Tobias Sautter BSYS HW8 **19.05.2023**

Question 1

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

```
.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
```

Question 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?

`Python3 ./x86.py -p flag.s -R ax,bx -M flag,count -c`

```
 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
```

Question 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?

`Python3 ./x86.py -p flag.s -R ax,bx -M flag,count -a bx=2,bx=2 -c`

The flag variable is still 0 after both threads but now each thread is looping twice through the critical section, and each thread is acquiring the lock twice.

Question 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?

An interrupt that is a multiple of 11 or 15 works.
We have 11 instructions in our assembly code and if those can always run completely we wont get any data races.
The other possible point of the interrupt would be after 15 instructions but I don't really know why.

Question 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?

```
.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
```

mutex	count	T1 ax	T2 ax	T1	T2	
0	0	1	0	`mov  $1, %ax`		set T1 %ax to 1
1	0	0	0	`xchg %ax, mutex`		atomical swap
1	0	0	0	`test $0, %ax`		test register == true
1	0	0	1		`mov  $1, %ax`	set T1 %ax to 1
1	0	0	1		`xchg %ax, mutex`	atomical swap
1	0	0	1		`test $0, %ax`	test register == false

Instead of checking the ‘global’ mutex variable we now check the thread specific ax register by setting it intentionally to 1 and then swapping it with the ‘global’ mutex variable. Now only the process that has a 0 in the %ax register can unlock mutex again.

Question 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?

`Python3 ./x86.py -p test-and-set.s -R ax,bx -M mutex,count -a bx=10,bx=10 -i 3 -c`

The code now works as expected.
The CPU is not really used efficiently since all threads that don't have the mutex will spin until the mutex is free again.

Question 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?

`Python3 ./x86.py -p test-and-set.s -R ax,bx -M mutex,count -a bx=10,bx=10 -P 011 -c`

No matter which order we use, because of the atomically swap we will never acquire a mutex that is locked by another thread. Performance and fairness are really bad though.

Question 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.

If a thread wants to access a critical section the thread will set its flag of the array to 1 to tell it is interested in accessing the critical section (CS). After that it will spin for a short time and then check if another process also wants to access the same CS. Which ever thread sets the turn variable last will have to wait for the other thread to set its flag to 0 again indicating the CS is done.
This ensures mutual exclusion but is still not fair or really performant

Question 9

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

`Python3 ./x86.py -p peterson.s -a bx=0,bx=1 -R ax,cx,bx -M turn,flag,count -i 3 -c`

The Algorithm works as expected and does ensure mutual exclusion

Question 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.

Type to enter text

Question 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?

The code seems to be the same as in the book and works but each thread spends a lot of time spin waiting for its ticket.

Question 12

How does the code behave as you add more threads?

`Python3 ./x86.py -p ticket.s -R ax,bx,cx -M ticket,turn,count -a bx=1000,bx=1000,bx=1000,bx=1000,bx=1000 -t 5 -c`

When adding more threads we will get even more spin waiting since every thread is spinning for an entire time slice until an interrupt happens.

Question 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?

```
python3 ./x86.py -p yield.s -R ax,bx -M count,mutex -a bx=3,bx=3 -c -i 7
python3 ./x86.py -p test-and-set.s -R ax,bx -M count,mutex -a bx=3,bx=3 -c -i 7
```

Question 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?

The test-and-test-and-set code tests if the mutex is 0 before trying to write to it. This saves 1 instruction when spinning and only writes to mutex when it is actually free and doesn't do so while spinning.