Operating Systems, Assignment 1

This assignment includes a few programming problems. When grading, we will compile and test your programs on one of the EOS Linux machines. In general, we'll compile with options like the following, although some programs require additional options (as noted below):

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

Here, the -g option tells the compiler to include symbol information in the executable, to help debuggers do their job. The -Wall option turns on lots of compiler warnings. This may help you to find errors in your code, even if the errors aren't enough to prevent compilation. The -std=c99 option enables some newer, C99 language features you're probably accustomed to.

If you develop a program on a different system, you'll want to make sure it compiles and runs correctly on EOS Linux machines using these options before you turn it in. 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) Using your textbook, please give a short definition of the following terms, in your own words. Write up all your answers into a plain text file called `terms.txt` and submit it using the assignment_1 assignment in Moodle.

   (a) NVRAM
   (b) Cache Coherency
   (c) Batch interface
   (d) Data section (of a process' memory)

2. (16 pts) Log in to one of the EOS Linux machines (one of the Linux desktop machines in EB 3 or Daniels, or via remote.eos.ncsu.edu) and use the online manual pages to answer each of the following questions. To read the online manual page for a particular command, function or system call, you can usually just type `man` *call*, where *call* is the call you want to read about. If two different manual pages have the same name, you can specify the section of the manual to make sure you get the page you want. Section 1 of the manual is for shell commands, and section 2 is for system calls. So, **man 1 stat** will tell you about the stat shell command, and **man 2 stat** will tell you about the system call with the same name.

   Write your answers into a plain text file called `problem2.txt` and submit it for assignment_1 Moodle.

   (a) The wait() system call will block the parent until a child process terminates, but it will also return to the parent when other state chanes occur in a child. What are the other two state changes for a child that will cause wait() to return in the parent?
   (b) The fork() system call may not always succeed. If, for example, a process has already created too many children, the process's resource limits will prevent creation of more. When this happens, how does fork() notify the caller of this? Specifically, how does fork() notify the caller that it failed and that reaching the parent's resource limit on the number of processes was the reason for the failure?
   (c) 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?
   (d) In class, we looked at a sample program that used sigaction() to register a signal handler. You'll sometimes see people doing this with the simpler signal() system instead, but using signal() is probably a bad way to register a signal hander. Briefly explain why.

3. (40 pts) For this problem, you're going to write a program that can use multiple CPU cores to solve a computationally intensive problem more quickly. Your program will be called square.c. It will read in a large grid of lower-case letters. In the grid, you'll look for a 6×6 square that contains at least one copy of every letter of the alphabet. We'll call this a *complete* square (I'm making up a name for these kinds of squares, so it's easy to talk about them later).

   For example, the following is one of the sample input files given out with this problem, input-2.txt. The first line gives the size of the grid of letters (number of rows, then number of columns). This is followed by a grid of lower-case letters of the given dimensions. In the following example there's a complete square that starts with the first letter on the first row (so, the 6×6 square in the upper left corner). There's another complete square that starts with the second letter of the second row (so, the 6×6 square in the lower right corner). The 6×6 square in the upper right isn't complete; it's missing the letter 'a'. The one in the lower left is missing the letter 'f'.

   ```
   7 7
   abcdefm
   ghijklg
   mmmmmmm
   nnnnnnn
   opqrsto
   uvwxyzu
   nabcdef
   ```

   To help you solve this problem, I'm giving you a starter. It parses the command-line arguments and reads in the grid of letters from standard input. You'll add code to look for complete squares by just checking every 6×6 square in the whole grid.

   The first command-line argument gives the number of worker processes to use (see below). Optionally, this can be followed by the word "report". If a report is requested, then the program will print a report of every complete square it finds, by printing the location of the square's upper left corner to standard output. So, for example, a report of the complete squares in the 7×7 grid above would consist of `0 0` and `1 1`. The report of squares can be in any order.

   Parallelization

   The interesting part of this problem is using multiple child processes to find complete squares in parallel. If you do this right, it should be able to solve the problem faster on a multi-core system.

   On the command line, the user gives an integer argument specifying the number of child processes to create to help solve the problem more quickly. We'll call these children, *workers*. After reading the grid of letters, the parent will fork its workers and then let them do all the real work of solving the problem.

