Computer Science 367

(Individual) Program 2 (50 points)
Due Wednesday, February 18th, 2015 at 10:00 PM

Read all of the instructions. Late work will not be accepted.

Overview

In our second network programming assignment, you will implement the server and two distinct types of client for a simple online chat application. The first type of client is an *observer*. Observer clients receive and display a stream of messages from the server in real-time; they do not send any data. The second type of client is a *participant*. Participants send messages, but do not receive anything. In a more sophisticated program, both of these clients would be integrated into a single display. To simplify the implementation of Program 2, we treat these separately. The server should be able to support up to 64 participants, and an equal number of observers. The following shows an example output of an observer client.

```
A new observer has joined
User 0 has joined
User 1 has joined
 1: hi
 0: hey
User 2 has joined
 2: hello!
 1: hi
 1: bye
User 1 has left
 2: where'd user 1 go?
 0: no idea
User 1 has joined
 1: I'm back!
 2: welcome back
 0: thanks
User 2 has left
User 0 has left
 1: hello?
```

The primary challenge of this program is that a range of events can happen at arbitrary times, and your program must be able to respond to them immediately. At any given time, a new participant or observer could join, a participant or observer could quit, or any of the participants could send a message. To accommodate this, you must use the `select` command in your implementation of Program 2. `select` is a command that allows a program to monitor several socket (or file) descriptors, and return as soon as any of them are ready to read (or write, or report an error). Letting `select` babysit your socket descriptors permits you to react immediately to the aforementioned events.

You are responsible for implementing (in the C programming language) the server and both clients for this program, using sockets. It must be developed in your Subversion version control repository.

Program 2 Specifications

Your program must be compliant with all of the following specifications in order to be considered correct. Non-compliance will result in penalties.

File and Directory Naming Requirements

All files should be contained in a `prog2` directory in your repository. Spacing, spelling and capitalization matter.

- The writeup and all of your source code should be found directly in `yourWorkingCopy/prog2`, where `yourWorkingCopy` is replaced with the full path of your working copy.
- Your participant client source code should be contained in a file named `prog2_participant.c`. If you create a corresponding header file you should name it `prog2_participant.h`.
- Your observer client source code should be contained in a file named `prog2_observer.c`. If you create a corresponding header file you should name it `prog2_observer.h`.
- Your server source code should be contained in a file named `prog2_server.c`. If you create a corresponding header file you should name it `prog2_server.h`.
- Your writeup must be a plain-text file named `writeup.txt`.

Command-Line Specification

The server should take exactly two command-line arguments:

1. The port on which your server will listen for participants.
2. The port on which your server will listen for observers.

An example command to start the server is:

```
./prog2_server 36798 36799
```

Either client should take exactly two command-line arguments:

1. The name or address of the server (e.g. `linux.cs.wwu.edu` or a 32 bit integer in dotted-decimal notation)
2. The port on which the server is running, a 16-bit unsigned integer

An example command to run the participant client is:

```
./prog2_participant 127.0.0.1 36798
```

An example command to run the observer client is:

```
./prog2_observer 127.0.0.1 36799
```

Compilation

Your code should be able to compile cleanly with the following commands on CF 416 lab machines:

```
gcc -o prog2_server prog2_server.c
gcc -o prog2_observer prog2_observer.c
gcc -o prog2_participant prog2_participant.c
```

Protocol Specification

The protocol is summarized by the following rules:

