Project-2-SP22
Development of a Linux Shell

Project 2: Developing a Linux Shell
NOTE What follows is the only the introduction to the problem statement.
**The full Project 2 description is
[here](https://github.com/CIS-3207/Project-2-Fall21/blob/main/Project%202%20Developing%20a%20Linux%20Shell.pdf)**.

An **Introduction to Developing a Shells** for this project is found
[here](https://github.com/CIS-3207/Project-2-Fall21/blob/main/3207%20Introduction%20to%20Developing%20a%20Linux%20Shell.pdf).

In this project, you'll build a simple Unix/Linux shell. The shell is the heart of the command-line interface, and thus is central to the Linux/Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.
There are three specific objectives to this assignment:
- To further familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.
Overview
In this assignment, you will implement a **command line interpreter (CLI)** or, as it is more commonly known, a **shell**. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Linux. If you don't know what shell you are running, it's probably bash. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials (and the associated Shell Introduction file).
Program Specifications
Basic Shell:

myshell

Your basic shell, called myshell is fundamentally an interactive loop: it repeatedly prints a prompt myshell> (note the space after the greater-than sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types *exit*

. The name of your final executable should be myshell.

The shell can be invoked with either no arguments (interactive) or a single argument (batch0; anything else is an error. Here is the no-argument way:
prompt> ./myshell
myshell>
At this point, myshell is running, and ready to accept commands. Type away!
The mode above is called /interactive/ mode, and allows the user to type commands directly. The shell also supports a /batch mode/, which instead reads input from a batch file and executes commands found in the file. Here is how you run the shell with a batch file named `batch.txt`
:

` prompt> ./myshell batch.txt`

There is a difference between batch and interactive modes: in interactive mode, a prompt is printed (myshell>). In batch mode, no prompt should be printed during execution of commands.

You should structure your shell such that it creates a process for each new command (the exceptions are /built-in commands/, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types 'ls -la /tmp', your shell should run the program `/bin/ls`
with the given arguments
` - la`
and ` /tmp`
(how does the shell know to run ` /bin/ls`
? It's something called the shell *path*; more on this below).

Your project is to develop/write a simple shell - myshell - that has the following properties:
1. The shell must support the following internal commands:
a. cd <directory> - Change the current default directory to
<directory>. If the <directory> argument is not present, report the current directory. If the directory does not exist an appropriate error should be reported. This command should also change the PWD environment variable.
b. clr - Clear the screen.
c. dir <directory> - List the contents of directory <directory>.
d. environ - List all the environment strings.
e. echo <comment> - Display <comment> on the display followed by a new line (multiple spaces/tabs may be reduced to a single space).
f. help - Display the user manual using the more filter.
g. pause - Pause operation of the shell until 'Enter' is pressed.

h. quit - Quit the shell.
i. The shell environment should contain shell=<pathname>/myshell where <pathname>/myshell is the full path for the shell executable(not a hardwired path back to your directory, but the one from which it was executed).

2. All other command line input is interpreted as program invocation, which should be done by the shell *fork*ing and *exec*ing the programs as its own child processes. The programs should be executed with an environment that contains the entry: parent=<pathname>/myshell where <pathname>/myshell is as described in 1.i. above.

3. The shell must be able to take its command line input from a file. That is, if the shell is invoked with a command line argument:
myshell batchfile
then batchfile is assumed to contain a set of command lines for the shell to process. When the end-of-file is reached, the shell should exit. Obviously, if the shell is invoked without a command line argument, it solicits input from the user via a prompt on the display.

4. The shell must support I/O - redirection on either or both *stdin* and/or *stdout*. That is, the command line
programname arg1 arg2 < inputfile > outputfile
will execute the program /programname/ with arguments /arg1/ and /arg2/, the stdin FILE stream replaced by /inputfile/ and the stdout FILE stream replaced by /outputfile/.
stdout redirection should also be possible for the internal commands
dir, environ, echo, & help.
With output redirection, if the redirection character is > then the outputfile is created if it does not exist and truncated if it does. If the redirection token is >> then outputfile is created if it does not exist and appended to if it does.

5. The shell must support background execution of programs. An ampersand (*&*) at the end of the command line indicates that the shell should return to the command line prompt immediately after launching that program.

6. You are to include an implementation of command line pipes. (essentially an extension of redirection) so that commands can be strung together. An example is
cat out.txt | wc –l

7. The command line prompt must contain the pathname of the current directory.
/Note:/ You can assume that all command line arguments (including the redirection symbols, <, > & >> and the background execution symbol, &) will be delimited from other command line arguments by white space - one or more spaces and/or tabs (see the command line in 4. above).

