Project Three: Chat Program

CS314 Operating Systems
Fall 2006
Date Assigned: 10-04-2006
Date Due (Chat Only): 10-25-2006 @ 11:59:59 PM
Date Due (Chat w/ Race Conditions Fixed): TBA

BACKGROUND:

In Project 2 you built a fairly sophisticated shell program. The shell supported not only the execution of any command, but had sophisticated i/o redirection to files and other programs via pipes. This level of sophistication was achieved through interprocess communication.

For this project, you are going to take IPC to the next level by creating an internet enabled chat application. By the time you are done, your chat program will allow buddies across the WORLD to talk to each other.

In order to prepare for this program you need to do a couple of things:

1) Review book and class notes regarding sockets.

2) Review the socket tutorial on the course website

3) Review book and class notes regarding pthreads

4) Review the "Prime Rib Server" pthread version on the course website

Please start the project now and work on it over the next several weeks. You’ll be glad that you did!

PROJECT DESCRIPTION:

You are going to build a working chat system consisting of the following:

**Chat Server** - Maintains a list of all chat clients or "buddies" in the system. There is only ONE chat server for the entire chat system. That chat server can run on any machine and all chat clients must know the IP address of the chat server machine.

**Chat Client** - Registers with chat server upon initialization. Allows a chat user to communicate with other chat clients. There can be many chat clients running at the same time.

**Chat Screen** - Displays messages that the local chat client has sent to remote chat clients. Displays messages from remote chat clients to the local chat client. There is a chat screen for each chat client in the system.
**Chat Server Details:**

The chat server is an independent process on a single machine with the following invocation:

```
server <serverPort>
```

`<serverPort>` is the port number that the server will be listening on for messages from chat clients.

The model for the design of the chat server is VERY similar to the design of the pthread prime rib server we have examined in class:

1. Wait for connection request from client using `accept()`
2. When connection request received:
   - create a pthread to handle the communication with client
   - main pthread returns to 1

The server's main responsibility is to keep a chat client informed of the whereabouts and status of other chat clients.

The chat server is responsible for responding to the following messages from a chat client:

1. **Register Message:** A chat client lets the server know that it exists and provides a screen name, ip address of the chat screen, port for the chat screen, and status of the chat client. If a chat client already exists with the same screen name, then the server will reject the registration and inform the client of the rejection. The client should then terminate with an error message. If registration is successful, the server adds the chat client information to a list of clients available for chatting.

2. **Unregister Message:** This message is the reverse of the registration message. When a chat client terminates, it must inform the server that the chat client's screen is no longer available to accept messages. The server removes the chat client from the list of clients available for chatting. The chat client provides the client's screen name for this message.

3. **Change Status Message:** A chat client can have one of the following statuses: active, idle, away. When the chat client process starts, the default status is ACTIVE. Using the chat client, the user can change the status at any time. Each time a chat client status changes, the server is informed of the change by providing the client's screen name and new status. The server updates the client's status in its list.

4. **Change Screen Name Message:** The user can change the screen name of a chat client at any time. When the screen name changes, the chat client informs the server by providing the original screen name, and the new screen name. The server will reject the change if a client with the new screen name already exists and inform the client of the rejection.
5 - Get Buddy Details Message: The chat client can request details about any other chat client from the server. The client provides the screen name of the chat client it's interested in and the server provides the buddy's screen name, ip address of buddy chat screen, port of buddy chat screen, and status of buddy chat client. If the buddy doesn't exist the server returns an error message to the client.

6 - Get Buddy List Message: The chat client can request a list of available chat clients from the server. The server will return a list of chat client screen names separated by '\n' in response to this request.

**Chat Client:**

The chat client provides the simple character based interface used to interact with the chat system. The client interacts with the chat screen and chat server by sending and receiving messages. Unlike the chat server, there is one chat client for each user.

