$Id: asg4-client-server.mm,v 1.30 2021-05-06 13:14:29-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cse111-wm/Assignments/asg4-client-server
URL: https://www2.ucsc.edu/courses/cse111-wm/:/Assignments/asg4-client-server/

1. Overview

This project will implement a client/server application using sockets. A daemon (**cxid**) listens on a socket for client connection requests. Each connection will cause the server to fork a child process to serve the client. The daemon runs in an infinite loop listening. The server exits when the client disconnects. A client (**cxi**) connects to a server and can send files, receive files, and get a summary listing of all files present.

A socket is a two-way means of communication between processes, not necessarily running on the same host. An IPv4 host is know by a 4-octet sequence such as 128.114.108.152, and a port is an unsigned 16-bit number (0 to 65535). Communication will be done via TCP/IP over IPv4 sockets.

2. Programs

In this project, two main programs are to be written for the three parts of the project : the daemon and server, and the client. There will also be several library files written to be used by the programs. The general function is similar to **sftp**(1).

cxid

Usage : **cxid** *port*

Creates a server socket and goes into an infinite loop : When it accepts a client socket, it uses **fork**(2) to create a child process, which functions as the server to communicate with the client. The daemon listens for connections on the given port.

The server is forked with an open socket communicating with the client. Its loop repeatedly reads commands and information from the client and acts on those commands, returning information back to the client. Its loop will be a receive followed by a send, responding to client requests. It exits when the client closes the socket. It does no terminal I/O except for debugging purposes.

cxi

Usage : **cxi** *host port*

The client interacts with the user. Commands are read from the terminal (or redirect), each of which is executed one at a time by communicating with the server. Results are then displayed at the terminal. The client connects to the given host and port.

3. Interactive Commands

The **cxi** client responds to commands read from the standard output and writes output to the standard output and error and accesses files. In the syntax below, **Courier Bold** are literal characters actually typed in, while *Roman Italic* stands for appropriate substitutions.

`exit`
Quit the program. An end of file marker or Control/D is equivalent.

`get` *filename*
Copy the file named *filename* on the remote server and create or overwrite a file of the same name in the current directory.

`help`
A summary of available commands is printed.

`ls`
Causes the remote server to execute the command **`ls -l`** and prints the output to the user's terminal.

`put` *filename*
Copies a local file into the socket and causes the remote server to create that file in its directory.

`rm` *filename*
Causes the remote server to remove the file.

4. Protocol used by the `cxi*` programs

For the client and server to communicate, a protocol needs to be established. Each message must be framed in terms of a header and a payload. The header always consists of a **`struct`** of size 64 bytes. All messages between client and server consist of these 64 bytes, possibly followed by a payload. For alignment purposes, the **`nbytes`** field is first. Before filling in the fields, use **`memset`**(3) to clear the struct.

```
enum class cxi_command : uint8_t {
   ERROR = 0, EXIT, GET, HELP, LS, PUT, RM, FILEOUT, LSOUT, ACK, NAK
};
size_t constexpr FILENAME_SIZE = 59;
struct cxi_header {
   uint32_t nbytes {0};
   cxi_command command {cxi_command::ERROR};
   char filename[FILENAME_SIZE] {};
};
```

The purposes of the fields are as follows:

`uint32_t nbytes;`
The number of bytes in the payload if there is any payload. Otherwise it must be zero (MBZ). This field is sent in network byte order and so must use the functions **`ntohl`**(3) and **`htonl`**(3) when loading and storing data.

`cxi_command command;`
A single byte containing one of the **`cxi_command`** constants. Note that the **`enum`** is specifically a **`uint8_t`** single byte type.

`char filename[59];`
The name of the file being transferred or removed. The filename may not have any slash (`'/'`) characters in it and must be null-terminated (with `'\0'`). All bytes following the null must also be null. Pathnames with slashes and filenames longer than 58 characters are prohibited.

Following are the meanings of each of the **`cxi_command`** values. Each is either client to server (C→S) or server to client (S→C), but never both.

`cxi_command::ERROR`
: An error flag to indicate an invalid header. Used internally.

`cxi_command::EXIT`
: Internal to **`cxi`**, not used in communication.

`cxi_command::GET` (C→S)
: Request a file from the server. The filename is used both remotely and locally. The payload length is 0.

`cxi_command::HELP`
: Internal to **`cxi`**, not used in communication.

`cxi_command::LS` (C→S)
: Request file (**`ls`**) information. The payload length and filename are zeroed.

`cxi_command::PUT` (C→S)
: The length of the payload is the number of bytes in the file. The contents of the file immediately follow the header. The bytes of the payload are unstructured and may contain null bytes. Binary files are acceptable.

