Operating Systems, Assignment 2

This assignment includes two programming problems. The `-lpthread` flag is for using pthread in problem 2. The define for `_XOPEN_SOURCE` is for the shared memory calls used in problem 3.

```
gcc -Wall -g -std=c99 -D_XOPEN_SOURCE=500 -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. (16 pts) For this problem, we're going to look at how different schedulers behave and how this can affect performance. Pretend the following table describes the next CPU burst for a collection of processes. The table also gives the time at which the process arrives in the ready queue and the process' priority values (lower values representing higher priority, like we did in class).

Process	Burst Length	Arrival Time	Priority
P_1	8	0	5
P_2	5	4	7
P_3	3	9	9
P_4	9	12	2
P_5	10	15	6
P_6	6	18	8

For each of the following scheduling algorithms, draw a Gantt chart like the ones in the CPU scheduling chapter of your textbook. Also, for each schedule, compute the average waiting time and the average turnaround time, where turnaround time is defined as the time from when a process arrives in the ready queue to when if finishes its CPU burst.

(a) SJF
(b) SRTF
(c) Preemptive Priority
(d) Round Robin with time quantum = 6

For SRTF, it's possible for there to be a tie between the remaining burst time of the currently running process and a newly arrived process (we saw this in our in-class example). If this happens, just let the currently running process continue running (so, all else being equal, avoiding a context switch).

For round robin, it's possible for there to be a tie between a process entering the ready queue and another process being preempted and returning to the ready queue (we saw this in our in-class example). If this happens, put the preempted process behind the entering process in the queue (so the newly arrived process will get to run earlier).

You may draw your Gantt charts by hand and scan them in if you'd like. Just make sure your drawing is clear enough to read easily. Alternatively, you may draw them using the drawing program of your choice. Either way, submit your charts as a PDF file called **hw2_p1.pdf** under the assignment named HW2_Q1 on Gradescope. Please also report waiting and turnaround time in your hw2_p1.pdf file. Be sure to include labels in your file, so we can tell what times you're reporting.

2. (40 pts) For this problem, you're going to write a multi-threaded program, `maxsum-thread.c`, that does the same job as the `maxsum.c` problem from homework assignment 1. Like the previous program, your `maxsum-thread.c` will take a command-line argument giving the number of workers to use to find the contiguous non-empty subsequence within the sequence of integers that has the largest sum. It

will also take an optional argument, `report`, that tells it to report the largest sum each worker finds from its own part of the work (just like in the previous assignment). It will read the input sequence from standard input, just like in the previous assignment.

This time we'll use threads as workers instead of separate processes. This will simplify our implementation a bit, since we won't have to use message passing to communicate results from the workers. Threads automatically share memory, so we can let the main thread[1] get each worker's result by just leaving it in a field in the argument structure passed to each worker thread.

After reading the input sequence, the main thread will create the requested number of worker threads. The workers will work in parallel to find the largest sum, with each worker examining just some of the possible ranges. You can divide up the work however you want, but you should divide it evenly enough that you get speedup with multiple workers on a multi-core system. You may not get perfect linear speedup, but you should be able to see some speedup with more workers, until you have a worker for every CPU core.

The main thread will probably need to pass a value to each worker, so it will know what part of the problem it's supposed to work on. Then, after starting all the workers, the main thread will wait for them to terminate and then compare their results to report the final maximum sum. To do this , it will need to get some result value from each of the worker threads. You can handle passing values to the new threads and getting their results by having an extra field in the argument structure passed to each thread.

Once you complete your program, you should be able to run it as follows. You can test your solution with the same input files we used on assignment 1. Just like with the previous assignment, you may get different orders from execution to execution; that's OK. Note that if the report option is given, each thread will report its thread id (tid) instead of pid.

