This homework covers OSTEP chapter 28. It is due in class Friday, April 10. Hand in a hardcopy of your answers in class.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

- Copy the directory `threads-locks` (and its contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code so the copied directories don't need to go into your `workspace` directory.)

The `threads-locks` directory contains a python program `x86.py` and a README with instructions for using the program. You can also find the text of the `README` in the appendix section A at the end of this document.

Exercises

Chapter 28 — these problems use the `x86.py` simulator in the `threads-locks` directory. See the `README` in the directory or in appendix section A at the end of this document for more information on the simulator.

Hint for the exercises: as the output gets longer, use `less` to help you page through the output instead of trying to scroll in the terminal window:

```
./x86.py -p race.s -M count -R bx -a bx=5 -t 2 -c | less
```

Press the space bar to advance forward one page and `enter` to advance forward one line, `b` to scroll back one page, and `q` to quit `less` and return to the terminal prompt.

1. The provided `race.s` increments a shared counter without synchronization. It implements the following, where the variable `count` is assumed to be initialized to 0 and `%bx` refers to register `bx`:

```
do {
  count = count+1;
} while ( %bx > 0 );
```

(a) Look at `race.s` and make sure you understand the code. The comments and the C-equivalent code above should give you a good start, and the README provides a more detailed explanation of the assembly language. Also try running it with a single thread:

```
./x86.py -p race.s -M count -R bx -a bx=5 -t 1 -c
```

Make sure you understand the output (in particular, observe what happens with `count` and `%bx`) and the options provided on the command line. (See the README again.)

(b) Introduce a second thread running another copy of the same code:

```
./x86.py -p race.s -M count -R bx -a bx=5 -t 2 -c
```

What happens? Is `count` still correct at the end? Why or why not?

(c) Now try

```
./x86.py -p race.s -M count -R bx -a bx=20 -t 2 -c
```

What does this command line specify differently? What happens this time? Is `count` still correct at the end? Why or why not?

(d) The `-P` flag lets you control when interrupts occur, to allow for fine-grained (and repeatable) study and testing of specific code sequences. Try the following:

```
./x86.py -p race.s -M count -R bx -a bx=5 -t 2 -c -P 000111011100000
```

The schedule `000111011100000` means that thread 0 runs for three instructions (`000`), then thread 1 runs for three (`111`), then thread 0 runs for only one (`0`), then thread 1 gets another three (`111`), then thread 0 runs for five (`0`). The whole pattern then repeats until both threads complete.

- Examine the output. Why does thread 0 get eight instructions in a row at times?
- Also observe `count` and `%bx`. Threads share an address space, so both threads share `count`. What about registers? Are those shared between threads as well or does each thread have its own execution context, like processes? How is this reflected in the output?

(e) Use the `-P` flag to demonstrate the race condition in `race.s` — give a schedule which results in an incorrect total for `count` at the end, and explain what is happening to cause this result.

2. `flag.s` attempts to use a simple flag to protect the critical section of code. It implements the following, where the variables `flag` and `count` are assumed to be initialized to 0 and `%bx` refers to register `bx`:

```
do {
  while ( flag == 1 ) {}
  flag = 1;
  count = count+1;
  flag = 0;
} while ( %bx > 0 );
```

(a) Look at `flag.s` and make sure you understand the code. Also run it with the following command line:

```
./x86.py -p flag.s -M flag,count -R bx -a bx=3 -t 1 -c
```

Make sure you understand the output, including when the flag is acquired and released.

(b) Add a second thread (`-t 2`) and give a schedule (`-P`) that demonstrates the flag "working" — the output trace should show that the end result for `count` is correct and there should be at least one instance of a thread spinning if it tries to acquire the lock while the other thread is in the critical section. (Setting up a schedule where no thread ever tries to enter a critical section while the other the other thread is there doesn't count.)

(c) Still with two threads, give a schedule that shows the race condition actually isn't fixed — `count` is incorrect at the end. Also explain what is happening to cause this result.

3. `test-and-set.s` uses the `xchg` instruction to successfully protect the critical section of code. It implements the following, where the variables `mutex` and `count` are assumed to be initialized to 0 and `%bx` refers to register `bx`:

```
do {
  while ( xchg(1,mutex) == 1 ) {}
  count = count+1;
  mutex = 0;
} while ( %bx > 0 );
```

(a) Look at `test-and-set.s` and make sure you understand the code. Also try running it with a single thread:

```
./x86.py -p test-and-set.s -M mutex,count -R bx -a bx=3 -t 1 -c
```

Make sure you understand the output, including when the lock (`mutex`) is acquired and released.

