Homework week9

Chapter 26

Q1. 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.

dx Thread 0

0

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

Q2. 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?

%dx starts with 3 and then gradually drops to -1.

No, because the interval is set to 100.

No.

Q3. 1. 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?

No.

Q4. 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.

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

Q5. 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`?

3 times, because bx is set to 3.

6, because each thread is executed three times.

Q6. 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?

I can not tell, it depends on whether the increased ax is put back before being taken out by another thread.

Yes, does matter.

Safely occur if outside of the critical section, while not safe in the critical section.

From line 6 to 8, the lines of code that reads from address 2000 to lines of code that put value back to address 2000.

Q7. 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?

1.

1, 2.

-i ≥ 3.

Q8. 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?

-i = 3 * x, where x is positive integer.

Q9. 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?

Thread 0 is signaller, who sets memory 2000 to 1 then halt.

Thread 1 is waiter, who keeps running in a loop until memory 2000 is 1.

Q10. 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 is waiter this time, and is waiting for signal.

When the interval is small the waiter might have to wait for more loops than when the interval is big.

No.

Chapter 27

Q1. First build `main-race.c`. Examine the code so you can see the (hopefully obvious) data race in the code. Now run `helgrind` (by typing `valgrind --tool=helgrind main-race`) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?

```
==72== ------------------------------------------------------------
==72==
==72== Possible data race during read of size 4 at 0x10C014 by thread #1
==72== Locks held: none
==72==    at 0x109236: main (in
/mnt/c/Users/ZackHu/Desktop/CS5600/homework/week9/main-race)
==72==
==72== This conflicts with a previous write of size 4 by thread #2
==72== Locks held: none
==72==    at 0x1091BE: worker (in
/mnt/c/Users/ZackHu/Desktop/CS5600/homework/week9/main-race)
==72==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-
linux.so)
==72==    by 0x48FBAC2: start_thread (pthread_create.c:442)
==72==    by 0x498CBF3: clone (clone.S:100)
==72==  Address 0x10c014 is 0 bytes inside data symbol "balance"
==72==
```

Yes. The conflict variable’s address and size.

Q2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does helgrind report in each of these cases?

No more errors.

Possible data race.

No errors found.

Q3. Now let’s look at `main-deadlock.c`. Examine the code. This code has a problem known as **deadlock** (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?

Two threads lock up each other, lol.

Thread 1 acquires m1 holding m2, while thread acquires m2 holding m1.

Q4. Now run `helgrind` on this code. What does `helgrind` report?

```
==164==
==164== Thread #3: lock order "0x10C040 before 0x10C080" violated
==164==
==164== Observed (incorrect) order is: acquisition of lock at 0x10C080
==164==    at 0x4850CCF: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==164==    by 0x109288: worker (in /mnt/c/Users/ZackHu/Desktop/CS5600/homework/week9/main-deadlock)
==164==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==164==    by 0x48FBAC2: start_thread (pthread_create.c:442)
==164==    by 0x498CBF3: clone (clone.S:100)
==164==
==164==  followed by a later acquisition of lock at 0x10C040
==164==    at 0x4850CCF: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==164==    by 0x1092C3: worker (in /mnt/c/Users/ZackHu/Desktop/CS5600/homework/week9/main-deadlock)
==164==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==164==    by 0x48FBAC2: start_thread (pthread_create.c:442)
==164==    by 0x498CBF3: clone (clone.S:100)
==164==
==164== Required order was established by acquisition of lock at 0x10C040
==164==    at 0x4850CCF: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==164==    by 0x10920E: worker (in /mnt/c/Users/ZackHu/Desktop/CS5600/homework/week9/main-deadlock)
==164==    by 0x485396A: ??? (in /usr/libexec/valgrind/vgpreload_helgrind-amd64-linux.so)
==164==    by 0x48FBAC2: start_thread (pthread_create.c:442)
==164==    by 0x498CBF3: clone (clone.S:100)
```

==164==
==164== followed by a later acquisition of lock at 0x10C080

Q5. Now run `helgrind` on `main-deadlock-global.c`. Examine the code; does it have the same problem that `main-deadlock.c` has? Should `helgrind` be reporting the same error? What does this tell you about tools like `helgrind`?

Yes the same problem occurs.

Yes.

The same error as last time.

After viewing the code, I think this time should work. However, after checking the helgrind, it does show me it is wrong. I guess there might be some problems with the global lock, is it a real atomic operation, from the result, two threads passed the global lock at the same time.

Q6. Let's next look at `main-signal.c`. This code uses a variable (`done`) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)

The parent falls in the loop and does nothing.

Q7. Now run `helgrind` on this program. What does it report? Is the code correct?

It shows possible race conditions. No

Q8. Now look at a slightly modified version of the code, which is found in `main-signal-cv.c`. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?

Both.

Q9. Once again run `helgrind` on `main-signal-cv`. Does it report any errors?

No errors found.

Chapter 28

Q1. 1. Examine `flag.s`. This code “implements” locking with a single memory flag. Can you understand the assembly?

Yes.

Q2.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`?

Yes. 0.

Q3. 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?

Each thread would have to run bx times. The flag is still equal to 0.

Q4. 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?

bad outcomes: 5,6,7,8,9,10,12,13,14

good outcomes: 11,15,16

Q5. 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?

For lock, it would set mutex to 1 and get the mutex’s old value back in one atomic operation.

For release, it just set mutex to 1.

Q6. 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?

python x86.py -p .\test-and-set.s -r -a bx=10 -c

It always works.

Yes, we might record how many time acquire loops, but all those loops consume CPU time and do nothing.

Q7. 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?

```
-p test-and-set.s -R bx -c -a bx=10,bx=10 -P 0110
```

Yes.

Q8. 1. 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.

Yes.

Q9. 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.

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

Q10. 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.

```
python ./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -P 0000011111
```