Homework 4
CS 345 Operating Systems

Members: Christy Keinsley, Courtney Arrowood, and Lauren Dickman

Chapter 26 Simulation Questions:

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.**

The sub $1, %dx subtracts the value by 1, which takes the 0 to -1. Once it reaches the test 0$,%dx it is compared to the jgte.top and it is compared and evaluated. It fails because the number is less than 0, so it will continue to a halt. It starts at 0 by default and will change to -1 and then stay that way. Running -c confirms that the value ends at -1.

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?**

There is no race in this code, everything runs to completion equally. The register values that seem to be in this process are: 3, 2, 1, 0, and -1. The 3 value being the first value for each thread, because we set the start value to 3, then sub subtracts by one until reaching -1 which the two threads seem to halt when reaching that value. The multiple threads do not seem to matter in this problem, since the program still halts and makes a context switch. Once it makes that context switch during the halt of the first thread (thread 0), the other thread (in this case thread 1) runs and after subtracting to the value -1 the thread halts.

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?**

Realization without -r: First without changing the seed, it would run the first thread (thread 0) until the jump, once hitting the jump it would interrupt and start the second thread (thread 1). Then thread one would repeat this process until its first jump, then again, it gets interrupted by thread 0 and goes until the jump again. It then repeats this process until both are halted at -1. For this simulation when changing the seeds it does not show any real change. Both thread 0 and thread 1 get interrupted 8 times and then thread 0 halts and there's a halt switch, then thread 1 halts. This is apparent across all seeds from 0 to 4. So this answer is only accurate without -r. The flag -r makes the interrupts random and does cause the different seeds to provide change.

Actual Answer: With seed 0, there are 10 interrupts before halt, and then 1 more interrupt after. With seed 1, there are 12 interrupts before halt, and then 0 after the first halt. Seed 2, there are 10 interrupts before halt, and 2 after halt. With seed 3, there are 12 interrupts before the first halt, and 1 interrupt after. With this data we see that the frequency of the interrupts do change when adjusting the value of seed.

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 the value (i.e., at memory address 2000) throughout the run? Use -c to check.**

In the beginning of the thread, the memory address of 2000 had the starting value of 0. After the add instruction, the %ax has been incremented to the value 1; after the second mov instruction (at PC=1002), the memory contents at 2000 are now also incremented. The thread appears to be incremented for about 6 times (PC=1006) until it halts.

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 the final value of value?**

The reason that each thread loops three times is because bx is set to 3. According to ostep-overview the %bx is the loop counter for the thread. The final value of this thread is 6.

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?**

Each thread is interrupted every 4 instructions, as it is specified via the -i 4 flag. The timing of the interrupt does matter since without including the -r flag (interrupts are random), the final value of the threads changes from 1 into the value of 2 in seed 1, for example. It appears that the critical section occurs whether the increased %ax is saved to the value before it resets. This is based on how without the random interrupts the value of the %ax increases and is saved before an interrupt occurs.

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?**

From interrupts less than 3 the final value of the threads are 1. While interrupts greater or equal to 3 have the final value of 2. Based on these results any interrupts that are equal to or greater than 3 seem to give the "correct" answer because the %ax is able to increase and save to the value before it resets, or an interrupt occurs.

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?**

In the interval of 1 (-i = 1), each thread does not loop once for each thread, and the final value is still 1. In the interval of 2 (-i =2), each thread does not loop twice for each thread, in fact it appears that each thread loops over 8 times for each different increment. In the interval of 3 (-i = 3), each thread does not loop three times for each thread. Interval 3 (-i=3), is surprising in terms that it continues beyond 100 and actually stops at 200. This is surprising because in intervals 1 and 2, both halt at 100 while interval 3 continues an extra 100 more loops. It is noticeable that intervals equal or greater than 3 continue to loop, passing the set looping count of 100, with 200 being the correct result. Multiples of 3 also give the correct result, 200, while all others give a variation of 100 up to but not including 200.

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?**

Once thread 0 sets address memory 2000 to value 1, the thread halts. Then thread 1 continues to loop through the thread until address memory 2000 is at value 1. Thus, the final value of these threads are 1. This is seen through the cx% which is the contents of the register (in parentheses) that forms the address, which does not change value until entering thread 1.