(b) Add a second thread (`-t 2`) and use the `-P` flag to thoroughly test the code — give a set of schedules that demonstrate that the lock works and it is not possible for two threads to end up in the critical section at the same time. Explain what each schedule demonstrates and why the combination covers all the bases.

4. Achieving mutual exclusion within critical sections is essential for correct execution of code, but how a lock is implemented can have a significant impact on performance.

   (a) `ticket.s` uses a ticket lock instead of a test-and-set approach. Look at the code and make sure you understand it — the implementation is a little different than what was discussed in class, though the spirit is the same. Also try running it with a single thread:

   ```
   ./x86.py -p ticket.s -M ticket,turn,count -R bx -a bx=3 -t 1 -c
   ```

   Make sure you understand the output, including the values of (`ticket` and `turn` and how that relates to when the lock is acquired and released.

   (b) What's the performance overhead of simply acquiring and releasing locks when there's no contention? (i.e. no thread has to wait) This can be measured by counting the number of instructions executed when only one thread is running. Start with a baseline: how many instructions are executed when running just the counting loop? To find out, run:

   ```
   ./x86.py -p race.s -a bx=1000 -t 1 | egrep "^[ ]*[0-9]{4}"  | wc -l
   ```

   The `egrep` pattern matches output lines consisting of instructions and `wc -l` (with a lowercase *l*, not the number 1) counts the number of lines — this command line runs 1000 repetitions of the unprotected counting loop and reports the number of instructions executed.

   Determine the number of instructions executed by `test-and-set.s` and `ticket.s`, respectively. How do they compare with the baseline of no locks and with each other?

   (c) When there's contention, how much CPU time is wasted spinning? (This is CPU time that could be used for actual computation rather than uselessly checking locks, and is not referring to an increase in turnaround time for a thread due to it having to wait for a lock to become available.) Ideally n threads all running the same code would not result in any more instructions executed than one thread running the code n times itself.

   To evaluate this, compare the average number of instructions executed per thread with the one-thread case for both `test-and-set.s` and `ticket.s` — repeat the command line from the previous part with 2–5, 10, and 15 threads to get the total number of instructions, then divide by the number of threads. (Be sure to use the right setting for `-t`.)

   What do you observe? Explain the behavior in terms of how spinlocks work.

   (d) The idea of yielding instead of spinning when a lock is not available was introduced to improve scalability and performance. Repeat the previous step with `yield.s` — gather data on the average number of instructions executed per thread with the same numbers of threads. What do you observe?

Consider both how the performance of `yield.s` scales with more threads and in comparison to `test-and-set.s`. Explain the observed behavior in terms of how the locks are implemented.

A x86

Welcome to this simulator! The idea is to gain familiarity with threads by seeing how they interleave; the simulator, `x86.py`, will help you in gaining this understanding.

The simulator mimicks the execution of short assembly sequences by multiple threads. Note that the OS code that would run (for example, to perform a context switch) is not shown; thus, all you see is the interleaving of the user code.

The assembly code that is run is based on x86, but somewhat simplified. In this instruction set, there are four general-purpose registers (`%ax, %bx, %cx, %dx`), a program counter (PC), and a small set of instructions which will be enough for our purposes. We've also added a few extra GP registers (`%ex, %fx`) which don't quite match anything in x86 (but that is OK, isn't it?).

Here is an example code snippet that we will be able to run:

```
.main
mov 2000, %ax   # get the value at the address
add $1, %ax     # increment it
mov %ax, 2000   # store it back
halt
```

The code is easy to understand. The first instruction, an x86 "mov", simply loads a value from the address specified by 2000 into the register %ax. Addresses, in this subset of x86, can take some of the following forms:

- `2000`: the number (2000) is the address
- `(%cx)`: contents of register (in parentheses) forms the address
- `1000(%dx)`: the number + contents of the register form the address
- `10(%ax,%bx)`: the number + reg1 + reg2 form the address
- `10(%ax,%bx,4)`: -¿ the number + reg1 + (reg2*scaling) form the address

To store a value, the same `mov` instruction is used, but this time with the arguments reversed, e.g.:

```
mov %ax, 2000
```

The `add` instruction, from the sequence above, should be clear: it adds an immediate value (specified by `$1`) to the register specified in the second argument (i.e., `%ax = %ax + 1`).

Thus, we now can understand the code sequence above: it loads the value at address 2000, adds 1 to it, and then stores the value back into address 2000.

The fake-ish `halt` instruction just stops running this thread.

