Operating Systems, Assignment 3

This assignment includes two programming problems. In general, we will compile your C programs using gcc and the following command-line options. Some programs require additional options as noted below.

```
gcc -Wall -g -std=c99 -o program program.c -lpthread
```

Also, remember that your programs need to be commented and consistently indented, and that you **must** document your sources and write at least half your code yourself from scratch.

1. (12 pts) Imagine that you have four semaphores initialized as follows:

```
sem w = 0;
sem x = 0;
sem y = 1;
sem z = 1;
```

Each of the following functions will be run concurrently by a different thread:

```
void thread1()        void thread2()        void thread3()        void thread4()
{                     {                     {                     {
  acquire( w );         release( w );         print( "g" );         acquire( w );
  print( "a" );         print( "d" );         acquire( z );         print( "j" );
  acquire( y );         release( x );         print( "h" );         release( y );
  print( "b" );         release( z );         release( z );         acquire( x );
  release( y );         print( "e" );         print( "i" );         print( "k" );
  print( "c" );         acquire( z );         release( w );         release( y );
  acquire( y );         print( "f" );       }                       print( "l" );
}                     }                                           }
```

Of the following strings, which ones could this code print out (if a different thread was running each function)? Write up your answers in a file, and mark each string with “yes” if it could be printed or “no” if it couldn’t be printed. There’s a starter file named **orders.txt** posted on course Moodle site. It has copies of each of the strings, in the same order as below. Just put a “yes” or “no” after each string to indicate your answer. Convert and submit your file as a PDF file called **hw3_p1.pdf** under the assignment named HW3_Q1 on Gradescope.

(a) `gabcdhefijkl`

(b) `abcdefgjklhi`

(c) `abcdefghijkl`

(d) `gjabchdiefkl`

(e) `deagjbcfklhi`

(f) `jgdhiefkablc`

2. (40 pts) For this problem, you're going to solve the max-sum problem we've done on homework Assignments 1 and 2. The input and output formats will be the same as last time, but we'll use a different technique for reading the input and distributing work. This time, you're going to solve it in C, using multiple worker threads to find the contiguous non-empty subsequence within the sequence of integers that has the largest sum. However, instead of dividing up the work statically (where each worker is always assigned the same subtasks of the problem), we'll divide up the work dynamically (so different workers may get different subtasks from execution to execution). You'll use anonymous, POSIX semaphores for synchronization. I'm giving you a skeleton to get you started, `maxsum-sem.c`, posted on the course homepage.

   As on the previous assignments, there will be one command-line argument giving the number of worker threads to use and an optional "report" argument that will get the workers to report the maximum sum they find from their part of the work. The sequence of values will still be read from standard input.

   Instead of reading all the input at the start, this version will start the workers first and then use the main thread to read the list of values. Initially, workers won't have anything to do. However, they'll use semaphore-based synchronization code to allocate more and more work to themselves as the main thread reads the input. If there's currently no work to give out, idle workers will have to wait for the main thread to read in more. This will continue until the main thread finishes reading all the input. Then, once all the work has been finished, the worker threads can terminate and the main thread can report the final maximum sum by comparing the maximum sums found by the workers from their part of the work. Compared to our previous versions, this would be a good approach if reading the input was really slow or if the whole input list wasn't available at program startup (say, if the input was being downloaded or if it was the result of some other computation). Workers can get started on the portion of the list read in so far and then start working on more input as it is read in.

   Being able to use semaphores will make it a lot easier to update the final maximum sum across all the workers. We can just use a global variable. When the workers are done, they can compare the largest sum they found to this global and update it if their finding is larger. Since every worker will be modifying this variable, we'll need to use semaphore-based mutual exclusion code to make sure two of them don't try to modify it at the same time (this is unlikely, but it's also really easy to prevent). After all the workers have finished, the main thread can just look at the value of this global variable to report the final maximum sum.

