

Chapter 26

1. Let's examine a simple program, "loop.s". First, just read and understand it. Then, run it with these arguments (`./x86.py -t 1 -p loop.s -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

```
./x86.py -t 1 -p loop.s -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
```

It decrements %dx from 0 to -1, checks if it's ≥ 0 (it's not), skips the jump, and halts immediately.

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?

```
./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 retrace 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, it does not affect the calculations.

Each thread has its own private register set. The %dx register in Thread 0 is completely independent from the %dx register in Thread 1. They don't share a registered state.

There is no race condition, as each thread operates on its own private registers (%dx is per-thread) and there is no shared memory being accessed.

Further, if the code accessed shared memory, then we could have a race condition. But registers are private to each thread, so they cannot interfere with each other.

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

the interrupt frequency and randomness create very different execution interleavings, but fundamentally they don't change anything important about the program's behavior.

Here, each thread operates in complete isolation. The %dx register belongs privately to each thread, so Thread 0's %dx cannot interfere with Thread 1's %dx. There's no shared memory being accessed, no communication between threads, and no dependencies between what one thread does and what the other thread does. Each thread is essentially running its own independent countdown from 3 to -1.

This situation would be completely different if the code accessed shared memory. If both threads were reading from and writing to the same memory location, then the interrupt frequency would be absolutely critical.

```
./x86.py -p loop.s -t 2 -i 3 -r -R dx -a dx=3,dx=3 T -c
ARG seed 0
ARG numthreads 2
ARG program loop.s
ARG interrupt frequency 3
ARG interrupt randomness True
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
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
```

./x86.py -p loop.s -t 2 -i 3 -r -R dx -a dx=3,dx=3 T -s 1 -c

ARG seed 1

ARG numthreads 2

ARG program loop.s

ARG interrupt frequency 3

ARG interrupt randomness True

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	
3	----- Interrupt -----	----- Interrupt -----
2	1000 sub \$1,%dx	
2	1001 test \$0,%dx	
2	1002 jgte .top	
2	----- Interrupt -----	----- Interrupt -----
2	1001 test \$0,%dx	
2	1002 jgte .top	
1	1000 sub \$1,%dx	
2	----- Interrupt -----	----- Interrupt -----
1	1000 sub \$1,%dx	
1	----- Interrupt -----	----- Interrupt -----
1	1001 test \$0,%dx	
1	1002 jgte .top	
1	----- Interrupt -----	----- Interrupt -----
1	1001 test \$0,%dx	
1	1002 jgte .top	
1	----- Interrupt -----	----- Interrupt -----
0	1000 sub \$1,%dx	
0	1001 test \$0,%dx	
1	----- Interrupt -----	----- Interrupt -----
0	1000 sub \$1,%dx	
0	1001 test \$0,%dx	
0	1002 jgte .top	
0	----- Interrupt -----	----- Interrupt -----
0	1002 jgte .top	
0	----- Interrupt -----	----- Interrupt -----
-1	1000 sub \$1,%dx	
0	----- Interrupt -----	----- Interrupt -----
-1	1000 sub \$1,%dx	
-1	1001 test \$0,%dx	

```

-1 1002 jgte .top
-1 ----- Interrupt ----- Interrupt -----
-1           1001 test $0,%dx
-1           1002 jgte .top
-1 ----- Interrupt ----- Interrupt -----
-1 1003 halt
-1 ----- Halt;Switch ----- Halt;Switch -----
-1           1003 halt

```

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.

```

./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 retrace
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?

Each thread loop three times because ARG argv bx=3.
The final value is 6

```

python3 ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 -c
ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG argv bx=3
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace
ARG cctrace False
ARG printstats False

```

```

ARG verbose False
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?

We can predict the final value. The key is to identify when both threads have loaded the same value from memory before either writes back.

The timing is everything. An interrupt that occurs after the mov-add-mov sequence completes is safe. An interrupt that occurs between the load and the store is dangerous because it allows another thread to interleave its operations, leading to lost updates.

For example, with seed 1: Final value = 1 we have a race here!

```

./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0 -c
ARG seed 0
ARG numthreads 2

```

```
ARG program looping-race-nolock.s
ARG interrupt frequency 4
ARG interrupt randomness True
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False
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 ----- Interrupt ----- Interrupt -----
 1          1000 mov 2000, %ax
 1          1001 add $1, %ax
 2          1002 mov %ax, 2000
 2          1003 sub $1, %bx
 2 ----- Interrupt ----- Interrupt -----
 2          1004 test $0, %bx
 2 1005 jgt .top
 2 ----- Interrupt ----- Interrupt -----
 2          1004 test $0, %bx
 2          1005 jgt .top
 2 ----- Interrupt ----- Interrupt -----
 2 1006 halt
 2 ----- Halt;Switch ----- Halt;Switch -----
 2          1006 halt
ARG seed 1
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 4
ARG interrupt randomness True
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False
2000      Thread 0      Thread 1
 0
 0 1000 mov 2000, %ax
 0 ----- Interrupt ----- Interrupt -----
 0          1000 mov 2000, %ax
 0          1001 add $1, %ax
 1          1002 mov %ax, 2000
 1          1003 sub $1, %bx
 1 ----- Interrupt ----- Interrupt -----
 1          1001 add $1, %ax
 1          1002 mov %ax, 2000
 1          1003 sub $1, %bx
 1          1004 test $0, %bx
 1 ----- Interrupt ----- Interrupt -----
 1          1004 test $0, %bx
 1          1005 jgt .top
 1 ----- Interrupt ----- Interrupt -----
 1          1005 jgt .top
 1          1006 halt
 1 ----- Halt;Switch ----- Halt;Switch -----
 1 ----- Interrupt ----- Interrupt -----
```

```

1          1006 halt

ARG seed 2
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 4
ARG interrupt randomness True
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False
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 ----- Interrupt ----- Interrupt -----
 1           1000 mov 2000, %ax
 1           1001 add $1, %ax
 2           1002 mov %ax, 2000
 2           1003 sub $1, %bx
 2 ----- Interrupt ----- Interrupt -----
 2 1004 test $0, %bx
 2 ----- Interrupt ----- Interrupt -----
 2           1004 test $0, %bx
 2 ----- Interrupt ----- Interrupt -----
 2 1005 jgt .top
 2 1006 halt
2 ----- Halt;Switch ----- Halt;Switch -----
 2           1005 jgt .top
 2           1006 halt

```

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?

The program gives the correct answer when the interrupt interval is greater than or equal to 3, or when the interval is a multiple of 3 that allows the critical section to complete atomically.

