ECE/CS 438: Communication Networks **Fall 2007**
Machine Problem 1 **Due: 9:00 p.m. Friday, September 21**

Transferring Files over TCP

Please read all sections of this document before you begin coding.

In this machine problem, you will develop a client-server application to transfer files to another computer. The client will accept interactive commands to obtain a directory listing or download a particular file and interact with the server to execute them. The server will accept concurrent connections from multiple clients and serve them all correctly.

Objectives

The primary goals of this machine problem are for you to learn to build simple client-server applications using the sockets API and for you to understand the process of sending messages over TCP. Servers generally manage multiple connections simultaneously, and you should also learn the basics of concurrency from this problem.

Guidelines

Please work on this MP individually. Collaboration, discussion, code sharing, and any other form of group work is forbidden. Offenders will be dealt with appropriately (see the general information sheet available from the class home page).

You are required to make sure that your programs compile on the `{dcllnx,gllnx,eelnx}*.ews.uiuc.edu` machines and to use the EWS lab. Programs suffering compilation errors when tested by the course staff will earn no credit.

You may find it useful to read Chapters 1-4, 6, and 8 in Stevens. Refer to the Unix man pages as necessary. You may particularly want to look at man pages for `opendir`, `readdir`, `stat`, and `getpwuid` in order to generate the directory listing

Please indent and document your code. Use meaningful names for variables and follow other style guidelines that enhance the clarity of your code.

Follow the guidelines in the "Hand In" section below when turning in this assignment.

Specification

Make a directory called MP1 somewhere inside your main directory to hold your code. Call the file for your server code `server.c` and call the executable `server`. Follow the same convention for the client. Note that the source file names are recommendations; in contrast, the executable file names are requirements. Create a Makefile that automatically generates the executables from your sources.

This problem consists logically of three components: a client thread, a main server thread, and a per-client server thread. This document refers to the problem components as threads, and you are encouraged to use Posix threads to implement them (see Chapter 23 of Stevens). In later machine problems, threads will become a requirement. However, for this problem, separate Unix processes are also acceptable. A third approach to supporting multiple clients is to develop your own connection-specific context and to use a single thread of control.

Client Specification:

The client will be started with the following command:

```
client serveraddress serverport
```

where `serveraddress` and `serverport` are the address and port where the server will be running. The client will then wait for input from the user, which will consist of the following commands:

- `dir`: Generate a listing of files in the current directory on the server.

- `get file`: Download a copy of `file` from the server and save it to the current directory. If a file does not exist on the server, display an error message.
- `quit`: Close the connection to the server and exit.
- `shutdown`: Ask the server process to close all connections and exit. The client should also exit at this point.

You should display a prompt "> " before accepting each command. If an invalid command is entered, you should display an error message and wait for the next command.

Server Specification:

The server will be started with the following command:

```
server serverport
```

It will listen for connections from clients on port `serverport`. The clients will send commands according to the protocol you designed and your server must respond to them appropriately. The server must be able to concurrently communicate with several clients. If a client sends a `shutdown` command, the server should close connections to all clients and exit.

Note: you should run your server and client in different directories, otherwise your client will be trying to overwrite files that the server is reading.

Listing Format:

The directory listing **must** follow the format below exactly. You will be graded on the correctness of your file listing and you will lose marks for deviations.

```
==== BEGIN DIRECTORY LISTING ====

file1 nikita 1234
file2 zhuang21 52333
old.file root 15

==== END DIRECTORY LISTING ====
```

In particular, your listing must begin and end with the header and footer, exactly as above, with a blank line after the header and before the footer. The listing contains three fields: the file name, the username of the file owner, and the file size (in bytes). The fields are separated by a tab character ("`\t`") and each line is terminated by a newline ("`\n`"). The files should be displayed in an alphabetically sorted order.

Protocol Documentation:

You will need to design a protocol for communicating between the server and the client. This protocol should be specified (at the byte level) in a design document you hand in, and your server and client *must* follow the protocol during their interactions. Your implementation of the protocol should account for byte order differences; you can test this by trying to compile your program on a Sun machine (`dclsn*` or `glsn*`) and downloading files from there.