   The tricky part of this program will be assigning work to workers as they finish what they were previously working on.This is really an example of the producer/consumer problem. The main thread reading the input is like the producer, and each of the workers is like a consumer, getting more work as it becomes available and waiting if all the currently available work has already been given out.

   Like on the previous versions of this program, you can decide how you want to split up work and give it out to the workers. You could consider assigning a different worker to each element of the sequence. That worker could check all the ranges that end to that element. This approach would have the advantage that, if a worker is given responsibility for some value x_i, then if x_i has been read in it means that all the previous values in the sequence will already have been read in. You can divide up the work however you want, but remember that a worker can't use values that haven't been read in yet.

   In my own solution, I put much of the logic for giving out work in a function called getWork(). Workers call this function when they become idle. It returns immediately if there's more to work on, or, if all available work has already been given out, the function blocks the worker until more input has been read by the main thread. My getWork() function returns the index of the value the worker is responsible for. Your solution may do something like this, or it could return something else, depending on how you divide up the work. If we think of the threads like producers and consumers, the getWork() function is kind of like the code the consumer might use to get the next item from the buffer. If there's more work to be done, it gets it; if not, it waits.

After we've read all the input and given out all the work, my getWork() function starts returning a special sentinel value to the workers. This indicates that there's no more work to do. Once all the work has been given out, it's like the main thread (the producer) starts putting dummy values (sentinels) into the buffer. When the workers get the dummy values, they know it's time to exit. You're not required to implement your solution this way, but this isn't a bad way to think about it, and it may help you simplify your code.

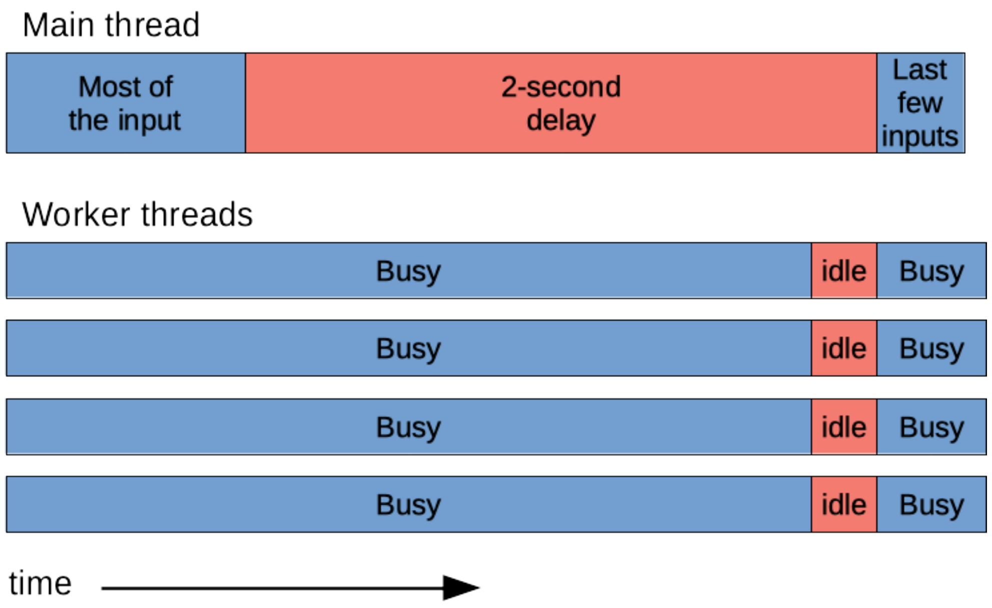
Once your program is working, you should be able to run it like the following to have it process one of the input files from the previous assignments. As before, you should get speedup as you run your solution on the largest input file with more than one worker (provided you have more than one core). In my own solution, I get almost 4 times speedup running with four workers on a system with at least four cores. Your performance results might be a little bit noisy, depending on what system you're running on. To get a good idea of your performance, you may want to run a few times and check the median.

```
$ time ./maxsum-sem 1 < input-5.txt
Maximum Sum: 227655

real    0m15.086s
user    0m15.099s
sys     0m0.003s

$ time ./maxsum-sem 4 < input-5.txt
Maximum Sum: 227655

real    0m3.784s
user    0m15.118s
sys     0m0.014s
```