```

./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 -c
ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 1
ARG interrupt randomness False
ARG argv bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False
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
```

./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 2 -c
ARG seed 0

ARG numthreads 2

ARG program looping-race-nolock.s

ARG interrupt frequency 2

ARG interrupt randomness False

ARG argv bx=1

ARG load address 1000

ARG memsize 128

ARG memtrace 2000

ARG retrace

ARG cctrace False

ARG printstats False

ARG verbose False

2000 Thread 0 Thread 1

```
0
0 1000 mov 2000, %ax
0 1001 add $1, %ax
0 ----- Interrupt ----- Interrupt -----
0          1000 mov 2000, %ax
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          1006 halt
```

./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 3 -c

```

ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 3
ARG interrupt randomness False
ARG argv bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False
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

```

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?

The correct outcomes occur when the interrupt interval is a multiple of 6, for multiple of 6 we see critical section are divided correctly until the end

```

198 ----- Interrupt ----- Interrupt -----
198           1000 mov 2000, %ax
198           1001 add $1, %ax
199           1002 mov %ax, 2000
199           1003 sub $1, %bx
199           1004 test $0, %bx
199           1005 jgt .top
199           1000 mov 2000, %ax
199           1001 add $1, %ax
200           1002 mov %ax, 2000
200           1003 sub $1, %bx
200           1004 test $0, %bx
200           1005 jgt .top
200 ----- Interrupt ----- Interrupt -----
200           1006 halt
200 ----- Halt;Switch ----- Halt;Switch --

```

But for multiple of 3 but not multiple of 6, we see

```

198 ----- Interrupt ----- Interrupt -----
198           1003 sub $1, %bx
198           1004 test $0, %bx
198           1005 jgt .top
198 ----- Interrupt ----- Interrupt -----

```

```

198 1000 mov 2000, %ax
198 1001 add $1, %ax
199 1002 mov %ax, 2000
199 ----- Interrupt ----- Interrupt -----
199           1000 mov 2000, %ax
199           1001 add $1, %ax
200           1002 mov %ax, 2000
200 ----- Interrupt ----- Interrupt -----
200 1003 sub $1, %bx
200 1004 test $0, %bx
200 1005 jgt .top
200 ----- Interrupt ----- Interrupt -----
200           1003 sub $1, %bx
200           1004 test $0, %bx
200           1005 jgt .top
200 ----- Interrupt ----- Interrupt -----
200 1006 halt
200 ----- Halt;Switch ----- Halt;Switch -----
200           1006 halt

```

The truly surprising result is that interval 3 works perfectly. I initially thought the phase drift would cause problems, but since the critical section is exactly 3 instructions long, an interrupt interval of 3 (or any multiple of 3) ensures the critical section never gets split, regardless of where in the loop the interrupt occurs. The interrupt might happen after the critical section or after the loop control, but never during the read-modify-write sequence.

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?

This program implements a simple signaling mechanism between two threads. Thread 0 acts as the signaller and Thread 1 acts as the waiter.

- Both threads share the same memory address 2000.
- Thread 0 writes 1 to it — acting like a signal flag.
- Thread 1 reads that location to see if it has become 1.

Essentially, this is a very simplified spin-wait signaling

The final value at memory location 2000 will be 1. Thread 0 sets it to 1 when it signals, and it remains 1 for the rest of execution.

```

./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000 -c
ARG seed 0
ARG numthreads 2
ARG program wait-for-me.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG argv ax=1,ax=0
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False

```

```

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?

With the inputs reversed (Thread 0 gets %ax=0, Thread 1 gets %ax=1), the roles are swapped. Thread 0 becomes the waiter and Thread 1 becomes the signaller.

Thread 0 is spinning in a busy-wait loop, wasting CPU cycles. It executes the same three instructions (load, test, jump) over and over, checking memory location 2000 repeatedly to see if Thread 1 has set the signal flag to 1.

With a larger interrupt interval like -i 1000, Thread 0 would spin wastefully for even longer. Busy-waiting is extremely inefficient.

```

./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000 -c
ARG seed 0
ARG numthreads 2
ARG program wait-for-me.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG argv ax=0,ax=1
ARG load address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False
2000  ax      Thread 0      Thread 1
0     0
0     0 1000 test $1, %ax
0     0 1001 je .signaller
0     0 1002 mov 2000, %cx
0     0 1003 test $1, %cx
0     0 1004 jne .waiter
0     0 1002 mov 2000, %cx
0     0 1003 test $1, %cx
0     0 1004 jne .waiter
0     0 1002 mov 2000, %cx
0     0 1003 test $1, %cx
0     0 1004 jne .waiter
0     0 1002 mov 2000, %cx

```


Chapter 28

1. Examine flag.s. This code “implements” locking with a single memory flag. Can you understand the assembly?

Yes, this code tries to implement mutual exclusion spin-lock using a single flag variable to protect a critical section that increments a count variable.

```
.var flag
.var count
.main
.top
.acquire
    mov flag, %ax    # get flag
    test $0, %ax    # if we get 0 back: lock is free!
    jne .acquire    # if not, try again
    mov $1, flag    # store 1 into flag
    # critical section
    mov count, %ax  # get the value at the address
    add $1, %ax    # increment it
    mov %ax, count # store it back
    # release lock
    mov $0, flag    # clear the flag now
    # see if we're still looping
    sub $1, %bx
    test $0, %bx
    jgt .top
    halt
```