For a bonus challenge, you can design your protocol to be more efficient when it transmits a directory listing. The top few projects in the class will receive bonus points. However, make sure that your code functions correctly first; no bonus points will be given to projects that do not correctly produce listings.

Refragmentation and Logging: One very important aspect of TCP byte streams that many people find difficult to grasp is the fact that calls to read and write (or send/recv, etc.) are not correlated. In particular, two or more writes to a connection can be received by the same read, and a single write can be split amongst several reads. The EWS environment makes this lesson harder because, on a local area network, the timing issues make it seem like correlation works.

To ensure that you gain a solid understanding of framing application data on a TCP connection, we have provided a routine to make the environment much less predictable. In particular, you must use the interface below in place of `read` for all of your MP1 client code (no calls to any other routine to read data from a socket are allowed):

```
size_t mp1_read (int fildes, void* buf, size_t nbyte);
```

The `mp1_read` function has exactly the same format and semantics as read[1], but will almost always split the contents of each write across several reads, and may also merge pieces of multiple writes into a single read. Only the order of the bytes is preserved, as is effectively the case with `read`.

Use `mp1_read` just as you would use `read`; you probably want to write a wrapper function (see Stevens p. 78). The header file `/homesta/ece438/MP1/mp1.h` contains a function prototype for `mp1_read`. Link the `mp1.o` file *from the class directory* into your client and server executables. For example, the link command for your server might appear as:

```
gcc -o server server.o /homesta/ece438/MP1/mp1.o -lpthread
```

You must link the file in the class directory rather than making a local copy to guarantee proper behavior when your assignment is graded. The `mp1_read` function is not thread-safe, but your client needs only one thread to read data from the server. For interested students, commented source code is available in the same directory as the object file.

In addition to performing refragmentation, the MP1 class routine records the number of calls made to it and the number of bytes returned in the file `mp1_results`.

Testing and Grading

You are responsible for testing your code to ensure that it complies with the specifications given in this document. This section is intended to give you some things to think about when testing, and to warn you about some pitfalls that people have encountered in past classes.

It may help (or amuse you) to think of the TA's as your customers, and your grade as a customer satisfaction rating.

For directory listings, you must follow the exact format specified, and you will lose points for deviating from the format.

The limits on the fields in the listing and file sizes are determined by the system; please make sure you can support the maximum values of these fields.

Hand In

Place your source code in your MP1 directory along with a Makefile that, on execution of the command `make`, causes the executable code for the programs `client` and `server` to be generated by the compiler.

Finally, create a file README containing two parts.

- *Part 1: Implementation Log*
  This part should briefly describe and justify your design and implementation decisions, including your approach to concurrency. Discuss any other interesting aspects of your implementation. As always, you should acknowledge the sources if you copied sections of your code from outside courses (even if you modified it later).
- *Part 2: Protocol Design*
  Provide a detailed description of the protocol that is used to communicate between the client and the server, with a byte-level specification of all messages.

When you get the C programs, README, and Makefile files within the directory MP2 in good working order, you are to transfer the directory electronically as follows. First, remove all object files and executables from the directory, leaving only the source code, the Makefile, and the README file. **Submission of executables will reduce your grade.** Next, from within the directory, type:

[1] The class routine does not respect Posix minimum fragmentation guarantees, but the impact on your code because of this difference is negligible.

`~ece438/Handin/handin`

Follow the prompts and hand in the C programs, the README file and Makefile. (Please check the course newsgroup for tips in case our handin program isn't running smoothly.)

Grading Guidelines

10 pts Code

- Well commented
- Meaningful variable and function names
- Readable and indented

10 pts Implementation Log

- Acknowledge sources

20 pts Protocol Design

- Complete and readable specification

10 pts Makefile/Executables

- Executables and source files have correct names
- No extra files handed in
- No errors in generating executables with `make`

15 pts Client and server interface

- Correctly responds to commands
- Produces error messages
- Client quit and server shutdown correctly implemented

10 pts File download

- File correctly transferred
- Error message for non-existent file

10 pts Directory listing

- Correct listing produced
- Format follows course specification

15 pts Concurrency

- Server can handle multiple concurrent clients

5 pts (bonus) Performance

- Efficient encoding of a directory listing