```
$ ./maxsum-thread 5 report < input-3.txt
I'm thread 4920. The maximum sum I found is 145.
I'm thread 4921. The maximum sum I found is 153.
I'm thread 4922. The maximum sum I found is 159.
I'm thread 4923. The maximum sum I found is 170.
I'm thread 4924. The maximum sum I found is 152.
Maximum Sum: 170
```

Feel free to try it out on an EOS Linux machine or on your own system. If you own a system with lots of cores, you may be able to see lots of speedup. Be sure to check the feedback from the auto-grader on Gradescope at least once before your final submission. Although your code will probably run on just about any system, we'll still be testing it on Gradescope.

When you're done, submit your source file, `maxsum-thread.c` under the assignment named HW2_Q2 on Gradescope.

3. (40 pts) For this problem, you're going provide similar functionality to the client/server program from homework assignment 1. This time, instead of using a server and interprocess communication to maintain the state of the game, you will store information about the game board and the most recent move in a shared memory segment. Any program that needs to access the game can get and attach the shared memory segment. Then it can look at or modify the state of the game just like any other data structure that's part of its memory.

   You will still need two different programs, but they won't be doing the same jobs as before. One of your programs will be called `reset.c`. Its job will be to read the initial game board state just like

[1]There's not really a "main" thread. Here, I just mean the thread that started running in main, the one that's not a worker thread.

the server did in the previous assignment and create a shared memory segment containing any needed information about the game. The `reset.c` program will just exit after creating the shared memory segment and initializing it based on the board description file.

The other program will be called `lightsout.c`. It will interpret a user command given in the command-line arguments and make requested changes to the game board stored in shared memory. You can also use a header file named `common.h` that can be included by both of your programs.

Your `lightsout.c` program will be used kind of like the client program from the last assignment. The user will run it with command-line arguments that say what to do in the game. But, instead of having to send requests to a separate server program, it will just directly access and modify the state of the game stored in the shared memory segment. Then, other copies of the `lightsout.c` program run on the same host will automatically see the changes to the game.

You should be able to run the `lightsout.c` program in the following ways (just like you could run the client in homework 1):

- `./lightsout move` *r c*
  Running the lightsout program like this will make a move at row *r*, column *c*. The row and column parameters and valid moves are defined just like in the previous assignment. The program will make the requested move if it's valid and print out either "success" or "error", just like the client did in assignment 1.
- `./lightsout undo`
  Running the lightsout program like this will undo the most recent move. It will print out either "success" or "error", to indicate whether it was successful (i.e., error if there was no move to undo).
- `./lightsout report`
  Running the lightsout program like this will print a 5×5 square of characters showing the current state of the board.

Starter Files

The course website has starter files for your three source files. They are mostly empty, but they include a few functions for reporting errors and probably all of the includes you will need.

Error Conditions

The `reset.c` program should expect a single command-line argument giving the initial state of the game board, just like the server did in the previous assignment. If the command line arguments are invalid, it should print the following usage message to standard error and then exit unsuccessfully.

```
usage: reset <board-file>
```

If the given board file is missing or isn't valid, it should print the following message to standard error (where filename is the name of the board file given on the command line) and exit unsuccessfully.

```
Invalid input file: filename
```

For both the `reset.c` and `lightsout.c` program, if an error is encountered in one of the system calls, (like not being able to create or attach its shared memory), you should print out a meaningful error message of your choice and then exit. These messages will also help you diagnose problems while you're developing your program.

Creating the Shared Memory

For this problem, you'll need to organize all the state information for your board and the most recent move into a single struct. We'll call this struct the GameState. Having the whole representation in this one struct is going make it easy to store in shared memory. To create the shared memory, we just need to ask for a segment that's the size of an instance of GameState.

When you attach the shared memory, you can just cast the pointer you get back from shmat() to a pointer to GameState. Then, you can easily access the contents of the shared memory via this pointer, as if the shared memory was just an instance of GameState. Notice, the way we're using shared memory is a lot like how we use malloc(); we tell shmget() how much memory we need to store a GameState structure, then we use the pointer returned by shmat() as a pointer to a new instance of GameState.

The shared memory segment will persist across multiple executions of our programs. The first time you run reset.c, it will create the shared memory and fill it in based on the given board file. After this, running lightsout.c will get and attach that shared memory segment (without creating a new one). It will be able to look at the GameState struct created by reset.c and change the board representation or access the most recent move as needed, based on the command-line arguments given by the user.

The shared memory segment will let multiple programs access the list at the same time[2]. Like with our client server, we should be able to open multiple terminals on the same host and access the same game board from any of the terminals.

Sample Execution

Once your program is working, you should be able to run it as demonstrated below. Here, I'm showing how you can access the shared game board from multiple terminals running on the same host. The commands on the left are from one terminal session and those on the right are from a different terminal. I've ordered the commands top-to-bottom to show the order I ran them. For example, when we make a move in the right-hand terminal, we can see the result with the report command in the left-hand terminal. The shell comments mixed in with the commands just give some explanation of what this example is doing (you don't need to enter them).