   The workers will cooperate to find complete squares. Each worker will be responsible for checking part of the grid of letters. You can divide up the work however you want. I used a simple technique in my solution. Let's say $r_0, r_1, r_2 \ldots$ are names for the rows of letters in the grid. Given m workers, I just made the first worker responsible for looking for complete squares that started on rows $r_0, r_m, r_{2m}, r_{3m} \ldots$. The next worker was responsible for looking at squares that started with any of the rows $r_1, r_{1+m}, r_{1+2m}, r_{1+3m}, \ldots$, and so on.

   Each worker should count all the complete squares it finds, reporting them to the terminal if the report option was given on the command line.[1] When they're done, the workers will send their count back to their parent process, so it can report a total count.

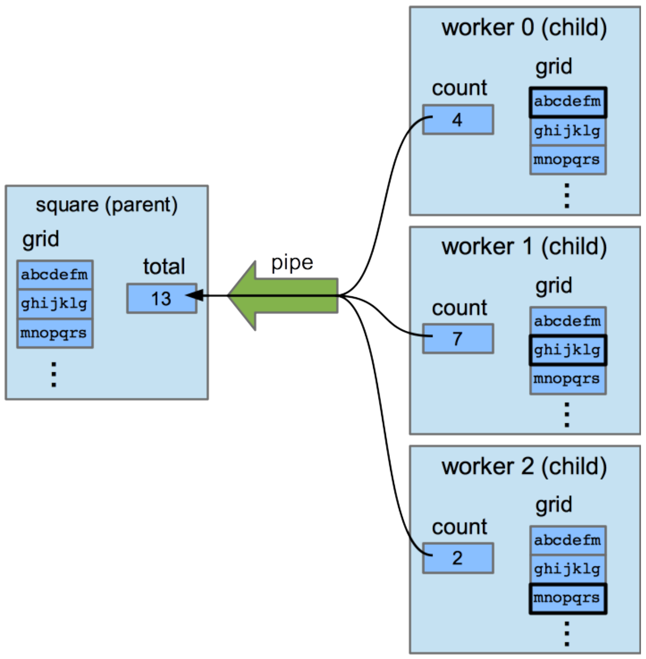
[1] This is why it's OK for you to report the squares in any order. Each work can just immediately report a complete square as soon as it finds on, without worrying about the overall order of the list.

Interprocess Communication

When the workers are created, they will automatically get a copy of the grid of characters 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 all the complete squares.

When the workers are done, they need a way of sending their resulting local counts back to their parent. 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 count 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 count in binary. Since it's just going between processes on the same host, we don't have to worry about byte order or other architectural differences. 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 count from each child, the parent will add them up to report an overall total. 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 be made to wait until the first one unlocks it. This technique wouldn't prevent malicious code from writing to the pipe, but, 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 define on the EOS Linux systems. You'll want to compile your program like the following.

```
  gcc -Wall -std=c99 -D_BSD_SOURCE -g square.c -o square
```

Testing

Once your program is working, you should be able to run it as follows:

```
# One worker on the smallest test input
$ ./square 1 < input-1.txt
Squares: 1

# Two workers, on the 7X7 input, reporting the solutions sas they find them.
$ ./square 2 report < input-2.txt
0 0
1 1
Squares: 2

# Four workers, on a 70x70 input
$ ./square 4 report < input-3.txt
12 25
24 8
29 45
2 56
45 46
49 60
11 25
53 59
27 51
43 64
58 26
Squares: 11

# Timinig the execution of 4 works on a larger input file.
# Your execution time may not be the same as what I got.
$ time ./square 4 < input-4.txt
Squares: 954

real 0m1.026s
user 0m3.617s
sys 0m0.079s
```

Be sure to try out your program on some of the large input files using the time command with multiple workers. As long as the number of workers is less than or equal to the number of CPU cores, you should see speedup that's close to linear in the number of workers. This is something we'll be looking for 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 look like they all just have one core.

