Washington State University Vancouver

Systems Programming - CS 360

Final Project

Instructor:
Ben McCamish

Overall Assignment - 100 points

- The miniature ftp system consists of two programs (executables), `mftpserve` and `mftp`, the server and client respectively.
- Both programs will be written in C, to execute in the school's Linux OS environment.
- The programs will utilize the Unix TCP/IP sockets interface (library) to communicate with each other across a TCP/IP network. Only IPv4 addressing will be supported by these programs.
- A demonstration of your programs is required. The instructor will direct the demonstration and observe program behavior.

User Interface

Command line

- The user initiates the client program using its name (mftp) with one required command line argument (in addition to the name of the program). This argument will be interpreted by the client program as the name of the host on which the mftpserve process is running and to which the client will connect. The hostname may be either a symbolic host name or IPv4 dotted-decimal notation. Optional debug related flags may be defined for the command line but must not interfere with the user's ability to specify the name of the remote host. The client will use some server port to make its connection. You might consider using a random port to avoid conflicts with other students on the lab server.

Login

- The server process will not require a login dialog from the client. The server process will retain the identity and privileges of the user who initiated it and these privileges, identity and initial working directory are the context in which client commands are executed by the server.

User Commands

- Following a successful connection to the server, the client program will prompt the user for commands from standard input. Commands are regarded as a series of tokens separated by one or more white-space characters and terminated by a new line character. The first token is the name of the command. Successive tokens are command arguments, if any. Note: the first token may be preceded by spaces. White-space characters are defined to be any character for which `isspace()` returns true. Commands and their arguments will be case-sensitive.
- The user commands are:

  1. `exit`: terminate the client program after instructing the server's child process to do the same.
  2. `cd <pathname>`: change the current working directory of the client process to `<pathname>`. It is an error if `<pathname>` does not exist, or is not a readable directory.
  3. `rcd <pathname>`: change the current working directory of the server child process to ¡pathname¿. It is an error if ¡pathname¿ does not exist, or is not a readable directory.
  4. `ls`: execute the "ls –l" command locally and display the output to standard output, 20 lines at a time, waiting for a space character on standard input before displaying the next 20 lines. See "rls" and "show".
  5. `rls`: execute the "ls –l" command on the remote server and display the output to the client's standard output, 20 lines at a time in the same manner as "ls" above.
  6. `get <pathname>`: retrieve `<pathname>` from the server and store it locally in the client's current working directory using the last component of `<pathname>` as the file name. It is an error if `<pathname>` does not exist, or is anything other than a regular file, or is not readable. It is also an error if the file cannot be opened/created in the client's current working directory.

7. `show <pathname>`: retrieve the contents of the indicated remote `<pathname>` and write it to the client's standard output, 20 lines at a time. Wait for the user to type a space character before displaying the next 20 lines. Note: the "more" command can be used to implement this feature. It is an error if `<pathname>` does not exist, or is anything except a regular file, or is not readable.
8. `put <pathname>`: transmit the contents of the local file `<pathname>` to the server and store the contents in the server process' current working directory using the last component of `<pathname>` as the file name. It is an error if `<pathname>` does not exist locally or is anything other than a regular file. It is also an error if the file cannot be opened/created in the server's child process' current working directory.

Notes:

- The client program should report command errors and operational errors in a manner that is clear and understandable to the user. If the error is recoverable, the command is aborted and a new command prompt is issued. If the error is not recoverable, the client should terminate.
- The get and put commands are expected to function equally well with both text and binary file data.

Server

The server process, `mftpserve`, will be initiated with no command line arguments other than optional flags related to debugging. The server will log its activity to stdout and its errors to stderr.

Server/Client Interface Protocol

1. The mftpserve process shall listen on the same ephemeral TCP port number as the client for connections. It shall permit a queue of 4 connection requests simultaneously (see `listen()`).
2. The client program (mftp), prior to prompting the user for commands shall establish a connection to the server process on TCP port. If the connection fails, the client will issue an appropriate error message and abort.
3. When a connection is established, an appropriate success message is displayed to the user by the client and logged to stdout by the server, displaying the client's name. This connection is regarded as the control connection. A second connection, the data connection, is required by certain other commands. The data connection will be established when needed and will be closed at the end of the associated data transfer for each command that establishes it.
4. Once the control connection is established, the server process (child) will attempt to read commands from the control connection, execute those commands as described below, and then read another command, until a "Q" command is received, or until an EOF is received, in which case the server child process will terminate.
5. Server commands have the following syntax:
   - No leading spaces.
   - A single character command specifier.
   - An optional parameter beginning with the second character of the command (the character following the single character command specifier) and ending with the character before the terminator.
   - The command terminator is either a new line character or EOF.
6. Server response syntax:
   - The server responds exactly once to each command it reads from the control connection.
   - The response is either an error response or an acknowledgement.
   - All responses are terminated by a new line character.
   - An error response begins with the character "E". The intervening characters between the "E" and the new line character comprise a text error message for display to the client's user (to be written to standard output by the client).

