CS 536 Fall 2021

Lab 1: System Programming Review [210 pts]

Due: 09/20/2021 (Mon), 11:59 PM

Objective

The objective of this introductory lab is to review C system programming which is the foundation for network/socket programming. If you are rusty, please use this practice period to bring yourself up-to-speed on system level C programming. Utilize the PSOs and office hours to resolve any questions. The goal of this lab is to understand the structure and implementation of concurrent client/server apps which comprise the bulk of networked applications in the real-world. This lab is an individual effort. Please note the lab assignment policy specified on the course home page.

Reading

Read chapter 1 from Peterson & Davie (textbook).

Problem 1 [40 pts]

The directory

/homes/park/pub/cs536/lab1 (symbolic link currently pointing to /u/riker/u3/park/pub/cs536/lab1)

accessible from our lab machines in LWSN B148 and HAAS G050 contains simsh.c which implements a prototypical concurrent server: a simple, minimalist shell. A concurrent server receives and parses a client request -- in this case, from stdin (by default keyboard) -- then delegates the actual execution of a requested task to a worker process or thread. Many apps in the real-world, including network software, follow the general structure of concurrent client/server code which are multithreaded programs. They constitute an important baseline for implementing network software.

Create a directory, lab1/, somewhere under your home directory, and a subdirectory v1/ therein. Copy simsh.c to v1/, compile, run and check that you understand how it works. Then modify simsh.c, call it simsh1.c, that accepts command-line arguments. That is, whereas simsh.c can handle "ls" and "ps", it cannot handle "ls -l -a" and "ps -gaux". Your modified code, simsh1.c, handles command-line arguments by parsing "ls -l -a" into string tokens "ls", "-l", "-a" and passing them to execvp() as its second argument which is an array of pointers to the parsed tokens. Make your code modular by performing the parsing task in a separate function that is placed in its own file. Provide a Makefile in v1/ that compiles your code and generates a binary executable simsh1.bin. Compile, test, and verify that your code works correctly. Annotate your code so that a competent programmer can understand what you are aiming to do.

Problem 2 [100 pts]

Create a subdirectory v2/ under lab1. Modify your code, simsh1.c, so that instead of accepting client requests from stdin (i.e., human typing on keyboard in v1/) the request is communicated from a client process. The server process running commandserver.bin creates a FIFO (i.e., named pipe) to be used as a communication channel where clients can submit requests. After creating a FIFO, its name is saved in a text file v2/serverfifo.dat that a client can read to find the name of the FIFO. A client running commandclient.bin starts after the server process has started executing and created serverfifo.dat. If the client cannot access

serverfifo.dat, it terminates after printing a suitable error message to stderr. When testing, run the server and client processes in separate windows for ease of observation.

After finding the name of the server's FIFO, the client uses open() to open the FIFO and sends its request using write(). The client's request is provided by a human user through stdin. That is, the client is itself a server who prints a prompt '> ' to stdout and accepts requests from stdin. The request is then submitted to the server process via its FIFO. To support multiple client processes, we will use a message format where a request ends with the newline character. Hence we are using '\n' as a separator or delimiter. When using FIFO to receive multiple client requests, the potential for interleaving of characters belonging to two or more requests must be considered. Check up to how many bytes in a write() system call Linux FIFOs guarantee interleaving will not occur -- i.e., write() behaves atomically -- and implement your client so that it rejects user requests that exceed the limit.

Implement your client/server app in a modular fashion and Provide a Makefile in v2/ to compile and generate commandserver.bin and commandclient.bin. Create README under v2/ that specifies the files and functions of your code, and a brief description of their roles. Test your client/server app with multiple client processes and verify correctness.

Problem 3 [70 pts]

This is an extension of Problem 2 where instead of showing the output of the command executed by the server on stdout of its associated window, the output is sent to the client who submitted the request. The client then outputs the response from the server on its stdout in its associated window. To facilitate communication from server back to client, a client sets up its own FIFO before sending a request to the server. The name of the FIFO is cfifo<clientpid> where <clientpid> is the pid of the client. For example, if a client has pid 22225 then its FIFO is named client22225. Since the server needs to know the client's pid to open the client's FIFO and write the response, we will change the format of the request message so that a client's message starts with its pid followed by newline which is then followed the command and ends with newline. For example, if the client with pid 22225 is requesting that 'ls -l -a' be executed by the server, the message would be the five characters of string "22225" followed by '\n' followed by the eight characters of string "ls -l -a" followed by '\n'.

In the above example, when a child process of the server process calls execvp() to run the legacy app /bin/ls with command-line arguments "-l" and "-a", the output of /bin/ls which has been coded to output to stdout must be redirected to the FIFO of the client process that sent the request. Use the dup2() system call make file descriptor 1 (i.e., stdout) point to the client's FIFO before calling execvp(). This ensures that when the legacy app ls outputs to stdout the characters are written to the client's FIFO. After submitting a sequest, a client blocks on its FIFO by calling read() to await the server's response. After reading the server's response, the client writes it to stdout which will output to the window associated with the client process.

Create subdirectory v3/, port and modify the code of Problem 2 so that the server's response is sent to a client's FIFO. Update Makefile and README to reflect the changes. Test and verify that the modified client/server app implementing bidirectional communication works with multiple clients.

Bonus problem [20 pts]

Use the ping application from one of our lab machines to measure the time it takes for a packet to reach the destination and receive a response, called the round-trip time (RTT). Do so for CS's web server www.cs.purdue.edu and IUPUI's web server www.iupui.edu in Indianapolis. Select five additional destinations: midwest, east coast, west coast, across the Atlantic, and across the Pacific. Some web servers of universities may be configured to ignore ping messages but it is not difficult to find many that will respond. For the 7 destinations, use a map to roughly estimate their physical distance from Purdue. Use SOL to calculate lower bounds of RTT for these destinations. Compare the lower bounds against the estimates obtained through ping. For each of the destinations, give your thoughts on what may be the main factors that result in the discrepancy. For example, if your distance estimate from Purdue is based on straight line

distance, then the fact that communication lines follow indirect routes through major communication hubs might be one such factor. Whether this is a major factor you may determine by overestimating the physical distance from Purdue and checking if SOL latency is significantly impacted. Submit your answer in a pdf file, lab1.pdf, under lab1/.

The Bonus Problem is completely optional. It serves to provide additional exercises to understand material. Bonus problems help more readily reach the 45% contributed by lab component to the course grade.

Turn-in instructions

Electronic turn-in instructions:

We will use turnin to manage lab assignment submissions. Go to the parent directory of the directory lab1/ where you deposited the submissions and type the command

turnin -v -c cs536 -p lab1 lab1

This lab is individual effort. Please note the assignment submission policy specified on the course home page.

Back to the CS 536 web page