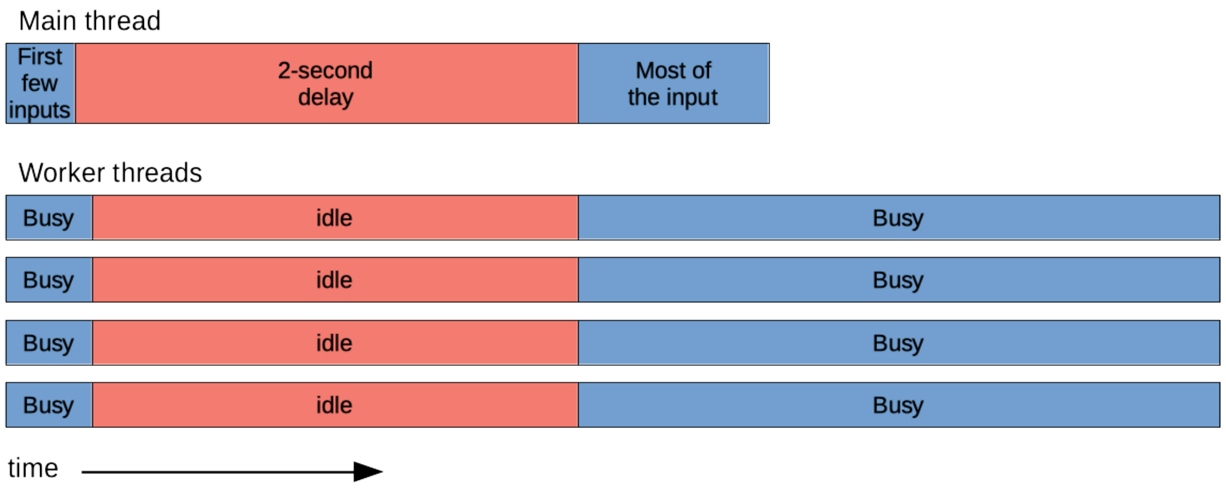
This solution should work just fine even if some of the input isn't there when the program first starts up. For more interesting results, try the following. Here, we're introducing some delay in delivery of the input. The comments in the shell commands below tell you what we're doing and what you should expect. In the examples below, I used a two-second delay in the delivery of the input, since my program took about four seconds on the largest input. You could try a longer delay if your program isn't as fast on this test case.

```
# Here, we're using the sed (stream editor) command to break the input
# into two parts.  We're giving most of the input (all but the last
# 5000 values) all at once at the start, then we give the remaining
# input after a 2-second delay.  The big batch of initial input gives
# the workers plenty to work on at the start.  So, they can stay busy
# while the main thread is waiting 2 seconds to get the last, little
# batch of input.  When those last 5000 values arrive, the workers can
# finish up the rest of the problem quickly.  The workers will
# probably stay busy all the time and the elapsed time will be about
# the same as when we got the inputs all at once.

$ (sed -n '1,95000p' input-5.txt; sleep 2; sed -n '95001,$p' input-5.txt) | time -p ./maxsum-sem 4
Maximum Sum: 227655
```

```
real 3.78
user 15.11
sys 0.02
```



```
# Below, we're giving the program the same input, but we're breaking
# it into a little 5000-element section, followed by the remaining
# 95000 elements after a 2-second delay. The program should be able to
# handle this, but it should be about 2 seconds slower than the
# previous example (measuring elapsed real time). Processing the first
# 5000 elements of the list takes very little time.  The workers will
# finish this quickly.  Then, the main thread and the workers will all
# have to wait 2 seconds to get the bulk of the work, the 95000
# remaining  values.  When they get the rest of the input file, it is
# still going to take them a while to process it.  So, the elapsed
# time will go up by about 2 seconds, because there wasn't enough work
# to keep the workers busy for the 2-second delay near the start.

$ (sed -n '1,5000p' input-5.txt; sleep 2; sed -n '5001,$p' input-5.txt) | time -p ./maxsum-sem 4
Maximum Sum: 227655
real 5.77
user 15.11
sys 0.00
```



When you're done, submit your source file, `maxsum-sem.c` under the assignment named HW3_Q2 on Gradescope.