Project-3-S22
CIS 3207 Project 3 Multi-Threaded Echo Server

Project Overview

Echo servers are often used to demonstrate and test networked communication. In this assignment, you'll create an echo server and echo client and evaluate their performance. The purpose of the assignment is to gain some exposure and practical experience with multi-threaded programming and the synchronization problems that go along with it, as well as with writing programs that communicate across networks.

You'll learn a bit about network sockets in lecture and lab. Much more detailed information is available in Chapter 11 of Bryant and O'Hallaron, and Chapters 57-62 in Kerrisk (see Canvas Files: Additional Textbook References). [Beej's Guide](http://beej.us/guide/bgnet/) and [BinaryTides' Socket Programming Tutorial](http://www.binarytides.com/socket-programming-c-linux-tutorial/) are potentially useful online resources.

For now, the high-level view of network sockets is that they are communication channels between pairs of processes, somewhat like pipes. They differ from pipes in that a pair of processes communicating via a socket may reside on different machines, and that the channel of communication is bi-directional.

Much of what is considered to be "socket programming" involves the mechanics of setting up the channel (i.e., the socket) for communication. Once this is done, we're left with a socket descriptor, which we use in much the same manner as we've used descriptors to represent files or pipes.

In this project you will develop a server program that echoes a text message on demand. Your echo server is to be a process that will read sequences of words (sentences) sent by clients. The sentences (text strings), sent by a client computer, will be sent back to the requesting client.

Server Program Operation

Server Main Thread

Your server program should take as a command line several control parameters. The first parameter is the maximum length of message that it can receive. If none is provided, DEFAULT_LENGTH is used (where DEFAULT_LENGTH is a named constant defined in your program). The program should also take as a parameter a port number on which to listen for

incoming connections. Similarly, if no port number is provided, your program should listen on DEFAULT_PORT (defined in your program).

Three other parameters are the number of element cells in the 'connection buffer', the number of worker threads, and the terminator character to end echoing for this client. These parameters are discussed in this document.

The main server thread will have two primary functions: 1) accept and distribute connection requests from clients, and 2) construct a log file of all echo activities.

When the server starts, the main thread creates a fixed-sized data structure which will be used to store the socket descriptor information of the clients that will connect to it. The number of elements in this data structure (shared buffer) is specified by a program input parameter ('size of the connection buffer'). The main thread creates a pool of worker threads ('the number of threads' specified as a program parameter), and then the main thread immediately begins to behave in the following manner (to accept and distribute connection requests):

```
while (true) {
  connected_socket = accept(listening_socket);
  add connected_socket information to the work buffer;
  signal any sleeping workers that there's a new socket in the buffer;
}
```

A second server thread will monitor a log queue and process entries by removing and writing them to a log file.

```
while (true) {
  while (the log queue is NOT empty) {
     remove an entry from the log
     write the entry to the log file
    }
 }
```

Connection Buffer Data

The cells in the connection buffer are filled by the main server thread. The cells are processed by the worker threads. Each cell is to contain the connection socket, and the time at which the connection socket was received.

Worker Thread

Each server worker thread's main loop is as follows:

```
while (true) {
  while (the work queue is NOT empty) {
    remove a socket data element from the connection buffer
    notify that there's an empty spot in the connection buffer
    service the client
    close socket
  }
}
```

and the client servicing logic is:

```
while (there's a message to read) {
  read message from the socket
  if (the message is NOT the message terminator) {
    echo the message back on the socket to the client;
  } else {
    echo back on the socket "ECHO SERVICE COMPLETE";
  }
  write the received message, the socket response message or "ECHO SERVICE COMPLETE")
and other log information to the log queue;
}
```

We quickly recognize this to be an instance of the Producer-Consumer Problem that we have studied in class. The work queue (connection buffer) is a shared data structure, with the main thread acting as a producer, adding socket descriptors to the queue, and the worker threads acting as consumers, removing socket descriptors from the queue. Similarly, the log queue is a shared data structure, with the worker threads acting as producers of results into the buffer and a server log thread acting as a consumer, removing results from the buffer. Because we have concurrent access to these shared data structures, we must synchronize access to them using the techniques that we've discussed in class so that: 1) each client is serviced, and 2) the queues do not become corrupted.

Echo Thread Processing

The server inserts socket descriptions into the buffer in the order received. Worker threads remove the socket descriptors in FIFO order from the buffer.

Once the message is received and the response sent to the client, the worker thread will create an entry in the log buffer. The log buffer entry is to contain the arrival time of the request, the time the reversal was completed, the message received, and the reply message sent to the client.

Synchronization.

Correctness

Only a single thread at a time may manipulate the work queue. We've seen that this can be guaranteed through the proper use of mutual exclusion. Your solution should include synchronization using locks and condition variables.

