Programming Project 2: Multithreaded FTP Client and Server

The aim of this project is to introduce to you the design issues involved in multi-threaded servers. In this project, you are going to extend your first project to make both the client and server multithreaded. The client will be able to handle multiple commands (from same user) simultaneously and server should be able to handle multiple commands from multiple clients. The overall interaction between the user and the system is similar except as indicated below. The client and server will support the same set of commands as indicated in project 1 (get, put, delete, ls, cd, mkdir, pwd, quit). In addition, they will support one more command called "terminate", which is used for terminating a long-running command (e.g., get and put on large files) from the same client.

The syntax of the terminate command is as follows:

```
terminate <command-ID> -- terminate the command identified by
<command-ID> (see below for a description of command ID).
```

The workings of the client and server are described below:

FTP Server (myftpserver program) - The server program takes two command line parameters, which are the port numbers where the server will wait on (one for normal commands and another for the "terminate" command). The port for normal commands are henceforth referred to as "nport" and the terminate port is referred to as "tport". Once the myftpserver program is invoked, it should create two threads. The first thread will create a socket on the first port number and wait for incoming clients. The second thread will create a socket on the second port and wait for incoming "terminate" commands.

When a client connects to the normal port, the server will spawn off a new thread to handle the client commands. The operation of this thread is same as described in project 1. However, when the serve gets a "get" or a "put" command, it will immediately send back a command ID to the client. This is for the clients to use when they need to terminate a currently-running command. Furthermore, the threads executing the "get" and "put" commands, will periodically (after transferring 1000 bytes) check the status of the command to see if the client needs the command to be terminated. If so, it will stop transferring data, delete any files that were created and will be ready to execute more commands.

Note that there might be several normal threads executing concurrency. The server should handle the consistency issues arising out of such concurrency. For example, if two put requests arrive concurrently for the same file, one request should be completed before the other starts.

You should carefully design the server paying close attention to which commands need mutual exclusion and which ones do not.

When a client connects to the "tport", the server accepts the terminate command. The command will identify (using the command-ID) which command needs to be terminated. It will set the status of that command to "terminate" so that the thread executing that command will notice it and gracefully terminate.

FTP Client: The ftp client program will take three command line parameters the machine name where the server resides, the normal port number, and the terminate port number. Once the client starts up, it will display a prompt "mytftp>". It should then accept and execute commands as in Project 1. However, if any command is appended with a "&" sign (e.g., get file1.txt &), then this command should be executed in a separate thread. The main thread should continue to wait for more commands (i.e., it should not be blocked for the other threads to complete). For "get" and "put" commands, the client should display the command-ID received from the server. When the user enters a terminate command, the client should use the tport to relay the command to the server. The client should also clean up any files that were created as a result of commands that were terminated.

Points to note:

1. You can assume that each client has only one connection to the server through the normal port (this will avoid complications with respect to changing directories in one thread while other thread is operating in another directory). However, note that different clients can still concurrently connect to the server.
2. You need to pay special attention to concurrency issues.
3. You can assume that the user uses the correct syntax when entering various commands (i.e., you are not required to check for syntax).
4. You do not need to support user logins. Also you do not need to check for user permissions when executing various commands at the server end.
5. You may use C/C++/Java for your project.
6. If you are using Java, you are not allowed to use any thread-safe data structures.
7. All programs will be tested on Linux platform (nike). It is your responsibility to test and make sure that your program works correctly on a Linux machine.

What to Submit:

A Zip folder containing:

1. Your code for client and server.
2. A readme file containing (a) names of the students in the project group; (b) any special compilation or execution instruction; and (c) the following statement – if it is true –

"This project was done in its entirety by <Project group members names>. We hereby state that we have not received unauthorized help of any form". If this statement is not true, you should talk to me before submission.

3. All submissions will be done on ELC.