- An acknowledgement begins with the character "A". It may optionally contain an ASCII-coded decimal integer between the "A" and the new line character, if required by the server command (only the "D" command below requires such an integer in the acknowledgement).

7. The server control connection commands are:

   - "D" Establish the data connection. The server acknowledges the command on the control connection by sending an ASCII coded decimal integer, representing the port number it will listen on, followed by a new line character.
   - "C<pathname>" Change directory. The server responds by executing a `chdir()` system call with the specified path and transmitting an acknowledgement to the client. If an error occurs, an error message is transmitted to the client.
   - "L" List directory. The server child process responds by initiating a child process to execute the "ls - l" command. It is an error to for the client to issue this command unless a data connection has been previously established. Any output associated with this command is transmitted along the data connection to the client. The server child process waits for its child (ls) process to exit, at which point the data connection is closed. Following the termination of the child (ls) process, the server reads the control connection for additional commands.
   - "G<pathname>" Get a file. It is an error to for the client to issue this command unless a data connection has been previously established. If the file cannot be opened or read by the server, an error response is generated. If there is no error, the server transmits the contents of <pathname> to the client over the data connection and closes the data connection when the last byte has been transmitted.
   - "P<pathname>" Put a file. It is an error to for the client to issue this command unless a data connection has been previously established. The server attempts to open the last component of <pathname> for writing. It is an error if the file already exists or cannot be opened. The server then reads data from the data connection and writes it to the opened file. The file is closed and the command is complete when the server reads an EOF from the data connection, implying that the client has completed the transfer and has closed the data connection.
   - "Q" Quit. This command causes the server (child) process to exit normally. Before exiting, the server will acknowledge the command to the client.

8. Each command is logged by the server, along with the arguments and its success or failure, to standard output.

Suggestions

- In response to a "D" command, the server child process will create a new socket using `socket()` and then use `bind()` to "give it a name". The family, port, address structure given to bind, should use a port number of zero, which is a wildcard for "pick any available ephemeral port". Following the `bind()` call, use `getsockname()` to obtain an address structure in which the ephemeral port selected and assigned by the kernel will be found. You will need to pass `getsockname()` an empty (zeroed) servaddr structure and a pointer to an integer containing its length. Upon return from `getsockname()`, `servaddr.sin_port` will contain the port number assigned by `bind()`. However, you will need to use `ntohs()` when storing it into an int so that its byte order is correct. This is the port number you will send in the command acknowledgement to the client. The client will then make a connection using `socket()` and `connect()` to that port on the server while the server listens (calls `accept()`) waiting for a the data connection to be made. Once made, this connection/socket is the data connection.

- Consider using `strtok()` to parse the client's user commands.

- Consider using `fork()` and `execlp()` to run the "more -20" command as a child process in the client, in order to obtain the output scrolling behavior needed for the "ls", "rls" and "show" commands. I/O redirection will be needed to route the data coming from the server process via the data connection to the process running "more -20".

- When sending and receiving commands and acknowledgements via the control connection, consider that `read()` and `write()` do not comprehend null-terminated strings. Also, do not assume that a single `read()` from the socket will receive an entire command or acknowledgement, your program must continue reading until it detects the termination character (a newline character).

- Be sure to have your server parent process eliminate terminated child processes that would otherwise accumulate as zombies. The `waitpid()` system call can be used for this purpose, consult the man pages for appropriate parameters.

- Consider defining a debug flag for both your programs such that when in debug mode, your programs print out all the details of what they are doing. If you want to be elegant about it, look for `argv[1]` to be "-d" to turn on debug mode (of course, in that case, the client will find the hostname as `argv[2]`). When you demonstrate and submit your programs, your debug printing should be disabled.
- Test for errors in user input, command input and from system calls and give sensible error messages to the user.
- Test your client and server against the client and server programs of your classmates, they should all be interoperable if everyone follows this specification. Be careful not to clobber each other's files when you do so, these programs offer no protections.
- You may also run my implementations of these programs (which reside on the website) and test your programs against them. Both of my programs will produce extensive debug output if you make the first command line argument "-d". I highly recommend doing this, as your programs must interface with mine during the demo.

Completion of this assignment

1. Organize your code into (at least) three source files:
   - `mftp.c` – client code
   - `mftpserve.c` – server code
   - `mftp.h` – header file with declarations common to both programs
   - Include a Makefile which will build both executables by default
2. When you believe your programs are completed, submit them to me on Canvas.
3. Schedule time with me to demonstrate your programs in the lab. (Details coming)
4. The demo will take place on the server. You may only use the terminal to view and demonstrate you code. If you are unfamiliar with terminal editors, then I suggest practicing a bit before the demo with your favorite flavor.
5. Your demo will consist of me giving you instructions on what to do in addition to me quizzing you on the content of your code.
6. You will be graded on effectiveness, quality (style), and some short questions during the demo.
7. Submission and demonstration of the programs are due by 11:59PM PDT, Sunday, April 23, 2021.