2. When you run with the defaults, does flag.s work? Use the -M and -R flags to trace variables and registers (and turn on -c to see their values). Can you predict what value will end up in flag?

```
x86.py -p flag.s -M flag,count -R ax,bx -c
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv
ARG load address 1000
ARG memsize 128
ARG memtrace flag,count
ARG retrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False
flag count    ax   bx      Thread 0          Thread 1
 0   0   0   0
 0   0   0   0 1000 mov flag, %ax
 0   0   0   0 1001 test $0, %ax
 0   0   0   0 1002 jne .acquire
 1   0   0   0 1003 mov $1, flag
```

```

1 0 0 0 1004 mov count, %ax
1 0 1 0 1005 add $1, %ax
1 1 1 0 1006 mov %ax, count
0 1 1 0 1007 mov $0, flag
0 1 1 -1 1008 sub $1, %bx
0 1 1 -1 1009 test $0, %bx
0 1 1 -1 1010 jgt .top
0 1 1 -1 1011 halt
0 1 0 0 ----- Halt;Switch ----- Halt;Switch -----
0 1 0 0 1000 mov flag, %ax
0 1 0 0 1001 test $0, %ax
0 1 0 0 1002 jne .acquire
1 1 0 0 1003 mov $1, flag
1 1 1 0 1004 mov count, %ax
1 1 2 0 1005 add $1, %ax
1 2 2 0 1006 mov %ax, count
0 2 2 0 1007 mov $0, flag
0 2 2 -1 1008 sub $1, %bx
0 2 2 -1 1009 test $0, %bx
0 2 2 -1 1010 jgt .top
0 2 2 -1 1011 halt

```

3. Change the value of the register %bx with the -a flag (e.g., -a bx=2,bx=2 if you are running just two threads). What does the code do? How does it change your answer for the question above?

%bx controls how many times each thread attempts to increment the counter.

```

x86.py -p flag.s -a bx=2,bx=2 -M flag,count -R ax,bx -c
ARG seed 0
ARG numthreads 2
ARG program flag.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched
ARG argv bx=2,bx=2
ARG load address 1000
ARG memsize 128
ARG memtrace flag,count
ARG retrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False
flag count    ax   bx      Thread 0          Thread 1
0 0 0 2
0 0 0 2 1000 mov flag, %ax
0 0 0 2 1001 test $0, %ax
0 0 0 2 1002 jne .acquire
1 0 0 2 1003 mov $1, flag
1 0 0 2 1004 mov count, %ax

```

```

1 0 1 2 1005 add $1, %ax
1 1 1 2 1006 mov %ax, count
0 1 1 2 1007 mov $0, flag
0 1 1 1 1008 sub $1, %bx
0 1 1 1 1009 test $0, %bx
0 1 1 1 1010 jgt .top
0 1 0 1 1000 mov flag, %ax
0 1 0 1 1001 test $0, %ax
0 1 0 1 1002 jne .acquire
1 1 0 1 1003 mov $1, flag
1 1 1 1 1004 mov count, %ax
1 1 2 1 1005 add $1, %ax
1 2 2 1 1006 mov %ax, count
0 2 2 1 1007 mov $0, flag
0 2 2 0 1008 sub $1, %bx
0 2 2 0 1009 test $0, %bx
0 2 2 0 1010 jgt .top
0 2 2 0 1011 halt
0 2 0 2 ----- Halt;Switch ----- ----- Halt;Switch -----
0 2 0 2 1000 mov flag, %ax
0 2 0 2 1001 test $0, %ax
0 2 0 2 1002 jne .acquire
1 2 0 2 1003 mov $1, flag
1 2 2 2 1004 mov count, %ax
1 2 3 2 1005 add $1, %ax
1 3 3 2 1006 mov %ax, count
0 3 3 2 1007 mov $0, flag
0 3 3 1 1008 sub $1, %bx
0 3 3 1 1009 test $0, %bx
0 3 3 1 1010 jgt .top
0 3 0 1 1000 mov flag, %ax
0 3 0 1 1001 test $0, %ax
0 3 0 1 1002 jne .acquire
1 3 0 1 1003 mov $1, flag
1 3 3 1 1004 mov count, %ax
1 3 4 1 1005 add $1, %ax
1 4 4 1 1006 mov %ax, count
0 4 4 1 1007 mov $0, flag
0 4 4 0 1008 sub $1, %bx
0 4 4 0 1009 test $0, %bx
0 4 4 0 1010 jgt .top
0 4 4 0 1011 halt

```

4. Set bx to a high value for each thread, and then use the -i flag to generate different interrupt frequencies; what values lead to a bad outcomes? Which lead to good outcomes?

For example when i = 2, 3, 4 etc we will have race conditions. That occurs when the interrupt is in between the critical sections, which subsequently affects the outcomes.

x86.py -p flag.s -a bx=3,bx=3 -i 2 -M flag,count -R ax,bx -c

```
x86.py -p flag.s -a bx=3,bx=3 -i 3 -M flag,count -R ax,bx -c  
x86.py -p flag.s -a bx=3,bx=3 -i 4 -M flag,count -R ax,bx -c
```

On the other hand, if the interrupt doesn't occur inside the critical section, for example when -i 5, we will have the correct result:

```
1   6   5   1 ----- Interrupt ----- ----- Interrupt -----  
0   6   5   1 1007 mov $0, flag  
0   6   5   0 1008 sub $1, %bx  
0   6   5   0 1009 test $0, %bx  
0   6   5   0 1010 jgt .top  
0   6   5   0 1011 halt  
0   6   6   1 ----- Halt;Switch ----- ----- Halt;Switch -----  
0   6   6   1 ----- Interrupt ----- ----- Interrupt -----  
0   6   6   1             1007 mov $0, flag  
0   6   6   0             1008 sub $1, %bx  
0   6   6   0             1009 test $0, %bx  
0   6   6   0             1010 jgt .top  
0   6   6   0             1011 halt
```

5. Now let's look at the program test-and-set.s. First, try to understand the code, which uses the xchg instruction to build a simple locking primitive. How is the lock acquire written? How about lock release?