3. (40 pts) Our shared-memory `reset.c / lightsout.c` program from homework assignment 2 had a race condition. If two users ran the lightsout.c program at the same time, they might both try to modify the GameState struct at the same time, potentially leaving it in an inconsistent state. Of course, if we tried to demonstrate this race condition, we'd have a hard time doing so by hand. It's very unlikely we would be able to run two copies of lightsout.c close enough together that they would actually interfere with each other.

   For this problem, you're going to do two important things. You're going to add mutual exclusion code to lightsout.c (and a little bit to reset.c) in order to eliminate the possibility of race conditions. You're also going to add a test interface, to simulate running the program over and over again very, very quickly. You're going to wrap conditional compilation directives around your mutual exclusion code so you can easily enable or disable mutual exclusion at compile time. With the test interface and the compile-time option for turning the mutual inclusion code on and off, we should be able to see the race condition actually cause an error (hopefully), and verify that enabling the mutual exclusion code prevents the error.

   Use a POSIX named semaphore to provide mutual exclusion. Using a named semaphore will let multiple copies of the lightsout.c program access the same semaphore. When a process running lightsout.c needs to use the shared GameState struct, it will acquire the semaphore to make sure no other process is working with the data structure at the same time. When it's done, it will release the semaphore so other processes can access the structure.

   The reset.c program will now need to create both the shared memory segment and the named semaphore (initialized to 1). The shared memory and the semaphore will continue to exist across multiple executions of `lightsout.c`. In lightsout.c you should be able to shmget() and sem_open() to get the shared memory and the semaphore, after reset.c has created them.

   We will all need to use different names for our semaphores, to make sure the programs belonging to one student don't interfere with those of other students, even if they happen to run on the same host. Use `/unityid-lightsout-lock` as the name of your semaphore (where unityid is your actual unity id).

   In addition to the move, undo and report commands from the previous assignments, your program needs to support a new command that gives access to the test interface:

   - `./lightsout test` *n r c*

Running lightsout.c like this will execute the `move` command n times as quickly as possible, making a move at row r, column c. The test command won't print anything; that would just slow it down. We want it to be able to exercise the GameState data structure as quickly as possible to help us check for a possible race condition.

To use the `test` command, we will run two copies of our program at once, each one running a test with a large value of n. Without mutual exclusion code to protect access to the shared memory, two programs trying to rapidly modify the GameState at the same time might leave it in an inconsistent state. We'll be able to check this by enabling or disabling the mutual exclusion code during compilation.

For the previous assignment, you may have written your program with all of the code in main(). For this assignment, we will need to organize the code a bit, so that the mutual exclusion code is testable. Each of your commands will be implemented in a function like the following. A pointer to the shared-memory GameState is passed to each function, and (for move and undo) the return value says whether or not the function was successful so the caller can print an error message if needed.

```
// Make a move at the given row, column location, returning true
// if successful.
bool move( GameState *state, int r, int c );

// Undo the most recent move, returning true if successful.
bool undo( GameState *state );

// Print the current state of the board.
void report( GameState *state );
```

After parsing the command-line arguments, your main function will call one of these functions to access the GameState. Inside each of these functions, you will acquire on your semaphore at the start and then release the semaphore before returning. If one of these functions returns from multiple places, be sure to release the semaphore before returning via any of these paths. **Don't** just lock the semaphore in your main and then unlock it before main exits. That will prevent the test function from working properly.