Let's run the simulator and see how this all works! Assume the above code sequence is in the file `simple-race.s`.

```
prompt> ./x86.py -p simple-race.s -t 1

       Thread 0
1000 mov 2000, %ax
1001 add $1, %ax
1002 mov %ax, 2000
1003 halt

prompt>
```

The arguments used here specify the program (`-p`), the number of threads (`-t 1`), and the interrupt interval, which is how often a scheduler will be woken and run to switch to a different task. Because there is only one thread in this example, this interval does not matter.

The output is easy to read: the simulator prints the program counter (here shown from 1000 to 1003) and the instruction that gets executed. Note that we assume (unrealistically) that all instructions just take up a single byte in memory; in x86, instructions are variable-sized and would take up from one to a small number of bytes.

We can use more detailed tracing to get a better sense of how machine state changes during the execution:

```
prompt> ./x86.py -p simple-race.s -t 1 -M 2000 -R ax,bx

 2000      ax    bx          Thread 0
    ?       ?     ?
    ?       ?     ?   1000 mov 2000, %ax
    ?       ?     ?   1001 add $1, %ax
    ?       ?     ?   1002 mov %ax, 2000
    ?       ?     ?   1003 halt
```

Oops! Forgot the `-c` flag (which actually computes the answers for you).

```
prompt> ./x86.py -p simple-race.s -t 1 -M 2000 -R ax,bx -c

 2000      ax    bx          Thread 0
    0       0     0
    0       0     0   1000 mov 2000, %ax
    0       1     0   1001 add $1, %ax
    1       1     0   1002 mov %ax, 2000
    1       1     0   1003 halt
```

By using the `-M` flag, we can trace memory locations (a comma-separated list lets you trace more than one, e.g., 2000,3000); by using the -R flag we can track the values inside specific registers.

The values on the left show the memory/register contents AFTER the instruction on the right has executed. For example, after the `add` instruction, you can see that %ax has been incremented to the value 1; after the second `mov` instruction (at PC=1002), you can see that the memory contents at 2000 are now also incremented.

There are a few more instructions you'll need to know, so let's get to them now. Here is a code snippet of a loop:

```
.main
.top
sub  $1,%dx
test $0,%dx
jgt .top
halt
```‘‘‘sh

A few things have been introduced here. First is the ‘test‘ instruction. This instruction takes two arguments and compares them; it then sets implicit "condition codes" (kind of like 1-bit registers) which subsequent instructions can act upon.

In this case, the other new instruction is the ‘jump‘ instruction (in this case, ‘jgte‘ which stands for "jump if greater than or equal to"). This instruction jumps if the second value is greater than or equal to the first in the test.

One last point: to really make this code work, dx must be initialized to 1 or greater.

Thus, we run the program like this:

```‘‘‘sh
prompt> ./x86.py -p loop.s -t 1 -a dx=3 -R dx -C -c

   dx   >= >  <= <  != ==        Thread 0
    3   0  0  0  0  0  0
    2   0  0  0  0  0  0  1000 sub  $1,%dx
    2   1  1  0  0  1  0  1001 test $0,%dx
    2   1  1  0  0  1  0  1002 jgte .top
    1   1  1  0  0  1  0  1000 sub  $1,%dx
    1   1  1  0  0  1  0  1001 test $0,%dx
    1   1  1  0  0  1  0  1002 jgte .top
    0   1  1  0  0  1  0  1000 sub  $1,%dx
    0   1  0  1  0  0  1  1001 test $0,%dx
```

```
    0   1  0  1  0  0  1  1002 jgte .top
    0   1  0  1  0  0  1  1003 halt
```

The `-R dx` flag traces the value of %dx; the `-C` flag traces the values of the condition codes that get set by a test instruction. Finally, the `-a dx=3` flag sets the `%dx` register to the value 3 to start with.

As you can see from the trace, the `sub` instruction slowly lowers the value of %dx. The first few times `test` is called, only the "¿=", "¿", and "!=" conditions get set. However, the last `test` in the trace finds %dx and 0 to be equal, and thus the subsequent jump does NOT take place, and the program finally halts.

Now, finally, we get to a more interesting case, i.e., a race condition with multiple threads. Let's look at the code first:

```
.main
.top
# critical section
mov 2000, %ax       # get the value at the address
add $1, %ax         # increment it
mov %ax, 2000       # store it back

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

halt
```

The code has a critical section which loads the value of a variable (at address 2000), then adds 1 to the value, then stores it back.