In flag.s, the read-test-write sequence was three separate instructions, allowing race conditions. Here, xchg does it atomically, preventing any other thread from sneaking in between the read and write operations.

```
.var mutex  
.var count  
.main  
.top  
.acquire  
mov $1, %ax  
xchg %ax, mutex # atomic swap of 1 and mutex  
test $0, %ax # if we get 0 back: lock is free!  
jne .acquire # if not, try again  
# critical section  
mov count, %ax # get the value at the address  
add $1, %ax # increment it  
mov %ax, count # store it back  
# release lock  
mov $0, mutex  
# see if we're still looping  
sub $1, %bx  
test $0, %bx  
jgt .top  
halt
```

6. Now run the code, changing the value of the interrupt interval (-i) again, and making sure to loop for a number of times. Does the code always work as expected? Does it sometimes lead to an inefficient use of the CPU? How could you quantify that?

Here it always works and generates the correct results. However we have a lot of busy-waitings, which wastes many cpu cycles.

```
x86.py -p test-and-set.s -M mutex,count -R ax,bx -a bx=3,bx=3 -i 2 -c
ARG seed 0
ARG numthreads 2
ARG program test-and-set.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG procsched
ARG argv bx=3,bx=3
ARG load address 1000
ARG memsize 128
ARG memtrace mutex,count
ARG regtrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False
mutex count    ax   bx      Thread 0          Thread 1
  0   0   0   3
  0   0   1   3   1000 mov $1, %ax
  1   0   0   3   1001 xchg %ax, mutex
  1   0   0   3   ----- Interrupt ----- ----- Interrupt -----
  1   0   1   3           1000 mov $1, %ax
  1   0   1   3           1001 xchg %ax, mutex
  1   0   0   3   ----- Interrupt ----- ----- Interrupt -----
  1   0   0   3   1002 test $0, %ax
  1   0   0   3   1003 jne .acquire
  1   0   1   3   ----- Interrupt ----- ----- Interrupt -----
  1   0   1   3           1002 test $0, %ax
  1   0   1   3           1003 jne .acquire
  1   0   0   3   ----- Interrupt ----- ----- Interrupt -----
  1   0   0   3   1004 mov count, %ax
  1   0   1   3   1005 add $1, %ax
  1   0   1   3   ----- Interrupt ----- ----- Interrupt -----
  1   0   1   3           1000 mov $1, %ax
  1   0   1   3           1001 xchg %ax, mutex
  1   0   1   3   ----- Interrupt ----- ----- Interrupt -----
  1   1   1   3   1006 mov %ax, count
  0   1   1   3   1007 mov $0, mutex
  0   1   1   3   ----- Interrupt ----- ----- Interrupt -----
  0   1   1   3           1002 test $0, %ax
  0   1   1   3           1003 jne .acquire
  0   1   1   3   ----- Interrupt ----- ----- Interrupt -----
  0   1   1   2   1008 sub $1, %bx
  0   1   1   2   1009 test $0, %bx
```

0 1 1 3 ----- Interrupt ----- ----- Interrupt -----
0 1 1 3 1000 mov \$1, %ax
1 1 0 3 1001 xchg %ax, mutex
1 1 1 2 ----- Interrupt ----- ----- Interrupt -----
1 1 1 2 1010 jgt .top
1 1 1 2 1000 mov \$1, %ax
1 1 0 3 ----- Interrupt ----- ----- Interrupt -----
1 1 0 3 1002 test \$0, %ax
1 1 0 3 1003 jne .acquire
1 1 1 2 ----- Interrupt ----- ----- Interrupt -----
1 1 1 2 1001 xchg %ax, mutex
1 1 1 2 1002 test \$0, %ax
1 1 0 3 ----- Interrupt ----- ----- Interrupt -----
1 1 1 3 1004 mov count, %ax
1 1 2 3 1005 add \$1, %ax
1 1 1 2 ----- Interrupt ----- ----- Interrupt -----
1 1 1 2 1003 jne .acquire
1 1 1 2 1000 mov \$1, %ax
1 1 2 3 ----- Interrupt ----- ----- Interrupt -----
1 2 2 3 1006 mov %ax, count
0 2 2 3 1007 mov \$0, mutex
0 2 1 2 ----- Interrupt ----- ----- Interrupt -----
1 2 0 2 1001 xchg %ax, mutex
1 2 0 2 1002 test \$0, %ax
1 2 2 3 ----- Interrupt ----- ----- Interrupt -----
1 2 2 2 1008 sub \$1, %bx
1 2 2 2 1009 test \$0, %bx
1 2 0 2 ----- Interrupt ----- ----- Interrupt -----
1 2 0 2 1003 jne .acquire
1 2 2 2 1004 mov count, %ax
1 2 2 2 ----- Interrupt ----- ----- Interrupt -----
1 2 2 2 1010 jgt .top
1 2 1 2 1000 mov \$1, %ax
1 2 2 2 ----- Interrupt ----- ----- Interrupt -----
1 2 3 2 1005 add \$1, %ax
1 3 3 2 1006 mov %ax, count
1 3 1 2 ----- Interrupt ----- ----- Interrupt -----
1 3 1 2 1001 xchg %ax, mutex
1 3 1 2 1002 test \$0, %ax
1 3 3 2 ----- Interrupt ----- ----- Interrupt -----
0 3 3 2 1007 mov \$0, mutex
0 3 3 1 1008 sub \$1, %bx
0 3 1 2 ----- Interrupt ----- ----- Interrupt -----
0 3 1 2 1003 jne .acquire
0 3 1 2 1000 mov \$1, %ax
0 3 3 1 ----- Interrupt ----- ----- Interrupt -----
0 3 3 1 1009 test \$0, %bx
0 3 3 1 1010 jgt .top
0 3 1 2 ----- Interrupt ----- ----- Interrupt -----
1 3 0 2 1001 xchg %ax, mutex