On the test command, your program will just call a function like the following. After checking the board location, it just calls the move() function over and over. You can see, test() doesn't acquire the semaphore itself; that's done inside the move() functions it calls.

```
// Test interface, for quickly making a given move over and over.
bool test( GameState *state, int n, int r, int c ) {
  // Make sure the row / column is valid.
  if ( r < 0 || r >= GRID_SIZE || c < 0 || c >= GRID_SIZE )
    return false;

  // Make the same move a bunch of times.
  for ( int i = 0; i < n; i++ )
    move( state, r, c );

  return true;
}
```

All of your calls to acquire and release semaphores will be wrapped in conditional compilation code like the following. This will let us disable the mutual exclusion by defining the preprocessor constant, UNSAFE, on the gcc line. You need to use this macro name, UNSAFE, exactly like this. That's part of how we'll test your program, by using this symbol to enable or disable your mutual exclusion code.

```
#ifndef UNSAFE
  sem_wait( mySemaphore );
#endif
```

Once your program is working, you should be able to run it just like you did in the previous assignment. You should also be able to test your program with and without the mutual exclusion code enabled. Of course, your program should work just fine with mutual exclusion, but it may (probably, hopefully) fail if you run two tests concurrently without mutual exclusion and with large enough values for n. Race conditions are more likely to appear on a multi-core system. For this program, you're unlikely to see them on a single-core machine (but, maybe).

The following is a possible execution of your program. All the commands from assignment 2 should still work, but here I'm trying to show the test interface. I've included some shell comments to explain what I'm doing.

```
# Build the reset program
gcc -g -Wall -std=c99 -D_XOPEN_SOURCE reset.c -o reset -lpthread

# Build the lightsout program with mutual exclusion enabled.
gcc -g -Wall -std=c99 -D_XOPEN_SOURCE lightsout.c -o lightsout -lpthread

# Reset to a blank board.
./reset board-2.txt

# Run two copies of the test at the same time.  The first one is run
# in the background (with the &) and the other immediately afterward
# in the foreground.  This will take a few seconds to run, and the
# value of n is large enough that both copies should have had a chance
# to run at the time.  With the first command run in the background,
# you'll eventually get a report from the shell that the backgrounded
# command is done.  That's OK; the shell does this for any commands
# started in the background.
./lightsout test 5000000 2 1 & ./lightsout test 5000000 2 1

# After running two copies of the test, we can check the final state
# of the board.  Making the same move over and over an even number of
# times should leave us with a blank board, just like we started with:
./lightsout report
.....
.....
.....
.....
.....

# Now, compile again, with mutual exclusion disabled.
gcc -g -Wall -std=c99 -DUNSAFE -D_XOPEN_SOURCE lightsout.c -o lightsout -lpthread

# Run the same test with mutual exclusion code compiled out.  It
# should run faster now because of reduced overhead from the missing
# synchronization code.  But, there's a race condition.  It's even
# possible running this test will crash your program, depending on how
# you implemented your program.
./lightsout test 5000000 2 1 & ./lightsout test 5000000 2 1

# Now, see what happened to the board.  I get a few lights that are
# left on.  This is because of the race condition between multiple
# concurrent clients.  If I reset and run again, I might get a
# different result.
./lightsout report
.....
.*...
..*..
.....
.....
```

Your program will use the same shared memory and named semaphore each time you run it. If either of these gets left in an unknown state, you may need to delete them and then use reset.c to re-create them in a known state. As in the previous assignment, you can see your shared memory segment by running `ipcs` and you can delete it by entering `ipcrm -m` along with the shared memory ID reported by ipcs. On a Linux machine, you should be able to see your semaphore by entering `ls /dev/shm`. You can remove it by just using the `rm` command to delete the file under `/dev/shm`, just like you would remove any other file.

When you're done, submit your three source files under the assignment named HW3_Q3 on Gradescope. This will include your source for **common.h**, **reset.c** and **lightsout.c**.