The chat client is an independent process which can be executed on a machine with the following invocation:

```
client <screenName> <screenPort> <serverIP> <serverPort>
```

*<screenName>* is a character string representing the chat client's screen name. For example: bob

*<screenPort>* is the port number the chat screen will be listening on for messages from chat clients

*<serverIP>* is the numeric IP address of the machine the chat server is running on

*<serverPort>* is the port number the chat server will be listening on for messages from chat clients

When the chat client starts up, it sends a Register Message to the chat server. Then displays a menu of choices for the user:

0 - Chat with buddies
1 - Change status
2 - List buddies
3 - Change screen name
4 - Exit

**0 - Chat with buddies**

This menu option causes the chat client to prompt the user for a text message to send to another chat client. The text message must be of the format:

```
<buddyScreenName> <text message>
```
When the user presses ENTER, the line the user typed in is parsed. The first string encountered is the `buddyScreenName`. The rest of the line is the message to send to the chat client identified by `buddyScreenName`.

First, the text message is sent from the chat client to its own chat screen process.

Second, a Get Buddy Details Message is sent from the chat client to the server to lookup the details for the buddy identified by `buddyScreenName`. Using these details, the chat client sends a text message to the buddy's chat screen process.

1 - Change status

This menu option causes the chat client to prompt the user to change the status to one of the following:

- 0 - active
- 1 - idle
- 2 - away

Once the new status has been selected, a Change Status Message is sent to the chat server informing it of the client's change of status.

2 - List buddies

This menu option causes the chat client to send a Get Buddy List Message to the chat server. The chat server returns a string which contains a list of all chat client screen names (excluding the requesting chat client) separated by '
' characters. The list is displayed for the user to view.

3 - Change screen name

This menu option causes the chat client to prompt the user for a new screen name. The user types in the new screen name and a Change Screen Name message is sent to the chat server. If the server accepts the screen name, then return to the main menu. If the server rejects the screen name, then prompt for a new screen name again and repeat the process.

4 – Exit

This menu option causes the chat client to send an Unregister Message to the chat server. This removes the chat client from the list of clients available for chatting. The client process also sends a message to its chat screen informing the screen process of the termination. In response, the screen process terminates.

Chat Screen Details:
Displays messages that the local chat client has sent to remote chat clients. Displays messages from remote chat clients to the local chat client. There is a chat screen for each chat client in the system.

The chat screen is an independent process which can be executed on a machine with the following invocation:

```
screen <screenPort>
```

`<screenPort>` is the port number the chat screen will be listening on for messages from chat clients

The model for the design of the chat screen is VERY similar to the design of the pthread prime rib server we have examined in class:

1 - Wait for connection request from client using `accept()`
2 - When connection request received:
   - create a pthread to handle the communication with client
   - main pthread returns to 1

The screen's main responsibility is to display messages from local and remote chat clients and to terminate when it receives a termination message from the local chat client.

The chat screen is responsible for responding to the following messages from a chat client:

1 - Chat Message: This is a text message from either the local or remote chat client. The message contains the `screenName` of the sender, the `screenName` of the receiver, and the text message sent. The chat screen should display this information to the user.

2 - Terminate Message: This is a message from the local chat client indicating that the chat client is terminating. In response to this message, the chat screen process should terminate.

SOME CODING HELP:

To make your life easier, and help you focus on the operating systems concepts in this project I've included a couple of helpful classes.

`buddy.cpp`, `buddy.h` contains a class which represents all the information that needs to be maintained about a chat client in the chat server.

`list.h`, contains a nice list template which you can use in the server to maintain a list of buddies, thanks to Ralph's CS104 class for this template

`testBuddyList` - shows off the use of the buddy class and the list template
client, server, and screen – are executable examples of the chat program you are supposed to build. You can play around with these to get a deeper understanding of the application you are constructing.

This first screen shot shows the command line arguments passed to various components of the chat system. There are two chat clients (bob and becky), a chat screen for each client, and a single chat server.

“./screen 15000” - creates a chat screen for the chat client bob. This chat screen will listen for messages on port 15000.

“./screen 16000” - creates a chat screen for the chat client becky.

“./server 17000” - creates the server that ALL chat clients will connect to. The server will listen for messages on port 17000.

