Operating Systems, Assignment 1

This assignment includes a few programming problems. When grading, we will compile and test your programs on Gradescope. In general, we'll compile with options like the following. Some programs require additional options. These are described with the programming problem when they're needed.

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

If you develop a program on a different system, you'll want to make sure it compiles and runs correctly on Gradescope - you can submit your code unlimited amount of times before the deadline to see the feedback from the auto-grader, and after the deadline, we will only grade your last submission. Please note that auto-grader can only help you test the given testing cases from the specification, but you're responsible to fully test your programs to make sure they satisfy all the requirements. We will add extra testing cases when we grade your work. 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) Log in to one of the EOS Linux machines, use the online manual pages to answer each of the following questions about system calls on Linux. As with the first assignment, be sure to look at the section of the online manual for system calls. Sometimes there are library functions or shell commands with the same name.

   Write your answers into a text file, convert it to a pdf file called `hw1_p1.pdf` and submit it to HW1_Q1 on Gradescope.

   (a) Let's say you need to figure out how large a file is. You can do this by opening the file with fopen(), then reading up to the EOF using getchar(). If you count all the bytes you read, you can determine the file size.

   A much better way to figure out file size is using the stat() system call. Describe how you could use stat to determine the size of a file.

   (b) Normally, you expect to use a blocking interface for reading and writing a file. However, the OS will let you use a non-blocking interface for file I/O, if that's what you want. When you open a file, you can ask the OS to send a signal to your process whenever the file becomes readable or writable. What do you need to do to open a file this way.

   (c) An operating system will normally keeps track of resource usage and other statistics for each of its processes. On a Unix system, a process can use the getrusage() system call to check this. A call to getrusage() fills in the fields of a struct, reporting various details about the process' execution. Using this struct, how could a process determine the number of times it has been in the running state?

   (d) Lots of system calls will return immediately if a signal is delivered while the process is waiting in the system call. For example, mq_receive() is like this. If a signal is delivered during a call to mq_receive(), how can a program tell that mq_receive() returned because of a signal rather than the arrival of a message?

2. (10 pts) Provide brief answers to the following questions. Write your answers into a text file, convert it to a pdf file called `hw1_p2.pdf` and submit it to HW1_Q2 on Gradescope. You don't have to write code for these questions; just answer each question in your own words. These should be good preparation for the types of questions you might see on an exam.

   (a) (2.5 pts) What are the steps involved in a context switch between two different processes?

   (b) (2.5 pts) What are the steps involved in a context switch between two different threads in the same process?

(c) (5 pts) Assume that the following two functions will be executed concurrently on two different threads (one thread will call p(), and another will call q()). Pretend that the individual statements in p() and q(), A, B, C and D, will each execute atomically. For example, the execution of A in p() can occur either before or after the execution of C in q(), but it can't occur at the same time.

```
void p()
{
  A;

  B;
}
```

```
void q()
{
  C;

  D;
}
```

List all the possible interleavings of the execution of statements in these two functions (one possible execution per line in your answer). For each possible interleaving, give a trace string showing the order in which these atomic statements are executed. For example, the string "ABCD" gives one possible execution order.

3. (40 pts) For this problem, you're going to write a program named `maxsum.c`. It will be able to use multiple processes and multiple CPU cores to solve a computationally expensive problem more quickly. The task is simple. You're given a sequence of positive and negative integer values, like the one shown below.

2	3	-2	4	1	7

Your job is to find a contiguous non-empty subsequence within the sequence that has the largest sum. For example, the largest sum we can get from the sequence above is 15, which is obtained by adding the values from index 0 to index 5.

Program Input

Your program will read input from standard input. As you can see in the examples below, we will use input redirection to get it to read input from a file rather than from the terminal. Input will be a sequence of up to 1,000,000 integer values, one value per input line. The following shows the contents of the first sample input file, `input-1.txt`. It contains the same sequence shown above.

```
2
3
-2
4
1
7
```

Program Output

Your program's last line of output should look like the following, where n is the maximum sum. For the example above, this would be 15.