- Immediately after any participant client connects to the server...
  - The server should assign the participant to the lowest available user number.[1]
  - If the limit of participants has been reached, the server simply closes the socket.
  - If the limit has not been reached, the server stores the corresponding socket and sends a message to each observer saying `User X has joined` (where `x` is replaced by the newly assigned user number). The message sent should include the final newline but **not** the null terminator. Until this user quits, he or she should be able to send chat messages at any time.
- Immediately after any observer client connects to the server...
  - If the limit of observers has been reached, the server simply closes the socket.
  - If the limit has not been reached, the server should add this observer to the set of observers and send a message to each observer (including the new one) saying `A new observer has joined`. The message sent should include the final newline but **not** the null terminator.
- Immediately after any message is received by the server from a participant...
  - The server should read all available bytes from the participant client, up to `1024` characters. Note that this message from the client **may** end with a newline character, but should **not** include the null terminator.
  - The server should prepend `XX: ` to the message, where `XX` is the user id and a space follows the colon. Since only 64 users are possible, two decimal places is sufficient.
  - If the original message does not end with a newline, the server should append one.
  - The server sends this modified message to each observer. The message sent should include the final newline but **not** the null terminator.
- Immediately after any message is received by an observer client it should print it to standard out. Note that the incoming messages will not have the null terminator, so some care must be taken to display the messages properly. The incoming messages will have newlines, so this client should not add any newlines. An observer client never calls `send`.
- A participant client should repeatedly read from standard input one line at a time. As each line is read, it should be sent to the server. The line should include the final newline but not include the null terminator. A participant client never calls `recv`.

[1]As users quit, their user numbers become available for future participants to reuse.

The Repository

- All code for this program will be developed under a Subversion repository. You should have already have one at this point. Please continue to use it (note that the numbering will pick off where you left off – this is fine).
- As noted above, your work will be done in a `prog2` sub-directory of your working copy. Therefore, exactly once at the beginning of working on Program 2, you will need to create this directory and `svn add` it to your repository. You will also need to `svn add` your writeup and every source code file to your repository. **If you do not add and commit your files, I cannot see them, and thus cannot give you any points for them.** You can use the `subversion_check.sh` script to help determine if all of your files have been successfully added and committed.
- You must actively use Subversion during the development of your program. If you do not have at least 5-8 commits for Program 2, points will be deducted.
- See the Subversion Guide on Canvas for all sorts of details on Subversion.
- *Note: you must never edit files in your repository directly. All interaction will be indirect, via your working copy.*

Grading

Submitting your work

When the clock strikes 10 PM on the due date, a script will automatically check out the latest committed version of your assignment. **(Do not forget to add and commit the work you want submitted before the due date!)** The repository should have in it, at the least:

- `prog2_observer.c`, `prog2_participant.c` and `prog2_server.c`
- Your write-up: `writeup.txt`
- Optionally, your header files

Your repository need not and **should not contain your compiled binaries / object files.** Upon checking out your files, I will compile all three of your programs, run them in a series of test cases, analyze your code, and read your documentation.

Points

By default, you will begin with 50 points. Issues with your code or submission will cause you to lose points, including (but not limited to) the following issues:

- Lose fewer points:
  - Not including a writeup or providing a overly brief writeup
  - Poor code style (inconsistent formatting, incomprehensible naming schemes, etc.)
  - Files and/or directories that are misnamed
  - Insufficient versioning in Subversion

- Lose a moderate to large amount of points:
  - Not including one of the required source code (`.c`) files
  - A server that blocks when there is an event to handle (e.g. a new message)

  - Not using `select`
  - Server cannot handle 64 participant clients and 64 observer clients concurrently
  - Code that does not compile
  - Code that generates runtime errors
  - A client or server that does not take the correct arguments
  - A client or server that does not follow the specified protocol

Write-Up

You need to create, add and commit a *plaintext*[2] document named `writeup.txt`. In it, you should include the following (numbered) sections.

1. Your name
2. Declare/discuss any aspects of your client or server code that are not working. What are your intuitions about why things are not working? What issues you already tried and ruled out? Given more time, what would you try next? Detailed answers here are critical to getting partial credit for malfunctioning programs.
3. In a few sentences, describe how you tested that your code was working.
4. What was the most challenging aspect of this assignment, and why?
5. What variant/extension of this assignment would you like to try (e.g. a variant that is more powerful, more interesting, etc.)

Academic Honesty

To remind you: you must not share code with anyone other than your professor: you must not look at any one else's code or show anyone else your code. You cannot take, in part or in whole, any code from any outside source, including the Internet, nor can you post your code to it. If you need help from anyone, all involved parties *must* step away from the computer and *discuss* strategies and approaches - never code specifics. I am available for help during office hours. I am also available via email (do not wait until the last minute to email). If you participate in academic dishonesty, you will fail the course.

[2]E.g. created with vim, kate or gedit.