This homework covers OSTEP chapters 26 and 27. It is due in class Friday, April 3. 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 directories `threads-intro` and `threads-api` (and their contents) from `/classes/cs331` to your `~/cs331` directory. (You won't be writing any code — just a couple of commenting/uncommenting steps — so the copied directories don't need to go into your `workspace` directory.)

The `threads-intro` 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.

The `threads-api` directory contains several C programs and a `README` with instructions. You can also find the text of the `README` in the appendix section B at the end of this document.

Exercises

Chapter 26 homework problems (at the very end of chapter 26) —

1. Do #1–8. Read through the README to understand the code in `loop.s` and the other programs. For each exercise, try to predict what the value of interest will be at each step before checking with `-c`. There is nothing to hand in for #1 and #4. For the other problems, hand in your answers to the questions posed and explain your answers — don't just say "yes" or "not".

Chapter 27 homework problems (at the very end of chapter 27) —

Note that you do *not* need to download and install `valgrind` and `helgrind` — they are available in the Linux VDI.
Also note that you need to include the path for the programs run through `helgrind` e.g. for #1, the command line should be

```
valgrind --tool=helgrind ./main-race
```

(That's `./main-race` instead of `main-race`.)

2. Do #1–7. Specific notes:

   - For #2, remove one of the offending lines by commenting it out. (Restore it before trying the locks part.) To add the lock(s), uncomment the relevant lines — you don't need to write new code for the locks. Don't forget to recompile with `make` each time you make a change.
   - For #3, explain what causes the deadlock.
   - When asked what `helgrind` reports in #4 and #7, explain the output — don't just copy and paste the output.

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.

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 forms 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
jgte .top
halt
```

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:

```
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
   -1   1  0  1  0  0  1  1000 sub  $1,%dx
   -1   0  0  1  1  1  0  1001 test $0,%dx
   -1   0  0  1  1  1  0  1002 jgte .top
   -1   0  0  1  1  1  0  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:

```
prompt> ./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, and the PC. In this version, there is no support for a "stack", nor are there call and return instructions.

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

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

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

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)' (as described above) - 'register' is one of %ax, %bx, %cx, %dx

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

```
Usage: x86.py [options]

Options:
  -h, --help            show this help message and exit
  -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
  -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
-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.

- `-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` is not really used (unlike most simulators in the book); use the tracing or condition codes.

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.

B threads-api

In this homework, you'll use a real tool on Linux to find problems in multi-threaded code. The tool is called `helgrind` (available as part of the valgrind suite of debugging tools).

See `http://valgrind.org/docs/manual/hg-manual.htm` for details about the tool, including how to download and install it (if it's not already on your Linux system).

You'll then look at a number of multi-threaded C programs to see how you can use the tool to debug problematic threaded code.

First things first: download and install `valgrind` and the related `helgrind` tool.

Then, type `make` to build all the different programs. Examine the `Makefile` for more details on how that works.

Then, you have a few different C programs to look at: - `main-race.c`: A simple race condition - `main-deadlock.c`: A simple deadlock - `main-deadlock-global.c`: A solution to the deadlock problem - `main-signal.c`: A simple child/parent signaling example - `main-signal-cv.c`: A more efficient signaling via condition variables - `common_threads.h`: Header file with wrappers to make code check errors and be more readable

With these programs, you can now answer the questions in the textbook.