No more than one worker thread at a time should manipulate the log queue at any one time. This can be ensured through the proper use of mutual exclusion. Again, synchronization should be using locks and condition variables.

Efficiency

A producer should not attempt to produce if the queue is full, nor should consumers attempt to consume when the queue is empty. When a producer attempts to add to the queue and finds it full, it should cease to execute until notified by a consumer that there is space available.

Similarly, when a consumer attempts to consume and finds the queue empty, it should cease to execute until a producer has notified it that a new item has been added to the queue. As we've seen in class, locks and condition variables can be used to achieve this. Your solution should not involve thread yields or sleeps.

Code Organization

Concurrent programming is tricky. Don't make it any trickier than it needs to be. Bundle your synchronization primitives along with the data they're protecting into a struct, define functions that operate on the data using the bundled synchronization primitives, and access the data only through these functions. In the end, we have something in C that looks very much like the Java classes you've written in 1068 and 2168 with some "private" data and "public" methods, or like monitor functions. Code and some very good advice can be found in Bryant and O'Hallaron Chapter 12.

Testing your program

At the beginning, as you are developing your server, you'll probably run the server and a client program on your own computer. When doing this, your server's network address will be the loopback address of 127.0.0.1. (do some research on this).

You are to write a basic client to test your server. This is part of the assignment. For initial testing of communication with the server, you could also use the Unix telnet client, which, in addition to running the telnet protocol, can be used to connect to any TCP port, or you could use a program such as [netcat](https://en.wikipedia.org/wiki/Netcat) . You will need to use your developed client to test and demonstrate your solution.

Once you're ready to deploy your program on a real network, please restrict yourself to the nodes on cis-linux2([system list](https://cis.temple.edu/~jfiore/2017/spring/3207/assignments/spell_check/cis-linux2_system_list)). Start an instance of your server on one of the cis-linux2 systems and _run multiple simultaneous instances_ of your client on other systems.

You should use many instances of clients requesting echo services at the same time (for the demo, use of multiple clients is required). These clients should be run from more than 1 computer system simultaneously, i.e., each client computer system should run many client instances at the same time. Your testing and demonstration should show this.

The server program has options for buffer size and number of worker threads as parameters. You must use variations in these parameters to demonstrate that you are able to ensure proper synchronization and performance under varying loads of requests (number of simultaneous clients and frequency of requests).

There will be weekly deliverables and they are to be submitted to Canvas as well.

WEEK 1
Create main server thread, worker threads and log manager thread.
Create buffer and management for socket descriptor elements from main server thread.
Create management of socket descriptors by worker threads
Create main thread socket for listening for incoming requests from clients
Create client send request socket for communication with main server thread
Create worker thread insertion in the log buffer

Week 2
Create worker thread communication with the client for echoing
Create log manager thread updates to log file
Create client generation and message sending

Testing with single and multiple clients and varied intervals of requests, buffer sizes and numbers of worker threads for message echo
Project completion and submission for demo

Project-4-s22 Signaling with Multi-Process Programs

Problem:
This project requires _a program development_ and an analysis of performance for the solution.

Summary:
The purpose of this project is to have a process and its threads responsible for servicing “signals” sent by other peer processes. The processes and threads must execute concurrently, and performance of all process and thread activities is to be monitored and analyzed.

Learning Objectives:
This project expands on our knowledge and experience of creating applications with multiple processes and multiple threads. The project also introduces signals as communication mechanisms. Synchronization in the form of protected access to shared variables is used.

Project Description:
This project is to be implemented using Linux and the signaling system calls available in Linux. A signal is an event that requires attention. A signal can be thought of as a software version of a hardware interrupt. There are many different signal types and each signal type has a defined action. The action of many of the signal types can be changed by developing a “signal handler” in code to process the receipt of a particular signal type.

There are signals that are considered “synchronous” and there are signals considered “asynchronous”. A synchronous signal is one that is the result of an action of a particular thread such as a divide by zero error or an addressing error. An asynchronous signal is one that is not related to a particular thread’s activity and can arrive at any time, that is with no specific timing or sequencing. In this project we are dealing with asynchronous signals.

In Linux, asynchronous signals sent to a multi-threaded process are **delivered to the process**. The decision as to which thread should handle the arriving signal is based upon the signal mask configuration of the individual threads within the process. If more than one thread has not ‘blocked’ the received signal, there is no guarantee as to which of the non-blocking threads will actually receive the signal.

You will need to do some research about signals and signal masks. There are a number of system calls related to signals and signal masks that you can use in the implementation of your solution to the project.

The **Program** (you are developing a single program with multiple processes) for this project requires developing a solution using a parent process, and a _signal handling_ process that contains 3 threads. In the signal handling process, two of the threads are signal receiving threads and one thread is for reporting signal receipts.