“./client bob 15000 127.0.0.1 17000” – creates a chat client with a screen name of bob. The chat client will send chat messages to the screen listening at port 15000, and update the chat server located at the machine with the numeric IP address 127.0.0.1 and port 17000. Recall from class that I said the machine address 127.0.0.1 is a special “localhost” address which always means the local machine. When you are ready to test your application across multiple machines, then you can find out the internet address of your machine by typing the command “/sbin/ifconfig”.
Pressing enter for each of these commands creates the following display:
Not that the server process on the right is displaying a list of clients that are available for chatting. Now we’ll look at a conversation between bob and becky. “bob” selects option 0 from the chat client menu and sends a message to becky:
Notice that Bob typed Becky’s screen name first, then a message. The message was displayed on both Bob’s and Becky’s chat screens. Now Becky can respond to the message by selecting option 0 from her chat menu and typing something to Bob:
When bob wants to return to the main menu, he types “/q”: 
Build the system in stages:

Stage 1: chat client with all menu options, but menu options are stubbed out
Stage 2: chat screen with ability of chat client to talk to its local chat screen
Stage 3: chat server with ability of chat client to send Register/Unregister Messages
Stage 4: chat server with ability of chat client to send Change Status/Change Screen Name Messages
Stage 5: chat server with ability of chat client to send Get Buddy List and Get Buddy Messages
Stage 6: chat client with ability to send remote messages to remote chat client

RACE CONDITIONS:

The chat server maintains a list of information about each chat client. This list is a SHARED RESOURCE manipulated by the worker threads in the chat server. Since this shared resource can be accessed simultaneously by several worker threads, we have the potential for a race condition.

Describe at least one scenario where a race condition could occur in your chat server program. Place this description in the project’s README.TXT file.
The likelihood that you have seen a race condition with your chat system is pretty small because you can’t effectively test with hundreds of chat users. However, if you were to release your program as the next “AIM”, those race conditions would start happening immediately.

Add a new kind of thread to the chat server called mrSpeedy. The goal of mrSpeedy is to trigger a race condition in your chat server. mrSpeedy should do the following:

```c
<somewhere in chat server main program>
pid_t tid;
pthread_create(&tid,NULL,mrSpeedy,NULL);
...

void* mrSpeedy(void* parameter)
{
    while (true)
    {
        register "Mr. Speedy" as a buddy
        request a list of all buddies
        unregister "Mr. Speedy" as a buddy
    }
}
```

Unlike regular clients, which run as separate processes and use sockets to communicate with the chat server, mrSpeedy is a thread that runs INSIDE the chat server. mrSpeedy doesn’t need sockets to communicate with the chat server since mrSpeedy has access to the server’s data. mrSpeedy will continuously manipulate the chat client list maintained by the chat server. If, while mrSpeedy is running, a chat client contacts the server, there’s a good chance that a race condition will be triggered.

Run your modified chat server with mrSpeedy, create some chat clients and try and trigger a race condition in the chat server. With a little luck.. you’ll see evidence of a race condition. If you can’t get a race condition to occur, then try adding another thread, mrsSpeedy, to the chat server. mrsSpeedy should do the same thing as mrSpeedy. This second active thread will almost guarantee a race condition. Note the results of your race condition experiments in the README.TXT file.

Use POSIX semaphores to eliminate race conditions in your chat server.

DELIVERABLES:

Emailed to bbugan@stonehill.edu the following:

- All the files necessary to compile, link, and run your program.
- An ELECTRONIC document describing how to run the program you created. This document should also include your description of at least one race condition scenario in the chat server. This document should also include your observations as you attempt to create race conditions with the mrSpeedy and mrsSpeedy threads in the chat server. Call this document README.TXT.
- These files should be placed in a directory called “<username>project3”.
- Use the tar command to place all the files in a single file called “<username>project3.tar”. See the first project for instructions on how to do this.
- Email the <username>project3.tar.gz as an attachment to bhuman@stonehill.edu.