 jgt .top
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1005 jgt .top
    1   ------ Interrupt ------  ------ Interrupt ------
    1   1006 halt
    1   ----- Halt;Switch -----  ----- Halt;Switch -----
    1   ------ Interrupt ------  ------ Interrupt ------
    1                           1006 halt
```

➜ -i 3

```
2000          Thread 0                  Thread 1
   0
   0   1000 mov 2000, %ax
   0   1001 add $1, %ax
   1   1002 mov %ax, 2000
   1   ------ Interrupt ------  ------ Interrupt ------
   1                           1000 mov 2000, %ax
   1                           1001 add $1, %ax
   2                           1002 mov %ax, 2000
   2   ------ Interrupt ------  ------ Interrupt ------
   2   1003 sub  $1, %bx
   2   1004 test $0, %bx
   2   1005 jgt .top
   2   ------ Interrupt ------  ------ Interrupt ------
   2                           1003 sub  $1, %bx
   2                           1004 test $0, %bx
   2                           1005 jgt .top
   2   ------ Interrupt ------  ------ Interrupt ------
   2   1006 halt
   2   ----- Halt;Switch -----  ----- Halt;Switch -----
   2                           1006 halt
```

➜ correct value is 2, because there are two threads who increase the value. Bx is set to 0 for both stacks, which means for each thread there is only one loop. And address 2000 has a starting value of 0.

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?

➜ Definitely an interval of 3 because this is the number of the operations in critical sections, and the program starts with these operations. Besides the code of the program has 6 operations with is dividable by three.
➜ Actually all intervals that % 3 = 0, lead to a correct outcome.
➜ Program has 6 lines of code, multiplied by 100 loops, results in 600. So one thread needs an interval of 600 to run till the end. Setting the interval to this value, should also lead to the correct value.

9. One last program: wait-for-me.s. Run: ./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000 This sets the %ax register to 1 for thread 0, and 0 for thread 1, and watches %ax and memory location 2000. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

➜ If ax ist equal to one, it jumps to the .signaller where the 1 is loaded into address 2000, and the thread exits. If it isn't 1, then it jumps to .waiter, where the value of address 2000 is loaded into cx. If cx is already 1, the thread exits. If it isn't 1, it jumps back to .waiter and repeats.

➜ The value at location 2000 is used as an exit clause, where the thread exits only when the value is equal to 1.

➜ Therefore the value should be 1.

```
2000      ax         Thread 0                 Thread 1
   0       1
   0       1   1000 test $1, %ax
   0       1   1001 je .signaller
   1       1   1006 mov  $1, 2000
   1       1   1007 halt
   1       0   ----- Halt;Switch -----  ----- Halt;Switch -----
   1       0                           1000 test $1, %ax
   1       0                           1001 je .signaller
   1       0                           1002 mov  2000, %cx
   1       0                           1003 test $1, %cx
   1       0                           1004 jne .waiter
   1       0                           1005 halt
```

10. Now switch the inputs: ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000 How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., -i 1000, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?

➜ Thread 0 goes into .waiter, because its ax value is 0. It will run in a loop because the value of adress 2000 is 0. It hopes that this value will change to 1, so it can leave. After the default interval has passed, the scheduler switches threads to thread 1. This sets the value to 1 and exits. Scheduler goes back to thread 0, which can also finally exit. So thread 0 was waiting for thread 1 to finish.

➜ Intervals 4 and 5 have the smallest trace. Big interval results in long trace, where the CPU is definitely not used efficiently. You can't get an endless loop, because .waiter doesn't write to the address 2000. In worst cast of a race condition, the finish time of thread 0 just gets delayed, because thread 1 may write the 1 to address 2000, but thread 0 may not get it immediately.

➜ Random can make the trace even smaller, if it interrupts at the right times. This would be, before thread 0 loads address 1000 into its cx. And interrupt again after thread 1, loads the value 1 into address 2000.

➜ The program doesn't use the CPU efficiently, because .waiter checks in a loop if value inside 2000 has changed. This wastes CPU cycles. It would be better if it would only check on the event that value inside 2000 has changed.