For the **Program**, the parent process creates one child process, sets up execution of the child process and waits for its completion. In addition, the parent process behaves as a signal generator. The **Program** is to use two signals for communication: SIGUSR1 and SIGUSR2. The parent process will execute in a loop, and during each loop repetition will randomly select either SIGUSR1 or SIGUSR2 to send to the signal receiving process. Each repetition through the signal generating loop will have a random time delay ranging between .01 second and .1 second before moving to the next repetition. After generating a signal, the process will log a 'signal sent time' and 'signal type' (SIGUSR! Or SIGUSR2). The time value must have sufficient resolution to distinguish among signals sent by the processes.

Signals should be sent with the function (look up the function and the meaning of the parameters)
int kill (pid_t pid, int signum)

The logs of the signal sending process are to be written to a file.

Each of the signal handling threads (in the child process) will process only one type of signal, _either_ SIGUSR1 or SIGUSR2 and ignore servicing the other signal type. (Thus, one of the signal receiving threads will handle SIGUSR1 signals and one signal receiving thread will handle SIGUSR2 signals). Each of the signal handling threads will also execute in a loop, waiting for its signal to arrive. When the signal arrives, the thread will save the signal type, time of receipt of the signal and the ID of the receiving thread in a shared buffer and increment the total count of the number of signals received. This counter and buffer are shared by the threads in the receiving process. The thread will then loop waiting for the next signal arrival.

The reporting thread in the receiving process will monitor the receive signal counter and when an aggregate of 20 signals have arrived, it will log the signal type, receive time and receiving thread ID for those signals to a file.

Access to shared structures by each thread requires the use of a critical section control. You are to use a lock for the critical section control. Use the lock to provide each thread exclusive access to each shared structure.

Results to Report:
We are interested in the performance of the threads receiving signals. This means you will need to “stress” the receiving process and its threads to see whether it is able to handle the signal service demand made by the signal senders. You may want to place additional limits (further **reducing** the loop delays) on the interval between signals sent to cause failures in signal processing.

You will run the **Program** until a *“kill”* signal is received from a user terminal. This will let you control the length of time the program executes. You should run the program for a variety of short and long times to see if time affects signal loss.

The kill signal sent from the terminal should cause both the sending and receiving processes to complete their operations, close their files and terminate.

To help you control the termination of the **Program**, we have provided a sample program showing the setting up of a signal handler to ‘catch’ a signal sent to multiple processes from a user terminal. (pgrpid_with_simple_handler.c)

There is some minimal analysis that you should perform to determine if the receiving threads are receiving all sent signals. This can easily be done using a spreadsheet (Excel, Google Sheets, …). You should read the data files into the spreadsheet to do analysis.

You are to examine the performance; that is, the number of signals of each type sent and received. Also examine the average time between signal receptions of each type by the reporting threads. Be sure to investigate and discuss any signal losses.

Since you are logging the times of signals sent by the sending process and the receive times by the threads, you should be able to determine when (time and conditions) the sending demand (intervals between sends) results in signal losses. This is the reason for many runs of different lengths of time as well as the variation in loop delays between signal sends.

Be sure to include a test plan for the program, and document the results of your testing such that you can **clearly demonstrate** that the program **executes correctly**.

You will develop your program using the supplied GitHub repository. You will submit your program code, testing documents and execution results through Canvas. Compress all of the files

(do not include executables) into one file for submission. Use clear naming to help identify the project components that are submitted. Include a link to your GitHub repository as a comment to your Canvas submission.

There will be weekly deliverables and they are to be submitted to Canvas as well.

Week 1:
* Creation of the main process, the signal catcher process and its threads. (Implement the code for each process as independent programs: one source for signal generator code, one source for signal catcher process and threads. This means use fork() and exec() in your implementation).
* Design signal masks for the threads.
* Design signal catcher routines

Week 2+:
* Create individual locks for shared structures.
* Description of testing and documentation (did they include logs in processes, e.g.).
* *Document Production*
    * Reporter receipt and time stamping of signals and intervals for process execution to a file
    * Description and analysis/comparison of results from execution – including multiple executions and comparison of data.
Final executable and Demo

Grading Rubric
- Creation of the main process, the additional signal receiver process, the 2 signal catcher threads and the reporter thread. (1.5 Pt)
- Create shared counters and individual locks for each counter (1 pt)
- Description of program design and comments describing each function (1 pt)
- Description of testing and documentation for signal senders (1 pt)
- Logging of signal send information at the sender processes (1 pt)
- Creation of proper functioning signal catcher routines (1 pt)
- Proper use of signal masks to enable/disable signal receptions (1 pt)
- Reporter receipt and time stamping of signals and intervals for Threads to a file (1 pt)
- Description and analysis/comparison of results from execution (1.5 pt)

Total 10 Points