Maximum Sum: n

Command-line Arguments

Your program will take up to two command-line arguments. The first argument gives the number of worker processes to use (see below).

Your program will take the word "report" as an optional, second command line argument. If the report option is given, then each child process will output its pid and the maximum sum it finds before it's done. So the program would report it as the following if the report flag was given on the command line, where xx is the pid number of the child process, and n is the maximum sum that child process found:

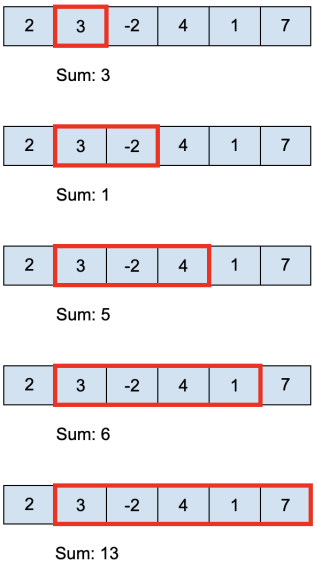
```
I'm process xx. The maximum sum I found is n.
```

Since your program is expected to report the maximum sum each child process found, different executions may report them in different orders. The output order will depend on the timing of the execution and how you chose to implement your program. It's OK if the order varies, but the "Maximum Sum:" output should always be last.

Partial Implementation

You get a starter file to help you with your implementation. The starter includes code to parse the input sequence and the command-line arguments. The input is guaranteed to be in the format described above; you don't have to add code to detect and respond to bad input. Also, you're guaranteed that all ranges from the input sequence will add up to a number that fits in a signed, 32-bit integer. You don't need to worry about overflow for it.

You'll add code to look for ranges of values that add up to the largest sum. We will do this by checking every range of values, starting from index i and ending at some index j with $j \geq i$. You should be able to do this in $O(n^2)$ time, where n is the number of elements in the sequence. For checking the sum of values in a range, you can organize your code so you check all ranges that start in the same place one after another (or, alternatively, all ranges that end in the same place). The following figure illustrates how you can handle all the ranges that start at index 1.



Computing the sum of the range that starts and ends at index 1 takes constant time (it's just the value 3). Then, we can compute the sum of the sequence that starts at index 1 and ends at index 2 by

adding just one value, -2. Then, we can check the sum of the range of values from index 1 up to index 3 by adding just one value, 4. Each time we extend the sequence by one element, we just have to add in the value of that one element. So, checking the sum of a particular range of values should just take linear time.

Parallelization

With an $O(n^2)$ solution, it could take a long time to handle a large sequences of values. As you can see below, my solution took several seconds on a 100,000-element sequence. However, the $O(n^2)$ approach is easy to parallelize, dividing the work among multiple processes.

On the command-line, the user specifies how many child processes the program should create. We will call each child process a `worker`. After reading the input sequence, the parent will fork() to create each worker and then let the workers do all the real work of solving the problem.

Each worker will be responsible for checking some ranges of values from the sequence. You can divide up the work however you want, but try to divide it evenly so all the workers have a similar amount of work to do. For example, you could make each worker responsible for checking ranges based on where the range starts (or, alternatively, where the range ends). If you had 3 workers, you could make the first one responsible for checking ranges starting at index 0, index 3, index 6 and so on. The next worker could be responsible for ranges starting from index 1, index 4, index 7 ..... The last worker could be responsible for ranges starting at index 2, index 5, index 8 ..... This isn't the only way to divide up the work, but it's an easy technique to implement. Also, it lets each worker efficiently compute the sums in each range the way it's described above.

Each worker should count (and optionally report) the largest sum that it finds in its part of the problem. When they're done, the workers will send their largest sum back to their parent process, so it can compare and report the final maximum sum.

If the user asks your program to report, your worker processes will just print out the maximum sum they found before they're done. You don't have to report them in any particular order.