`cxi_command::RM` (C→S)
: Request to remove a file. The payload length is 0.

`cxi_command::FILEOUT` (S→C)
: Response to a **`cxi_command::GET`**. The filename is the same as in the request and the payload length reflects the number of bytes in the file. The payload consists of the bytes of the file.

`cxi_command::LSOUT` (S→C)
: Response to a **`cxi_command::LS`**. The filename is zeroed and the payload length is the number of bytes sent in the payload. The payload is the output of the command **`ls -l`**.

`cxi_command::ACK` (S→C)
: Response to either a **`cxi_command::PUT`** or a **`cxi_command::RM`** indicating that the request was successfully completed.

`cxi_command::NAK` (S→C)
: Response to any request that fails. There is no payload. The filename field is the same as was in the original request. The nbytes field is set to the value of **`errno`** in the server's attempt to preform a task.

5. Procedures

Each of the above commands requires procedures for accessing files, including reading files from disk and writing files to disk, as well as accessing directories. When any of the system calls fails in the server, the server immediately terminates the operation and sends the value of **`errno`** back to the client in a **`cxi_command::NAK`** message.

(a) For the client or server to send a file it must first be read into a buffer. Binary files must be properly handled, so protocols which assume text files won't work. To load a file from disk, use **ifstream::read()**, collecting characters into a buffer. Read the entire file into a buffer then close it. After that, it may be sent down the socket.

(b) Alternatively, **stat**(2) the file to see how large it is, and send the file down the socket piecemeal. In conjunction with **stat**(2), it is also possible to may the file into memory using **mmap**(2), provided that the memory thus acquired is released with **munmap**(2) when the file operation is complete.

(c) When receiving a file from the socket, Receive the header and determine the size of the file. Create an **ofstream** and use **ofstream::write()** to write the parts of the file as they are received from the socket. A C++ stream is closed when the variable goes out of scope, or you can call **close**.

(d) To delete a file for the **cxi_command::RM** command, use **unlink**(2):

```
rc = unlink (filename);
```

(e) To execute the **cxi_command::LS** command use **popen**(2) and **pclose**(2) to create a pipe stream from the **ls**(1) command

```
FILE* pipe = popen ("ls -l", "r");
```

Then read the characters from the pipe in the easiest way, probably by using **fgets**(3). Finally, **pclose(pipe)**. Then send the output back the client in a **cxi_command::LSOUT** message.

6. Modules

There will need to be several modules in this suite of programs. Each of the programs, of course, will have its own source file with a main function in it.

The **sockets** module will be a useful inclusion into the program as its own module.

There should also be a **cxi_protocol** module to implement the protocols and contain code for accessing files and sockets, since these will be used by both the client and the server.

7. Use of sockets

The sysadmin operating **unix.ucsc.edu** has blocked all ports except for the ssh port for security reasons, so you will not be able to run the server on one of the Linux servers and your client on your own machine. When testing your program, open two terminals on the ***same*** host. You may log into any host to run your server, then start it up. Then, in another window, log into the same host to run the client. The client and the server must be run in different directories. Use the **hostname**(1) command to discover which host you are logged into. Example:

```
-bash-71$ hostname
unix2.lt.ucsc.edu
```

The name **unix.ucsc.edu** is not actually a real host. It is just an alias for one of the real hosts, which have name patterns like the one shown here.

8. Use of ports

If your daemon listens on a port that has been bound by another process, you will get the message "Address already in use" because only one process at any given time is allowed to listen on a particular process. To avoid this, choose a port number not being used by anyone else on the same server.

Port numbers are divided into three ranges :

(a) Ports 0 through 1023 (0x3FF) are well known. They are associated with services in a static manner and are reserved to privileged processes. A list of these services may be found in the file **`/etc/services`**.

(b) Port numbers from 1024 (0x400) through 49151 (0xBFFF) are registered. They are used for multiple purposes. Choose any of these ports for your server if not already in use.

(c) Dynamic and private ports are those from 49152 (0xC000) through 65535 (0xFFFF) and services should not be associated with them.

9. Runaway Processes

Be careful in using **`fork`**(2) so that you don't accidentally create a fork-bomb. The command **`pkill`**(1) can be used to kill all processes matching a particular pattern. So the command

```
pkill cxi
```

will kill all of your processes whose executables contain the string "**`cxi`**". A really quick way to log out is to use **`kill`**(1) :

```
kill -9 -1
kill -s KILL -1
```

will send **`SIGKILL`** to all of your processes, thus logging you out.

10. What to Submit

Submit **`Makefile`** which builds both programs, all necessary C++ header and implementation files. And if doing pair programming, the **`PARTNER`** file. When the grader uses the command **`make`** in the submit directory, the both binaries should be built.