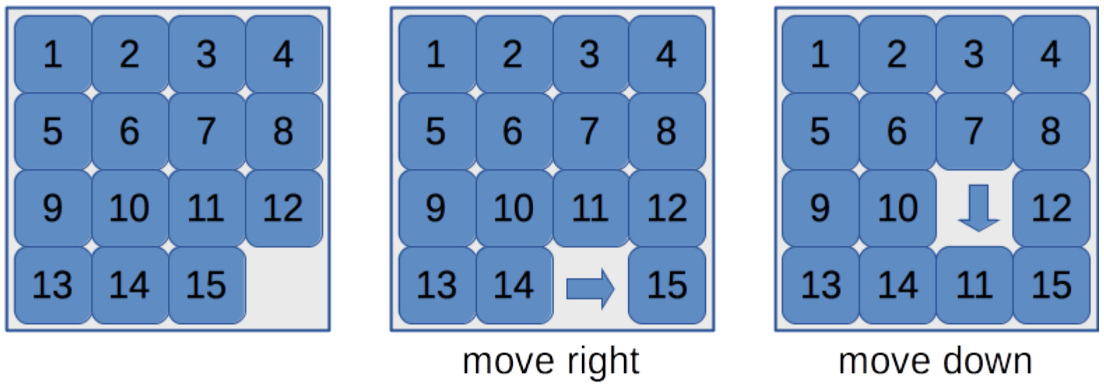
Submission

When you're done, submit your source file, `square.c`, using the assignment_1 assignment in Moodle.

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.

Client/Server Operation

The client and server work together to implement a simulation of the 15-puzzle. Probably, you've seen this kind of puzzle. It's a 4×4 board with sliding tiles numbered $1 \ldots 15$. There's one blank space that you can slide tiles into by moving an adjacent tile left, right, up or down. Normally, the goal is to try to get all the tiles in order, but our program won't worry about this. You'll just let the user move tiles around and look at the current state of the board.



When the server is first started it will create a data structure representing the fifteen puzzle in its solved state, the picture on the left. The client/server program supports five commands. The `board` command prints out a picture of the current state of the board, and movement commands let the user move tiles into adjacent spaces.

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.

- `./client board`
  Running the client like this requests that the server report the current state of the board. The client should print out a picture of the board like in the sample execution below, with each tile number printed in a two column field and with a space between columns of the board. Print a dash to indicate the empty space on the board.
- `./client up`
  `./client down`
  `./client left`
  `./client right`
  Running the client like this sends a request to the server to move a tile in the indicated direction,

into the empty space. So, for example, the up command will move the tile below the empty space up into the empty space. The client should respond with the word `success` if the move was successful, or with `invalid command` if it wasn't. For example, if the empty space was on the bottom row of the board, then the `up` direction would be an invalid command.

- Invalid command
  Any other command-line arguments should be considered invalid, and the client should print the same `invalid command` message.

Server Execution

The server will continue running until you kill it, accepting requests from any client that sends them (so, we won't be doing any authentication of clients). 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 blank line, then print out a the final state of the board, just like what we get when we run `./client board`. Printing a blank line after the ctrl-C will make the final report look better. The first line won't be printed with a `^C` to the left.

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 to report to the user.

We have a pair of example programs from class that demonstrate 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 or open an existing message queue with an agreed-upon name. Then, one process can use `mq_send()` to put a messages into the queue, and another program can use `mq_receive()` to get 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. For example, when the client asks to see a copy of the board, it could just send the string "board" to the server and the server could respond with a big string containing the current state of the board. If you use a string encoding for messages, be sure to think about null termination for your string. For example, if you send the string "board" as a 5-byte message, then the receiver won't actually get the null terminator marking the end of the string.

For client/server communication, 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 website 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 board representation. You'll need to read, process and respond to messages and handle the SIGINT sent when ctrl-C is pressed.

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. I'm not providing a partial implementation for the client. You get to write that part yourself (but, 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.

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 multiple EOS linux hosts. **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 your ssh client.

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.

```
$ ./server

                                  # Make a couple of legal moves, then
                                  # check the state of the board.
                                  $ ./client down
                                  success
                                  $ ./client right
                                  success
                                  $ ./client board
                                  1  2  3  4
                                  5  6  7  8
                                  9 10  - 11
                                 13 14 15 12

                                 # Two more right moves are OK
                                 $ ./client right
                                 success
                                 $ ./client right
                                 success
                                 # ... but then you reach the edge.
                                 $ ./client right
                                 invalid command
                                 $ ./client board
                                  1  2  3  4
                                  5  6  7  8
                                  -  9 10 11
                                 13 14 15 12

                                 # Up and left commands
                                 $ ./client up
                                 success
                                 $ ./client left
                                 success
                                 $ ./client board
                                  1  2  3  4
                                  5  6  7  8
                                 13  9 10 11
                                 14  - 15 12

                                 # The server rejects nonsense commands
                                 # just like invalid moves.
                                 $ ./client jfksaljf
                                 invalid command

# Kill the server with ctrl-C
# and see the final board state.
^C
^C
 1  2  3  4
 5  6  7  8
13  9 10 11
14  - 15 12
```

Submission

When you're done, submit the source code for your **client.c**, your modified **server.c** and your modified **common.h** using the assignment_1 submission in Moodle. 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.