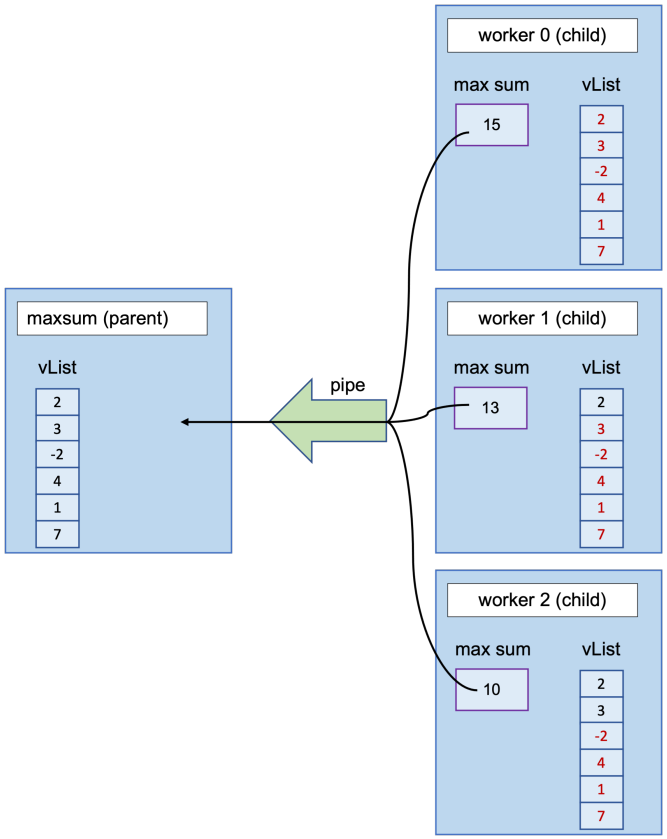
At the end, your parent process will print out the final maximum sum. This is going to require communication from the child processes to the parent (see below). Once the parent has compared the values and identified the largest sum, the parent should print out the final maximum sum.

Interprocess Communication

When the workers are created, they will automatically get a copy of the list of values read in at program start-up, since children get a copy of everything in the parent's memory. So, after creating the workers, the parent will just wait for the workers to find the maximum sums in their parts of the work.

When the workers are done, they need a way of sending their resulting local local largest sum back to their parent process. We'll use an anonymous pipe for this. Before creating its children, the parent will make a pipe. Then, when a child is done, it will write its local largest sum into the writing end of the pipe and the parent will be able to read it out of the other end.

We'll send this value in binary; that will be easier. Since communication is just going between processes on the same host, we don't have to worry about byte order or other architectural differences. Parent and child will always be running on the same hardware. To send the binary representation of an integer through a pipe, we just need to use the starting address of that integer as the buffer to send. The size of an integer is the number of bytes we want to send. On the receiving side, we can use a similar technique to read the contents of the integer into a variable of type int.



After reading the local largest sum from each child, the parent will compare among them and print out the final maximum sum at the end of execution. Before exiting, the parent should use `wait()` to wait for each of its children to terminate.

File Locking

There is a small risk that two workers will try to write their locally computed count into the pipe at the same time. For a moderate number of workers, I don't think this would really cause a problem, but, either way, it's easy to prevent. Before writing its result into the pipe, a worker will lock the pipe. This can be done with a call like the following.

```
lockf( fd, F_LOCK, 0 );
```

Then, after writing its value, a worker can let others use the pipe by making a call like:

```
lockf( fd, F_ULOCK, 0 );
```

This is an example of advisory file locking. If two workers try to lock the pipe at the same time, one of them will be granted the lock and the other will automatically wait until the first one unlocks it. As long as all our workers lock the pipe before using it, two of them won't be able to write into it at the same time.