1 3 0 2 1002 test \$0, %ax
1 3 3 1 ----- Interrupt ----- ----- Interrupt -----
1 3 1 1 1000 mov \$1, %ax
1 3 1 1 1001 xchg %ax, mutex
1 3 0 2 ----- Interrupt ----- ----- Interrupt -----
1 3 0 2 1003 jne .acquire
1 3 3 2 1004 mov count, %ax
1 3 1 1 ----- Interrupt ----- ----- Interrupt -----
1 3 1 1 1002 test \$0, %ax
1 3 1 1 1003 jne .acquire
1 3 3 2 ----- Interrupt ----- ----- Interrupt -----
1 3 4 2 1005 add \$1, %ax
1 4 4 2 1006 mov %ax, count
1 4 1 1 ----- Interrupt ----- ----- Interrupt -----
1 4 1 1 1000 mov \$1, %ax
1 4 1 1 1001 xchg %ax, mutex
1 4 4 2 ----- Interrupt ----- ----- Interrupt -----
0 4 4 2 1007 mov \$0, mutex
0 4 4 1 1008 sub \$1, %bx
0 4 1 1 ----- Interrupt ----- ----- Interrupt -----
0 4 1 1 1002 test \$0, %ax
0 4 1 1 1003 jne .acquire
0 4 4 1 ----- Interrupt ----- ----- Interrupt -----
0 4 4 1 1009 test \$0, %bx
0 4 4 1 1010 jgt .top
0 4 1 1 ----- Interrupt ----- ----- Interrupt -----
0 4 1 1 1000 mov \$1, %ax
1 4 0 1 1001 xchg %ax, mutex
1 4 4 1 ----- Interrupt ----- ----- Interrupt -----
1 4 1 1 1000 mov \$1, %ax
1 4 1 1 1001 xchg %ax, mutex
1 4 0 1 ----- Interrupt ----- ----- Interrupt -----
1 4 0 1 1002 test \$0, %ax
1 4 0 1 1003 jne .acquire
1 4 1 1 ----- Interrupt ----- ----- Interrupt -----
1 4 1 1 1002 test \$0, %ax
1 4 1 1 1003 jne .acquire
1 4 0 1 ----- Interrupt ----- ----- Interrupt -----
1 4 4 1 1004 mov count, %ax
1 4 5 1 1005 add \$1, %ax
1 4 1 1 ----- Interrupt ----- ----- Interrupt -----
1 4 1 1 1000 mov \$1, %ax
1 4 1 1 1001 xchg %ax, mutex
1 4 5 1 ----- Interrupt ----- ----- Interrupt -----
1 5 5 1 1006 mov %ax, count
0 5 5 1 1007 mov \$0, mutex
0 5 1 1 ----- Interrupt ----- ----- Interrupt -----
0 5 1 1 1002 test \$0, %ax
0 5 1 1 1003 jne .acquire
0 5 5 1 ----- Interrupt ----- ----- Interrupt -----

```

0 5 5 0 1008 sub $1, %bx
0 5 5 0 1009 test $0, %bx
0 5 1 1 ----- Interrupt ----- ----- Interrupt -----
0 5 1 1 1000 mov $1, %ax
1 5 0 1 1001 xchg %ax, mutex
1 5 5 0 ----- Interrupt ----- ----- Interrupt -----
1 5 5 0 1010 jgt .top
1 5 5 0 1011 halt
1 5 0 1 ----- Halt;Switch ----- ----- Halt;Switch -----
1 5 0 1 ----- Interrupt ----- ----- Interrupt -----
1 5 0 1 1002 test $0, %ax
1 5 0 1 1003 jne .acquire
1 5 0 1 ----- Interrupt ----- ----- Interrupt -----
1 5 5 1 1004 mov count, %ax
1 5 6 1 1005 add $1, %ax
1 5 6 1 ----- Interrupt ----- ----- Interrupt -----
1 6 6 1 1006 mov %ax, count
0 6 6 1 1007 mov $0, mutex
0 6 6 1 ----- Interrupt ----- ----- Interrupt -----
0 6 6 0 1008 sub $1, %bx
0 6 6 0 1009 test $0, %bx
0 6 6 0 ----- Interrupt ----- ----- Interrupt -----
0 6 6 0 1010 jgt .top
0 6 6 0 1011 halt

```

7. Use the -P flag to generate specific tests of the locking code. For example, run a schedule that grabs the lock in the first thread, but then tries to acquire it in the second. Does the right thing happen? What else should you test?

```

yes
x86.py -p test-and-set.s -a bx=1,bx=1 -P 0000011111 -M mutex,count -R ax,bx -c
ARG seed 0
ARG numthreads 2
ARG program test-and-set.s
ARG interrupt frequency 50
ARG interrupt randomness False
ARG procsched 0000011111
ARG argv bx=1,bx=1
ARG load address 1000
ARG memsize 128
ARG memtrace mutex,count
ARG retrace ax,bx
ARG cctrace False
ARG printstats False
ARG verbose False
mutex count    ax   bx      Thread 0          Thread 1
0  0  0  1
0  0  1  1  1000 mov $1, %ax
1  0  0  1  1001 xchg %ax, mutex
1  0  0  1  1002 test $0, %ax