```
# Set up an initial game board.
$ ./reset board-3.txt

# Get a look at the game board.
$ ./lightsout report
....*
..**.
*.**.
*.**.
*.*..

                               # make a move
                               $ ./lightsout move 1 1
                               success

# See the result of the move made in the other termial.
$ ./lightsout report
.*..*
```

[2]If you really ran two copies of lightsout.c at the same time, there's the potential for a race condition. If they both tried to modify the contents of shared memory at the same time, they might interfere with each other and leave it in an unknown state. We'll just ignore this potential problem for this assignment. Maybe we'll fix it on assignment 3.

```
**.*.
****.
*.**.
*.*..

                               # Make another move and have a
                               # look at the result.
                               $ ./lightsout move 0 4
                               success

                               $ ./lightsout report
                               .*.*.
                               **.**
                               ****.
                               *.**.
                               *.*..

# Undo the move from the other termianl and see the result.
$ ./lightsout undo
success
$ ./lightsout report
.*..*
**.*.
****.
*.**.
*.*..

# Try a few invalid commands.
$ ./lightsout undo
error

                               $ ./lightsout move 5 5
                               error
                               $ ./lightsout a b c
                               error

$ ./reset board-4.txt
Invalid input file: board-4.txt
$ ./reset
usage: reset <board-file>
$ ./reset bad-filename
Invalid input file: bad-filename
```

Naming Your Shared Memory

You will need your own key (number) to create a shared memory segment. For our in-class example, I used a hard-coded key I just made up, 9876, but we can't all use this key if some of us might be developing on the same, shared systems. Instead, let's use the ftok() function to assign different keys to each member of the class. This function maps a pathname in the filesystem to a unique key. Just use the name of your EOS home directory as the path, and you should get a key that's unique to you.[3]

Resetting Your Shared Memory

Your lightsout.c program will use the same segment of shared memory every time you run it. If the contents of this memory get corrupted (say, because of a bug while you're developing your program), you will need to reset the contents of this memory. Your reset.c program will make this easy. You should be able to just re-run reset whenever you need to erase the old GameState data and overwrite it with a new one.

If you need to completely delete your shared memory segment so reset.c can make a new one, there are a couple of command-line utilities to help you out. The `ipcs` shell command will list all your shared memory segments (along with some other IPC objects). You can delete a shared memory segment by running `ipcrm` with the `-m` option followed by the id of the segment you want to delete. `ipcs` reports the ids for each segment, along with the segment size. If you own multiple shared memory segments (maybe created by other programs begin run by you), you can use the size to help you decide which one to delete. When I tried this, the shared memory segment for my GameState was much smaller than the other segments owned by me.

Accessing Shared Memory

Like the message queues we used on assignment 1, shared memory only works for programs running on the same host. If you access the shared memory from two different terminals (like in the example) you need to make sure terminals are logged in on the same host. If you're using remote.eos.ncsu.edu, use the hostname command to figure out what machine you're logged in on and login on that particular host for any other terminals you're using. Or, if you use a Linux desktop machine, it'll be easy to open as many terminals as you want on the host you're sitting in front of.

Submitting Your Work

When you're done with your program, submit three source files. This should include your common header file, **common.h**, the implementation file for the reset program, **reset.c** and the implementation file for the lightsout program, **lightsout.c**. Submit all of these to the assignment named HW2_Q3 on Gradescope.

[3] It's important that you get this path right, or the TA may not be able to grade your work. If you're not sure what your home directory is on EOS, you can enter `echo $HOME` on an EOS Linux machine. Also, for ftok(), there's sometimes some confusion about what to do with the proj_id parameter. It's just there to let you get multiple unique keys based on the same pathname. Since you just need one key, you can use any constant you want for this parameter, but be sure to use the same constant every time in order to get the same key.