Compiling

Using the lockf() call requires an extra preprocessor flag on the EOS and Gradescope Linux systems. You'll want to compile your program like the following. The `-lm` option links with the math library, in case you need to use the `sqrt()` function.

```
gcc -Wall -std=c99 -D_XOPEN_SOURCE=500 -g maxsum.c -o maxsum
```

Sample Execution

Once your program is working, you should be able to run it as follows. I've included some shell comments to help explain these examples (the lines starting with a pound sign). Of course, you don't need to type these in, but the shell should just ignore them if you do.

```
# Use one worker on the smallest test case.
$ ./maxsum 1 report < input-1.txt
I'm process 125820. The maximum sum I found is 15.
Maximum Sum: 15

# Try the next, larger test case using two workers.
# Your output may be in a different order, but the \Maximum Sum:" report
# should still be last.
$ ./maxsum 2 report < input-2.txt
I'm process 126659. The maximum sum I found is 33.
I'm process 126660. The maximum sum I found is 35.
Maximum Sum: 35

# Use four workers on input-3.txt (a 100-element sequence)
$ ./maxsum 4 report < input-3.txt
I'm process 127708. The maximum sum I found is 153.
I'm process 127709. The maximum sum I found is 159.
I'm process 127710. The maximum sum I found is 170.
I'm process 127711. The maximum sum I found is 152.
Maximum Sum: 170

# See how long it takes on input 5 with just one worker, a 100,000 -
# element sequence.  Here, we're not using the report option so it won't
# slow down the execution.  You may get different timing results than
# I do, depending on your implementation and the system you run
# your solution on.

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

real    0m14.165s
user    0m14.157s
sys     0m0.001s

# Try again with 4 workers.  This should take about the same amount of
# total CPU time, but, if you have multiple cores it should finish
# faster. With more than one CPU core, workers should be able to run in
```

```
# parallel, reducing the amount of time from start to finish.  Read
# the online manual page for the time command to make sure you understand
# what the time output means.

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

real    0m3.620s
user    0m14.261s
sys     0m0.003s
```

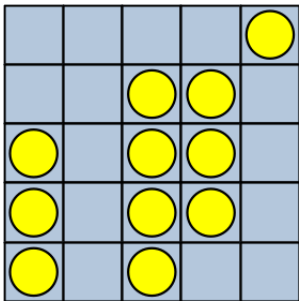
Be sure to try out your program with the time command on the largest input file, using different numbers of workers. As long as you have at least as many cores as workers, you should see speedup that's close to linear in the number of workers. So, if you use two workers, it should take about half as much time as 1 worker; if you use three workers, it should take about a third of the time. There will be some overhead and maybe some uneven distribution of work, so don't expect perfect linear speedup. However, if you're not getting much speedup at all, that's a sign that you've done something wrong, or you could divide up the work more evenly among the workers. We will be looking at your execution time when we test your solution. To test performance, you may need to use your own Linux machine or one of the EOS Linux lab machines; the systems you access via remote.eos.ncsu.edu are not good for this test (i.e., don't use them for this). It looks like they only have two cores each, and since they are shared systems, you will be competing with other users for CPU time.

Submission

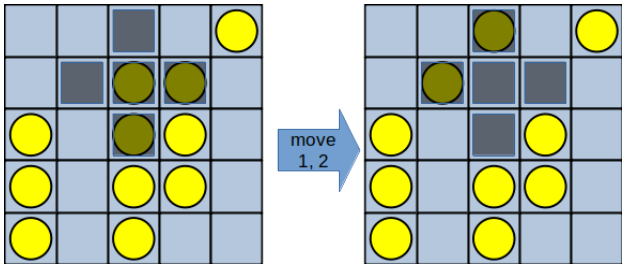
When you're done, submit your source file, `maxsum.c` under the assignment named HW1_Q3 on Gradescope.

4. (40 pts) For this problem, you're going to use Inter-Process Communication (IPC) to create a pair of programs, `client.c` and `server.c` that work together. I've already written part of the server for you.