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?**

In switching the inputs thread 0 is continuously looping until thread 1 changes the value of the address memory 2000 to 1. Including more intervals or including intervals at random times, actually shortens the waiting period that thread 0 goes through in looping until thread 1 changes the value. In this case, the program is not efficiently using the CPU because it is wasting CPU cycles by having thread 0 looping and waiting for a change from thread 1.

Chapter 27 Coding Questions:

1. 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?

When running, valgrind does point to the correct lines of code. Valgrind states that thread 1 "conflicts with a previous write of size 4 by thread #2," which is another way to state that thread has a conflict address and size.

2. 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?

In removing the balance++ variable within the worker function, results in no errors occurring when running valgrind. Now when adding a lock on one of the updates, the program outputs two errors and it is similar to the first question because it conflicts with address and size. Then when adding a lock to both, helgrind reports 0 errors.

3. 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?

In the code there's an if statement where both outcomes result in a lock, therefore the threads are resulting in a deadlock.

4. Now run helgrind on this code. What does helgrind report?

The error summary of helgrind showcases that it produces 1 error on just simply running the program. It states that the lock order was violated and results in a deadlock.

5. 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?

It appears to give the same error, that being a lock order violation. In both problems it specified thread 3 lock order was violated. Helgrind should not be reporting the exact same error. The global code should not be producing any error because it has an additional global lock, g ,which would prevent a deadlock, since it never locks. This shows us that a tool, helgrind, is not perfect.

6. 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?)

This code is inefficient because the parent thread falls into a spinning cycle, a wait status, and this requires cpu resources and that is why it is inefficient. Overall, the parent is waiting and doing nothing else.

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

Helgrind reports 23 errors from two contexts. There is a conflict of address and size of thread. The code compiles and runs without regular error, but does produce error when run with helgrind, and is inefficient so no this code is not correct. There is also the fact that this program wastes CPU cycles, which results in this code being incorrect.

8. 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?

This code does not produce errors, as well as it does not waste cpu resources. This uses a signaling variable that helps decide when a thread should wait and when the next should execute. For this code it is correct and performs better.

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

When running this program there are no errors to be reported.

Chapter 28 Simulation Questions:

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

The code first gets the flag and first to check if lock is free or not, based on the value of the flag. If the flag has the value 0, then it is free. If the flag is not free then the code jumps if tested values are not equal to zero and try again. In this code the flag is not free and jumps to try to get the flag free. The code flag.s enters the critical section when jumping to acquire the lock free, by trying to get the value at the address, increment the value, and store that value back into the address memory. After flag.s code has gotten past the critical section, it then releases the lock and then checks if the code is still looping. Finally the program halts once the check is completed.

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 the flag?

In running ./x86.py -p flag.s -M flag,count -R ax,bx -C -c, the final value was 0. This makes sense since at the end of the code of flag.s the value is set to 0 in order to free the lock before halting the program. Overall, with this information in mind it is easy to determine that the final value with the flag.s code will always be 0.

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?

The code appears to run each thread into two loops, with each flag being freed and locked twice within each thread. The two threads both have a final value of 0, which does not change the reasoning that flags will always set the value to 0 before halting inorder to free the lock or clear the flag.

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 bad outcomes? Which leads to good outcomes?

It appears that when -i is smaller than 3 for example, the bad results are produced, but when -i is larger than 3 good results are produced. When the bx is different it is difficult to determine a specific critical value of -i. When we did count 20, we saw that there was an add 1$, %ax on thread 0, but before the count was updated there was a switch and thread 1 updated the count. When it switched back to thread 0 it changed its count to 21 because add was executed. But if you look at it, add was executed twice with the value only incremented by 1. This causes the value at the end to not be accurate and not be the values of the two threads bx.

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?

The xchg is a register involving memory. The register stands for atomic exchange, in which it puts a value of register into memory and returns the old contents of memory into the register. The exchg does both things atomically. Both the lock acquire and lock release are written in one command. The lock acquire is written after using the test to check if the lock is free, if not then acquire is used to try again. The lock is released after the thread has passed through the critical section.