The code after just decrements a loop counter (in `%bx`), tests if it is greater than or equal to zero, and if so, jumps back to the top to the critical section again.

```
prompt> ./x86.py -p looping-race-nolock.s -t 2 -a bx=1 -M 2000 -c

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

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

Here you can see each thread ran once, and each updated the shared variable at address 2000 once, thus resulting in a count of two there.

The `Halt;Switch` line is inserted whenever a thread halts and another thread must be run.

One last example: run the same thing above, but with a smaller interrupt frequency. Here is what that will look like:

```
[mac Race-Analyze] ./x86.py -p looping-race-nolock.s -t 2 -a bx=1 -M 2000 -i 2

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

As you can see, each thread is interrupt every 2 instructions, as we specify via the `-i 2` flag. What is the value of memory[2000] throughout this run? What should it have been?

Now let's give a little more information on what can be simulated with this program. The full set of registers: `%ax, %bx, %cx, %dx, %ex, %fx` and the PC and a stack pointer `%sp`.

The full set of instructions simulated are:

```
mov immediate, register     # moves immediate value to register
mov memory, register        # loads from memory into register
mov register, register      # moves value from one register to other
mov register, memory        # stores register contents in memory
mov immediate, memory       # stores immediate value in memory

add immediate, register     # register  = register  + immediate
add register1, register2    # register2 = register2 + register1
sub immediate, register     # register  = register  - immediate
sub register1, register2    # register2 = register2 - register1

neg register                # negates contents of register

test immediate, register    # compare immediate and register (set condition codes)
test register, immediate    # same but register and immediate
test register, register     # same but register and register

jne                         # jump if test'd values are not equal
je                          #                         ... equal
jlt                         #     ... second is less than first
jlte                        #               ... less than or equal
jgt                         #            ... is greater than
jgte                        #               ... greater than or equal

push memory or register     # push value in memory or from reg onto stack
                            # stack is defined by sp register
pop [register]              # pop value off stack (into optional register)
call label                  # call function at label

xchg register, memory       # atomic exchange:
                            #   put value of register into memory
                            #   return old contents of memory into reg
                            # do both things atomically

yield                       # switch to the next thread in the runqueue

nop                         # no op
```

Notes: - 'immediate' is something of the form `$number` - 'memory' is of the form

'number' or '(reg)' or 'number(reg)' or 'number(reg,reg)' or 'number(reg,reg,scale)' (as described above) - 'register' is one of %ax, %bx, %cx, %dx, %ex, %fx, %sp

Finally, here are the full set of options to the simulator are available with the `-h` flag:

```
prompt> ./x86.py -h

Usage: x86.py [options]

Options:
  -s SEED, --seed=SEED  the random seed
  -t NUMTHREADS, --threads=NUMTHREADS
                        number of threads
  -p PROGFILE, --program=PROGFILE
                        source program (in .s)
  -i INTFREQ, --interrupt=INTFREQ
                        interrupt frequency
  -P PROCSCHED, --procsched=PROCSCHED
                        control exactly which thread runs when
  -r, --randints        if interrupts are random
  -a ARGV, --argv=ARGV  comma-separated per-thread args (e.g., ax=1,ax=2 sets
                        thread 0 ax reg to 1 and thread 1 ax reg to 2);
                        specify multiple regs per thread via colon-separated
                        list (e.g., ax=1:bx=2,cx=3 sets thread 0 ax and bx and
                        just cx for thread 1)
  -L LOADADDR, --loadaddr=LOADADDR
                        address where to load code
  -m MEMSIZE, --memsize=MEMSIZE
                        size of address space (KB)
  -M MEMTRACE, --memtrace=MEMTRACE
                        comma-separated list of addrs to trace (e.g.,
                        20000,20001)
  -R REGTRACE, --regtrace=REGTRACE
                        comma-separated list of regs to trace (e.g.,
                        ax,bx,cx,dx)
  -C, --cctrace         should we trace condition codes
  -S, --printstats      print some extra stats
  -v, --verbose         print some extra info
  -H HEADERCOUNT, --headercount=HEADERCOUNT
                        how often to print a row header
  -c, --compute         compute answers for me
```

Most are obvious. Usage of `-r` turns on a random interrupter (from 1 to intfreq as specified by `-i`), which can make for more fun during homework problems.

- `-P` lets you specify exactly which threads run when; e.g., 11000 would run thread 1 for 2 instructions, then thread 0 for 3, then repeat
- `-L` specifies where in the address space to load the code.
- `-m` specified the size of the address space (in KB).
- `-S` prints some extra stats
- `-c` lets you see the values of the traced registers or memory values (otherwise they show up as question marks)
- `-H` lets you specify how often to print a row header (useful for long traces)

Now you have the basics in place; read the questions at the end of the chapter to study this race condition and related issues in more depth.