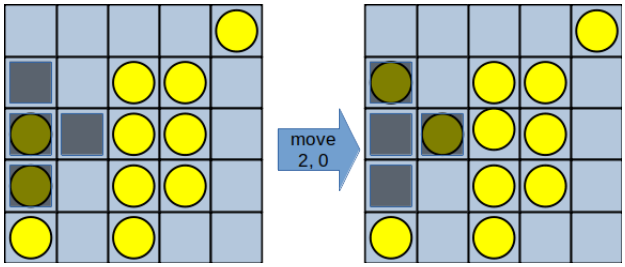
   The client and server work together to implement a multi-player version of the lights-out game from the 1990s. This game uses a playing board like the one illustrated below. The game uses a 5×5 grid of lights, each of which can be off or on.



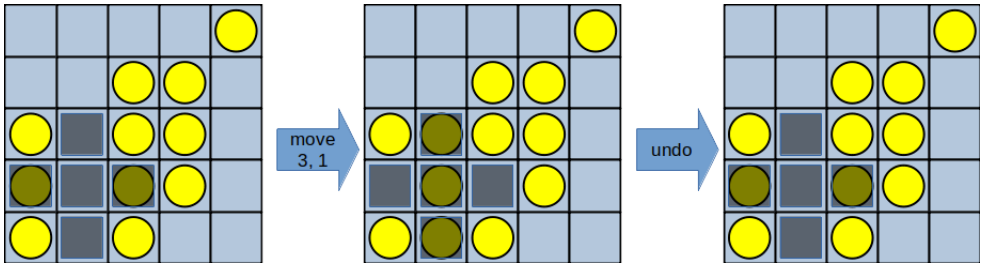
   Players make a move by selecting a cell on the grid. A move toggles the state of the light at the selected cell and at the four neighboring cells around it (above, below, to the left and to the right). Toggling a light changes it from on to off or from off to on. For example, a move selecting the cell at row 1, column 2 will toggle the state of five lights as shown below.



Cells on the edges of the board don't have four neighbors. The player can still make moves at these cells; they just toggle only the cells that actually exit on the board. For example, selecting the cell at row 2, column 0 will only change three neighboring cells; the cell to the right, the one above and the one below. The selected cell has no cells to the left.



Your game will support a one-move undo operation. After making a move, the player can choose to undo the move and automatically return the board to the state it was before that move. The undo history is just one move long, so it's invalid to attempt an undo operations immediately after undo. Likewise, it's invalid to perform an undo before any other move has been made.



There's a Wikipedia page for the lights-out game, at `https://en.wikipedia.org/wiki/Lights_Out_(game)`.

Server Operation

Your server will be responsible for storing the state of the game board and keeping up with the most recent move to support the undo operation. The client will send commands to the server to make a move, undo the most recent move or check the current state of the board.

After the server starts successfully, it will continue running indefinitely, responding to client requests until the user terminates it with ctrl-C.

The server expects a single command-line argument. When you start it, you will give it the name of a file containing the initial state of the board. A board description file will contain five lines of text, each with 5 characters. This defines the initial 5×5 state of the lights. Each character can be a period, '.', indicating a light that's off, or an asterisk, '*' indicating a light that's on.

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

The sample input file above is `board-3.txt` provided with the starter. As expected, it has five lines of five characters each. It has 11 lights on and 14 lights that are off.

If the user runs the server with bad command-line arguments (e.g., too many arguments or missing the board file name), it should exit unsuccessfully and print the following usage message to standard error.

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

If the given board file is invalid (if the file doesn't exist, if it isn't in this format or it contains characters other than '.', '*' and newlines) the server should terminate unsuccessfully and print the following usage message to standard error, where filename is the name of the name of the board file given on the command line:

```
Invalid input file: filename
```

In your server, you will need to come up with your own representation for storing the state of the board in the server. For future assignments, you will want to store everything about the board state (the lights and the most recent move) in a single struct, but you're not required to for this assignment.