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?

In this command line ./x86.py -p test-and-set.s -i 4 -a bx=10,bx=10 -M mutex,count -R ax,bx, the final value of the test-and-set.s is similar to the flag.s program which is 0. Again this makes sense considering before the program halts, it sets the value of mutex to 0 to free the lock. I do not think that there is an inefficient use of the CPU, with the intervals (-i) with no thread looping continuously waiting for one thread to finish. If there were an inefficient use to quantify, it would be if the mutex was set to the value of 1, and never freed the lock, thus causing it to spin, and cause an inefficient use of the cpu.

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?

The flag -P lets you specify exactly which threads run when; for example, 11000 would run thread 1 for 2 instructions, then thread 0 for 3, then repeat. The right thing appears to happen, when running ./x86.py -p test-and-set.s -M mutex,count -R ax,bx -c -a bx=10,bx=10 -P 0011, in the sense that the threads are provided mutual exclusion. Once thread 0 is done with the lock, then thread 1 is given the chance of using the lock. The fairness and performance of the locks should be tested, since the amounts of interrupts differ between the two threads.

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 Pereterson's algorithm used the concept of loads and stores. The idea of Pereterson's algorithm is to ensure that two threads never enter a critical section at the same time. Pereterson's algorithm also ensures that all threads receive mutual exclusion and deadlock avoidance.

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.

For the first value we used 2 for -i. For every two lines it gets updated, or interrupter. However, thread 0 after it has halted it no longer messes with the value or adds to the ax%, while thread 1 constitutes adding to the %ax. For a second value we used 5 for -i. This time for every five lines it gets updated, or interrupted. There seems to be a pattern that once thread 0 halts it never messes, or adds to the %ax. While thread 1 continuously interrupts and adds to the %ax.

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.

The -P flag stands for procsched, this flag controls which thread runs when. In running this command, *./x86.py -p peterson.s -M count,flag,turn -R ax,cx -a bx=0,bx=1 -c -P 0000011111* thread 0 is first up to five times and then switches to thread 1 up to five times. By running this code we are able to control whether the code works or not. By placing the 0000011111, we are able to control what thread executes first, and then so on and so forth. By doing this you can prove the code works, and even when messing with the order of it, because of the spins in the code, it allows for them to finish. In this code, there is a variable called turn, and this allows for mutual exclusion because it tells when one thread will obtain a lock and when another will not. In the code in the book it also has a flag variable, this variable is used to show if a thread acquired the lock and if it is not available or if it does not have the lock and would be available. This helps deadlock avoidance by mutual exclusion and by enforcing a hold and wait mechanism, by a spin in the code.

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?

In running this line ./x86.py -p ticket.s -M count,ticket,turn -R ax,bx,cx -a bx=1000,bx=1000 -c, the process was fast and there did not seem to be much time spent on the spin-waiting for lock. There was no issue with the lock spinning forever, the code for the ticket lock in ticket.s seems to match the code in the chapter in terms of functionality.

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

We attempted ten threads and the screen went wild. It was rapidly going through the process, and it took about 4 minutes to complete. There were also a few points in time where the screen seemed to pause in it's rapid increase in numbers, which could have been a result in one lock spinning for a longer time than others. It ended at 10,000.

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 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?

In running these scenarios *./x86.py -p test-and-set.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7* and *./x86.py -p yield.s -M count,mutex -R ax,bx -a bx=5,bx=5 -c -i 7* that mutex is wasting 3 CPU cycles while the yield only wastes two CPU cycles. It is also noticeable in the yield program that yield is saved as 1004 yield throughout the program at least in one interrupt per thread. Based on the yield saved within each interrupt it is assumed that the yield instructions were saved for each cycle.

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.s* lock appears to be doing an extra test to ensure that the lock is really free. So it basically has two tests prior to going through the critical section. While the *test-and-set.s* only has one test prior to entering the critical section. The *test-and-test-and-set.s* ensures that no unnecessary writing or spinning is required if it double checks that the lock is truly free.