```

```

1 0 0 1 1003 jne .acquire
1 0 0 1 1004 mov count, %ax
1 0 0 1 ----- Interrupt ----- ----- Interrupt -----
1 0 1 1 1000 mov $1, %ax
1 0 1 1 1001 xchg %ax, mutex
1 0 1 1 1002 test $0, %ax
1 0 1 1 1003 jne .acquire
1 0 1 1 1000 mov $1, %ax
1 0 0 1 ----- Interrupt ----- ----- Interrupt -----
1 0 1 1 1005 add $1, %ax
1 1 1 1 1006 mov %ax, count
0 1 1 1 1007 mov $0, mutex
0 1 1 0 1008 sub $1, %bx
0 1 1 0 1009 test $0, %bx
0 1 1 1 ----- Interrupt ----- ----- Interrupt -----
1 1 0 1 1001 xchg %ax, mutex
1 1 0 1 1002 test $0, %ax
1 1 0 1 1003 jne .acquire
1 1 1 1 1004 mov count, %ax
1 1 2 1 1005 add $1, %ax
1 1 1 0 ----- Interrupt ----- ----- Interrupt -----
1 1 1 0 1010 jgt .top
1 1 1 0 1011 halt
1 1 2 1 ----- Halt;Switch ----- ----- Halt;Switch -----
1 2 2 1 1006 mov %ax, count
0 2 2 1 1007 mov $0, mutex
0 2 2 0 1008 sub $1, %bx
0 2 2 0 1009 test $0, %bx
0 2 2 0 1010 jgt .top
0 2 2 0 1011 halt

```

we can also x86.py -p test-and-set.s -a bx=1,bx=1 -P 01010101010101 -M mutex,count -R ax,bx -c to test alternating every instruction during lock acquisition to see if the atomic xchg really prevents races. and it does.

ARG seed 0

ARG numthreads 2

ARG program test-and-set.s

ARG interrupt frequency 50

ARG interrupt randomness False

ARG procsched 01010101010101

ARG argv bx=1,bx=1

ARG load address 1000

ARG memsize 128

ARG memtrace mutex,count

ARG retrace ax,bx

ARG cctrace False

ARG printstats False

ARG verbose False

mutex	count	ax	bx	Thread 0	Thread 1
-------	-------	----	----	----------	----------

0	0	0	1		
---	---	---	---	--	--

0 0 1 1 1000 mov \$1, %ax
0 0 0 1 ----- Interrupt ----- Interrupt -----
0 0 1 1 1000 mov \$1, %ax
0 0 1 1 ----- Interrupt ----- Interrupt -----
1 0 0 1 1001 xchg %ax, mutex
1 0 1 1 ----- Interrupt ----- Interrupt -----
1 0 1 1 1001 xchg %ax, mutex
1 0 0 1 ----- Interrupt ----- Interrupt -----
1 0 0 1 1002 test \$0, %ax
1 0 1 1 ----- Interrupt ----- Interrupt -----
1 0 1 1 1002 test \$0, %ax
1 0 0 1 ----- Interrupt ----- Interrupt -----
1 0 0 1 1003 jne .acquire
1 0 1 1 ----- Interrupt ----- Interrupt -----
1 0 1 1 1003 jne .acquire
1 0 0 1 ----- Interrupt ----- Interrupt -----
1 0 0 1 1004 mov count, %ax
1 0 1 1 ----- Interrupt ----- Interrupt -----
1 0 1 1 1000 mov \$1, %ax
1 0 0 1 ----- Interrupt ----- Interrupt -----
1 0 1 1 1005 add \$1, %ax
1 0 1 1 ----- Interrupt ----- Interrupt -----
1 0 1 1 1001 xchg %ax, mutex
1 0 1 1 ----- Interrupt ----- Interrupt -----
1 1 1 1 1006 mov %ax, count
1 1 1 1 ----- Interrupt ----- Interrupt -----
1 1 1 1 1002 test \$0, %ax
1 1 1 1 ----- Interrupt ----- Interrupt -----
0 1 1 1 1007 mov \$0, mutex
0 1 1 1 ----- Interrupt ----- Interrupt -----
0 1 1 1 1003 jne .acquire
0 1 1 1 ----- Interrupt ----- Interrupt -----
0 1 1 0 1008 sub \$1, %bx
0 1 1 1 ----- Interrupt ----- Interrupt -----
0 1 1 1 1000 mov \$1, %ax
0 1 1 0 ----- Interrupt ----- Interrupt -----
0 1 1 0 1009 test \$0, %bx
0 1 1 1 ----- Interrupt ----- Interrupt -----
1 1 0 1 1001 xchg %ax, mutex
1 1 1 0 ----- Interrupt ----- Interrupt -----
1 1 1 0 1010 jgt .top
1 1 0 1 ----- Interrupt ----- Interrupt -----
1 1 0 1 1002 test \$0, %ax
1 1 1 0 ----- Interrupt ----- Interrupt -----
1 1 1 0 1011 halt
1 1 0 1 ----- Halt;Switch ----- Halt;Switch -----
1 1 0 1 1003 jne .acquire
1 1 1 1 1004 mov count, %ax
1 1 2 1 1005 add \$1, %ax
1 2 2 1 1006 mov %ax, count

0	2	2	1	1007 mov \$0, mutex
0	2	2	0	1008 sub \$1, %bx
0	2	2	0	1009 test \$0, %bx
0	2	2	0	1010 jgt .top
0	2	2	0	1011 halt

8. Now let's look at the code in peterson.s, which implements Peterson's algorithm (mentioned in a sidebar in the text). Study the code and see if you can make sense of it.

The most important logic here is the spinning:

```
.spin1
mov 0(%fx,%cx,4), %ax  # flag[other]
test $1, %ax
jne .fini      # If flag[other] == 0, enter!

.spin2
mov turn, %ax
test %cx, %ax      # Is turn == other?
je .spin1      # If yes, keep spinning
# If no (turn == self), fall through and enter
```

Thread stays in spin loop if BOTH conditions are true:

flag[other] == 1 (Other thread wants to enter)
turn == other (It's the other thread's turn)

Thread enters critical section if EITHER:

flag[other] == 0 (Other thread doesn't want to enter), OR
turn == self (Both want to enter, but it's my turn)

This ensures the turn variable breaks ties - when both threads want in, whoever wrote to turn LAST is the one who waits.