Running the Client

Each time you run the client, you indicate with command-line arguments what command it should perform. It sends the command to the sever and then prints out the server's response. The client can be run the following ways:

- `./client move` r c
  Running the client like this requests that the server make a move at row r, column c. Rows are numbered from 0 (the top) to 4 (the bottom) and columns are numbered from 0 (the left) to 4 (the right). The command is invalid if the r or c arguments are missing or if they are not integers between 0 and 4. The client will send the server a message indicating the requested move, and the server should send the client a response indicating success or failure of the request. Before terminating, the client will print out "success" if the command succeeded, or "error" if it failed.
- `./client undo`
  This command requests the sever to undo the most recent move. The server is expected to keep a one-move undo history, so this command is invalid if no move has been made yet, or if the most recent move has already been undone. The server will send back a response indicating the success of the command, and the client will print out either "success" or "error", like it does with the move command.
- `./client report`
  When the user enters this command, the client will request a report of the current state of the board. It will print it out as a 5×5 square of '.' and '*' characters, just like the contents of a board input file. The client doesn't need to print out "success" and "error" for this command; the printout of the board will show that the command was successful.

Error Conditions

If the user gives the client command-line arguments that aren't one of the commands described above, the client should print out "error" to standard output. Whenever the client exits with the "error" message, it should exit unsuccessfully.

You can detect some usage errors in the client, without ever sending a request to the server. It's OK if you want to report the "error" message for these without even contacting the server. For others, you will need to contact the server and receive some kind of response indicating error.

We're not requiring a particular error message if, say, you can't open a message queue or if there's a problem when you try to receive a message. You will want to report if these errors happen (that may help with debugging) but any reasonable error message will be fine.

In the past, I've seen lots of students have trouble with the mq_receive() call. Be sure you look at the documentation for this, and maybe check the return value from this system call and check the value of errno if it fails. This may give you some information about what's going wrong.

Server Termination

The server will continue running until you kill it, accepting requests from any client that sends a request. The server will use the sigaction() call we looked at in class to register a signal handler for SIGINT. When the user kills the server with ctrl-C, the server will catch the signal, print out a newline character, then print out a report of the final state of the board, in the same format as the report command. If the server was started as shown below then immediately killed with ctrl-C, this is what you would expect to see in the terminal. Printing the newline character right before the board makes the formatting look better. It prevents the first line of the board from printing on the same line as the "`^C`" that gets shows up when you press ctrl-C.

```
$ ./server board-3.txt
^C
....*
..**.
*.**.
*.**.
*.*..
```

Client/Server Communication

The client and server will communicate using POSIX message queues. Each time the client is run, it will send a message to the server and then wait for a response. The server's response may contain the result of the command (e.g., for report) or it may just need to indicate success or failure (e.g., for move or undo).

We have a pair of example programs from class that demonstrates how to create and use message queues. A message queue is unidirectional communication channel that lets us send messages (arbitrary sequences of bytes) between processes on the same host. Each message queue needs a unique name. Two processes can communicate by using `mq_open()` to create a new message queue or open an existing one with an agreed-upon name. Then, one process can use `mq_send()` to put a messages into the queue, and another process can use `mq_receive()` to obtain copies of the messages from the queue in the same order they were sent. When a program is done using a message queue, it can close it using `mq_close()`. The message queue will continue to exist (maybe even with some queued messages) until it is deleted with `mq_unlink()`.

Message queues let us send an arbitrary sequence of bytes. You can decide how you want to encode client requests and server responses as messages. For example, you can just encode them as null-terminated strings with spaces separating different values in the message. Using a text format for messages might help with debugging, but it's fine to use a binary format if you'd prefer. How you encode your messages is up to you.

We're each going to need two message queues, one to send requests to the server and another to send responses back to the client. On a single host, every different message queue needs a unique name. To avoid name collisions between students, we'll just include our unity IDs in every message queue name. Use the name "/*unityID*-server-queue" for the message queue used to send messages to your server. Use "/*unityID*-client-queue" as the name of the queue to send responses back to the client.

Partial Implementation

When started, the server creates both of these message queues, opening one for reading and the other for writing. Then, the server repeatedly reads messages from the server-queue and sends back the response via the client-queue. The course Moodle site has a partial implementation of the server. It already creates the needed message queues and unlinks them when it's done. You will need to add code to store and initialize the game board and related state stored in the server. You'll also need code to read, process and respond to messages and handle the SIGINT sent when ctrl-C is pressed, reporting a final state of the board before the program exits.

In addition to the partial server implementation, you also get a `common.h` header file, defining some constants needed by both the client and server. You'll need to update this file to use your unity ID in the message queue names. We're not providing a partial implementation for the client. You get to write that part yourself. The client is easier; mine is less than half the size of my server. Remember, the server is responsible for creating both message queues. The client just needs to open and use them; be sure you don't accidentally delete or overwrite the message queues when the client starts up.

Compilation

To use POSIX message queues, you'll need to link your programs with the `rt` library. Also, to use sigaction() you will need to define the preprocessor symbol, _POSIX_SOURCE. You should be able to compile your client and server using commands like the following:

```
gcc -D_POSIX_SOURCE -Wall -g -std=c99 -o server server.c -lrt

gcc -Wall -g -std=c99 -o client client.c -lrt
```

Execution

Once your code is working, you should be able to leave the server running in one terminal window and connect to it repeatedly by running the client in a different window.

Logging in via remote.eos.ncsu.edu or remote-linux.eos.ncsu.edu takes you to a pool of several different hosts, not necessarily to the same specific machine every time you connect. **Be careful.** If you log in to this address twice, using putty or another ssh client, you may not actually get two terminal windows connected to the same host. Message queues only let you communicate between processes on the same host, so they won't work if you accidentally login on two different machines.

You can type `hostname` in each terminal to see what host it's running on. To be sure you get two logins on the same host, you can login once, use `hostname` to figure out what host you're on, then login directly to that same host (rather than remote.eos.ncsu.edu) using a second remote login window.

Sample Execution

Once your client and server are working, you should be able to run them as follows. Here, the left-hand column shows the server running in one terminal, and the right-hand column shows the client being run repeatedly in a different terminal window. I've put in some vertical space to show the order these commands are executed, and I've included some comments to explain some of the steps. We start the server on the left, run the client several times and then kill the server. When the server is terminated, you can see a printout for the ctrl-C (printed as `^C`), then the server's report of the final state of the board.

```
# Run the server starting from board-3.txt
$ ./server board-3.txt
```

```
# Look at the board via the client
$ ./client report
....*
..**.
*.**.
*.**.
*.*..

# Make a move and look at the result
$ ./client move 0 0
success
$ ./client report
**..*
*.**.
*.**.
*.**.
*.*..

# Make another move in the lower-left corner
$ ./client move 4 0
success
$ ./client report
**..*
*.**.
*.**.
..**.
.**..

# Undo the last move.
$ ./client undo
success

# Undo history is just one move deep.
$ ./client undo
error

# Result after the first undo.
$ ./client report
**..*
```

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

                                    # Try some invalid commands from the client.
                                    $ ./client move abc xyz
                                    error
                                    $ ./client move 5 5
                                    error
                                    $ ./client bad-command
                                    error

# Kill the server with ctrl-C and see the final state of the board
^C
**..*
*.**.
*.**.
*.**.
*.*..

# Try some invalid arguments for the server.

# A bad board file.
$ ./server board-4.txt
Invalid input file: board-4.txt

# Too many arguments
$ ./server a b c
usage: server <board-file>
```

Submission

When you're done, submit the source code for your **client.c**, your modified **server.c** and your modified **common.h** under the assignment named HW1_Q4 on Gradescope. From previous years teaching this class, I've seen that students sometimes forget to submit their header files, so be sure to submit your **common.h** file. Otherwise, we won't be able to compile your code.