9. Now run the code with different values of -i. What kinds of different behavior do you see? Make sure to set the thread IDs appropriately (using -a bx=0,bx=1 for example) as the code assumes it.

I tried different i, the Peterson's algorithm always works.

```
x86.py -p peterson.s -a bx=0,bx=1 -M flag,turn,count -i 2 -c
x86.py -p peterson.s -a bx=0,bx=1 -M flag,turn,count -i 3 -c
x86.py -p peterson.s -a bx=0,bx=1 -M flag,turn,count -i 4 -c
etc...
```

10. Can you control the scheduling (with the -P flag) to “prove” that the code works? What are the different cases you should show hold? Think about mutual exclusion and deadlock avoidance.

```
./x86.py -p peterson.s -t 2 -a bx=0,bx=1 -M flag,turn,count -R ax,bx,cx  
-P000000000000000000001111111111111111 -c
```

```
./x86.py -p peterson.s -t 2 -a bx=0,bx=1 -M flag,turn,count -R ax,bx,cx -P  
0101010101010101010101010101 -c
```

```
./x86.py -p peterson.s -t 2 -a bx=0,bx=1 -M flag,turn,count -R ax,bx,cx -P  
0000111111111111000000000000 -c
```

Thread 0: flag[0]=1, then pauses

Thread 1: Completes full acquisition (flag[1]=1, turn=0), sees flag[0]=1, but turn=0 so enters CS
Thread 0: Resumes, sets turn=1, eventually sees flag[1]=0, enters

```
./x86.py -p peterson.s -t 2 -a bx=0,bx=1 -M flag,turn,count -R ax,bx,cx -P  
00000111100000000000011111111111 -c
```

After setup: flag[0]=1, flag[1]=0, turn=0

Thread 0 checks: flag[1]=0, turn=0 (my turn) → Enters

Thread 1 checks: flag[0]=1, turn=0 (other's turn) → Spins

11. Now study the code for the ticket lock in ticket.s. Does it match the code in the chapter? Then run with the following flags: -a bx=1000,bx=1000 (causing each thread to loop through the critical section 1000 times). Watch what happens; do the threads spend much time spin-waiting for the lock?

Each thread gets a unique ticket number. When it's a thread's turn, the thread can enter. However, while one thread holds lock, other thread spins checking turn repeatedly. With 1000 iterations, there will be significant spinning.

```
./x86.py -p ticket.s -a bx=1000,bx=1000 -c
```

```
.var ticket  
.var turn  
.var count
```

```
.main  
.top
```

```
.acquire  
mov $1, %ax  
fetchadd %ax, ticket # grab a ticket  
.tryagain  
mov turn, %cx # check if it's your turn  
test %cx, %ax  
jne .tryagain
```

```

# critical section
mov count, %ax      # get the value at the address
add $1, %ax          # increment it
mov %ax, count       # store it back

# release lock
mov $1, %ax
fetchadd %ax, turn

# see if we're still looping
sub $1, %bx
test $0, %bx
jgt .top

halt

```

12. How does the code behave as you add more threads?

```

./x86.py -p ticket.s -t 3 -a bx=100,bx=100,bx=100 -M ticket,turn,count -c
./x86.py -p ticket.s -t 4 -a bx=100,bx=100,bx=100 -M ticket,turn,count -c
./x86.py -p ticket.s -t 5 -a bx=100,bx=100,bx=100 -M ticket,turn,count -c

```

The result is as we increase the num of threads, the spinning time becomes more and more significant. However, this is very fair, as it strictly follows the FIFO principle.

13. Now examine yield.s, in which a yield instruction enables one thread to yield control of the CPU (realistically, this would be an OS primitive, but for the simplicity, we assume an instruction does the task). Find a scenario where test-and-set.s wastes cycles spinning, but yield.s does not. How many instructions are saved? In what scenarios do these savings arise?

The key difference here is If lock is held, voluntarily gives up CPU to let other threads run. For example in a scenario where one thread holds lock for many instructions:

```

./x86.py -p test-and-set.s -t 2 -a bx=1,bx=1 -i 6 -M mutex,count -R ax,bx -c
./x86.py -p yield.s -t 2 -a bx=1,bx=1 -i 10 -M mutex,count -R ax,bx -c

```

Here, yield yields instead of spinning, letting other threads finish faster.

```

.var mutex
.var count

.main
.top

.acquire
mov $1, %ax
xchg %ax, mutex  # atomic swap of 1 and mutex

```

```

test $0, %ax      # if we get 0 back: lock is free!
je .acquire_done
yield            # if not, yield and try again
j .acquire
.acquire_done

# critical section
mov count, %ax  # get the value at the address
add $1, %ax     # increment it
mov %ax, count  # store it back

# release lock
mov $0, mutex

# see if we're still looping
sub $1, %bx
test $0, %bx
jgt .top

halt

```

14. Finally, examine test-and-test-and-set.s. What does this lock do? What kind of savings does it introduce as compared to test-and-set.s?

Test-and-test-and-set adds a regular read (mov mutex, %ax) before the atomic operation (xchg) reduces threads spin by repeatedly reading the lock value and only attempting the atomic swap when it appears free. This saves significant overhead. Although this still busy waiting, but at least we are not doing xchg...

```

.var mutex
.var count

.main
.top

.acquire
mov mutex, %ax
test $0, %ax
jne .acquire
mov $1, %ax
xchg %ax, mutex  # atomic swap of 1 and mutex
test $0, %ax      # if we get 0 back: lock is free!
jne .acquire      # if not, try again

# critical section
mov count, %ax  # get the value at the address
add $1, %ax     # increment it
mov %ax, count  # store it back

# release lock

```

```

mov $0, mutex

# see if we're still looping
sub $1, %bx
test $0, %bx
jgt .top

halt

```

Chapter 29

Q2:

Here we have 24 cores available, however the implementation only has 1 mutex lock, which becomes the bottleneck of the program.

```

lscpu
Architecture:      x86_64
CPU op-mode(s):   32-bit, 64-bit
Byte Order:        Little Endian
CPU(s):           24
On-line CPU(s) list: 0-23
Thread(s) per core: 1
Core(s) per socket: 24
Socket(s):        1
NUMA node(s):     1
Vendor ID:        GenuineIntel
CPU family:       6
Model:            85
Model name:       Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz
Stepping:          7
CPU MHz:          2394.238
BogoMIPS:         4788.74
Hypervisor vendor: Xen
Virtualization type: full
L1d cache:        32K
L1i cache:        32K
L2 cache:         1024K
L3 cache:         16896K
NUMA node0 CPU(s): 0-23
Flags:             fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush acpi mmx fxsr sse
sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl cpuid pn1 pclmulqdq ssse3 fma cx16 pcid sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch
cpuid_fault invpcid_single ssbd ibrs ibpb stibp ibrs_enhanced fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid
rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves pku ospke md_clear flush_l1d arch_capabilities

```

```

[zw335812@login-students homework7]$ ./counter 1 1000000
Threads: 1, Time: 0.0209 sec, Counter: 1000000
[zw335812@login-students homework7]$ ./counter 2 1000000
Threads: 2, Time: 0.1344 sec, Counter: 2000000
[zw335812@login-students homework7]$ ./counter 4 1000000
Threads: 4, Time: 0.2427 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./counter 8 1000000
Threads: 8, Time: 0.7674 sec, Counter: 8000000
[zw335812@login-students homework7]$ ./counter 12 1000000

```

```
Threads: 12, Time: 1.1477 sec, Counter: 12000000
[zw335812@login-students homework7]$ ./counter 24 1000000
Threads: 24, Time: 2.8954 sec, Counter: 24000000
```

q3:

This is faster than a single lock implementation but it is still slower than 1 thread.

```
zw335812@login-students homework7]$ ./approx 4 1000000 1
Threads: 4, Threshold: 1, Time: 0.6350 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./approx 4 1000000 10
Threads: 4, Threshold: 10, Time: 0.3539 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./approx 4 1000000 100
Threads: 4, Threshold: 100, Time: 0.4821 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./approx 4 1000000 1000
Threads: 4, Threshold: 1000, Time: 0.4726 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./approx 4 1000000 10000
Threads: 4, Threshold: 10000, Time: 0.5478 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./approx 4 1000000 100000
Threads: 4, Threshold: 100000, Time: 0.5106 sec, Counter: 4000000
```

```
[zw335812@login-students homework7]$ gcc -o approx approx_counter.c -pthread
[zw335812@login-students homework7]$ ./approx 1 1000000 1024
Threads: 1, Threshold: 1024, Time: 0.0221 sec, Counter: 1000000
[zw335812@login-students homework7]$ ./approx 2 1000000 1024
Threads: 2, Threshold: 1024, Time: 0.2310 sec, Counter: 2000000
[zw335812@login-students homework7]$ ./approx 4 1000000 1024
Threads: 4, Threshold: 1024, Time: 0.4748 sec, Counter: 4000000
[zw335812@login-students homework7]$ ./approx 8 1000000 1024
Threads: 8, Threshold: 1024, Time: 0.8500 sec, Counter: 8000000
[zw335812@login-students homework7]$ ./approx 12 1000000 1024
Threads: 12, Threshold: 1024, Time: 0.7814 sec, Counter: 12000000
[zw335812@login-students homework7]$ ./approx 24 1000000 1024
Threads: 24, Threshold: 1024, Time: 1.2390 sec, Counter: 24000000
```

q4:

```
[zw335812@login-students homework7]$ ./list 4 10 10000
Threads: 4, List: 10, Lookups: 10000
Standard : 0.0051 sec
Hand-Over-Hand: 0.0186 sec
[zw335812@login-students homework7]$ ./list 4 1000 10000
Threads: 4, List: 1000, Lookups: 10000
Standard : 0.1779 sec
Hand-Over-Hand: 0.4382 sec
[zw335812@login-students homework7]$ ./list 8 100 10000
Threads: 8, List: 100, Lookups: 10000
Standard : 0.0449 sec
Hand-Over-Hand: 0.3005 sec
[zw335812@login-students homework7]$ ./list 16 100 10000
Threads: 16, List: 100, Lookups: 10000
Standard : 0.1078 sec
Hand-Over-Hand: 0.6154 sec
```

q5, q6:

```
[zw335812@login-students homework7]$ ./hashtable 1 1000 100000
Threads: 1, Items: 1000, Lookups: 100000, Buckets: 101
Global Lock : 0.0048 sec
Per-Bucket Lock: 0.0050 sec
[zw335812@login-students homework7]$ ./hashtable 2 1000 100000
Threads: 2, Items: 1000, Lookups: 100000, Buckets: 101
Global Lock : 0.0300 sec
Per-Bucket Lock: 0.0166 sec
[zw335812@login-students homework7]$ ./hashtable 4 1000 100000
Threads: 4, Items: 1000, Lookups: 100000, Buckets: 101
Global Lock : 0.0596 sec
Per-Bucket Lock: 0.0219 sec
[zw335812@login-students homework7]$ ./hashtable 8 1000 100000
Threads: 8, Items: 1000, Lookups: 100000, Buckets: 101
Global Lock : 0.2150 sec
Per-Bucket Lock: 0.0384 sec
[zw335812@login-students homework7]$ ./hashtable 12 1000 100000
Threads: 12, Items: 1000, Lookups: 100000, Buckets: 101
Global Lock : 0.3345 sec
Per-Bucket Lock: 0.0522 sec
[zw335812@login-students homework7]$ ./hashtable 24 1000 100000
Threads: 24, Items: 1000, Lookups: 100000, Buckets: 101
Global Lock : 0.6043 sec
Per-Bucket Lock: 0.0918 sec
```