Introduction

In this assignment you will be creating a multi-threaded chat server much like Google's Hangouts, called **Petr Chat**. The goal is to learn about network programming and concurrency with threads in C. The client and server will communicate using the **Petr Protocol**, a simple protocol. The base code includes an implementation of a chat client which follows the **Petr Protocol** to use to connect to your server. This will ensure that the protocol is implemented correctly in your server program.

Takeaways

This assignment is an opportunity to bring together all the C programming skills you have learned throughout the term to create a larger project. In addition, this project is more open ended with regard to the implementation approach of the components. This will allow for your creativity to shine!

After completing this assignment you will have experience with and a better understanding of:

- Concurrency
- Thread execution and thread-safety
- Working with the POSIX thread library
- Using mutexes and semaphores
- Producer-Consumer problem
- Working with sockets & network connections

Restrictions & Important Information

In this assignment,

- We strongly recommend that you check the return codes of all system calls (you can use the book wrappers or create your own). This will help you to catch errors.
- Use the debug macros provided in `debug.h`. These were used in the homeworks as an illustration. Follow this approach to ensure your debugging output never interferes with the standard operation of the server.
- Your server should NEVER crash. We will be deducting points for each time your program crashes during grading. Make sure that your code handles invalid usage gracefully.
- Remember to use the man pages! Man pages for the pthread functions are available via the command-line. Additionally, opengroup.org provides a list of all the functions in pthreads and for semaphores.

Getting Started

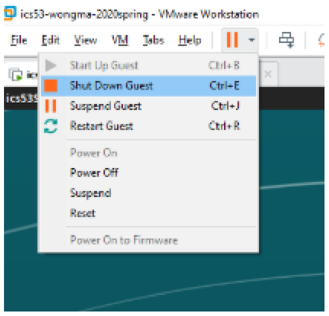
Download the base code for this assignment: `53finalproj.tar` It contains the following file structure.

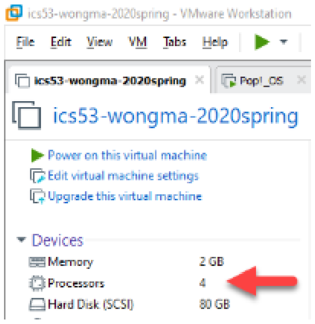
```
53finalproj
├── include              // You may add header files as needed
│   ├── chat.h
│   ├── debug.h
│   ├── protocol.h
│   └── server.h
├── lib                  // Provided libraries
│   ├── chat.o
│   ├── petr_client      // Will be copied to bin/ folder on compilation
│   └── protocol.o
├── Makefile
└── src
    ├── chat             // Modifications only needed for EC
    │   └── chat.c
    └── server           // Implementation should go within this folder
        └── server.c     // Direct from Lab #8 multithreaded version 2
```

[~~5/17/2020~~ 5/18/2020 @10pm - ~~We are still working on the basecode. This information will be added as soon as possible.~~ Basecode is available.]
[5/19/2020 @12:25pm Client code updated to fix issue with REALLY LONG user inputs.]
[5/19/2020 @10:38pm Client code updated to add Quality of Life commands, more informative chat window titles, and a segfault bug.]

A small modification is needed to the provided course VM to allow the VM to run simulating concurrent threads.
To modify the processor cores for your VM:

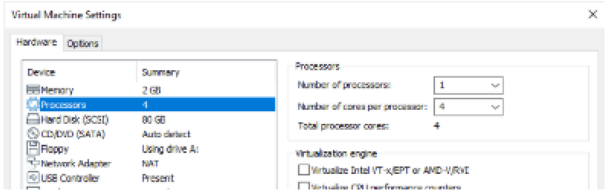
- Shutdown your Guest VM (Ctrl-E).
- Under Devices, double click on the processors.




ics53-wongma-2020spring - VMware Workstation
File Edit View VM Tabs Help
ics53-wongma-2020spring
Pop!_OS
ics53-wongma-2020spring
Power on this virtual machine
Edit virtual machine settings
Upgrade this virtual machine
Devices

Memory	2 GB
Processors	4
Hard Disk (SCSI)	80 GB

- This brings up a pop-up window. Change the number of processors and/or the Number of cores per process to result in a total processor cores value of 4+.



Device	Summary
Memory	2 GB
Processors	4
Hard Disk (SCSI)	80 GB
CD/DVD (SATA)	Auto detect
Floppy	Using drive A:
Network Adapter	NAT
USB Controller	Present

Processors
Number of processors: 1
Number of cores per processor: 4
Total processor cores: 4

Virtualization engine
☐ Virtualize Intel VT-x/EPT or AMD-V/RVI
☐ Virtualize CPU performance counters

Hints and Tips

The following are a list of additional resources which may be useful to you.

- In addition to the course resources, Beej's Guide to Network Programming (http://beej.us/guide/bgnet/) has a good mix of tutorials, explanations, and descriptions of the functions you will be working with for this assignment.
- Information on how to use gdb to debug with threads. Monitor your threads using the htop and named threads.
- Tools such as wireshark (https://www.wireshark.org/) or tcpdump (http://www.tcpdump.org/manpages/tcpdump.1.html) can also be used to monitor the network packets sent between the client and the server.

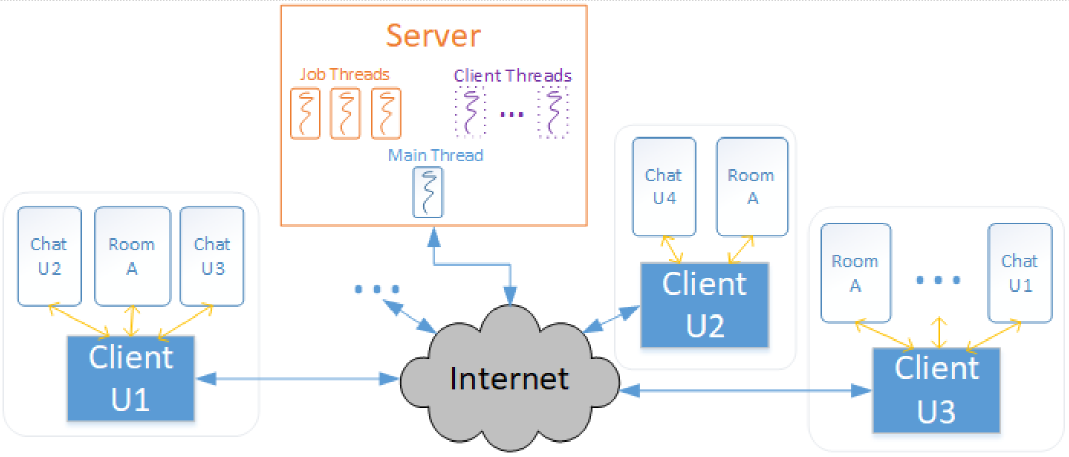
None of these are required to complete the assignment, but may make your debugging easier.

Petr Chat

The Petr Chat server facilitates group chats, called a room, and direct messaging. Each room or direct chat is displayed on the client with an independent chat window, just as Google Hangouts does. As shown in the figure below, multiple clients can be connected to the server at any time. Clients can be "chatting" in a combination of direct chats with a user and in rooms with other users. The main client program will spawn an xterm window for each type of communication.

The server has 3 types of threads:

- A main thread to accept connection from clients.
- A set of short-lived (dotted-line) client threads, one per connected client to read input from the client socket..
- A set of N job threads (consumer) to process the input from the clients. These threads never terminate (solid-line).



Petr Server - Overview

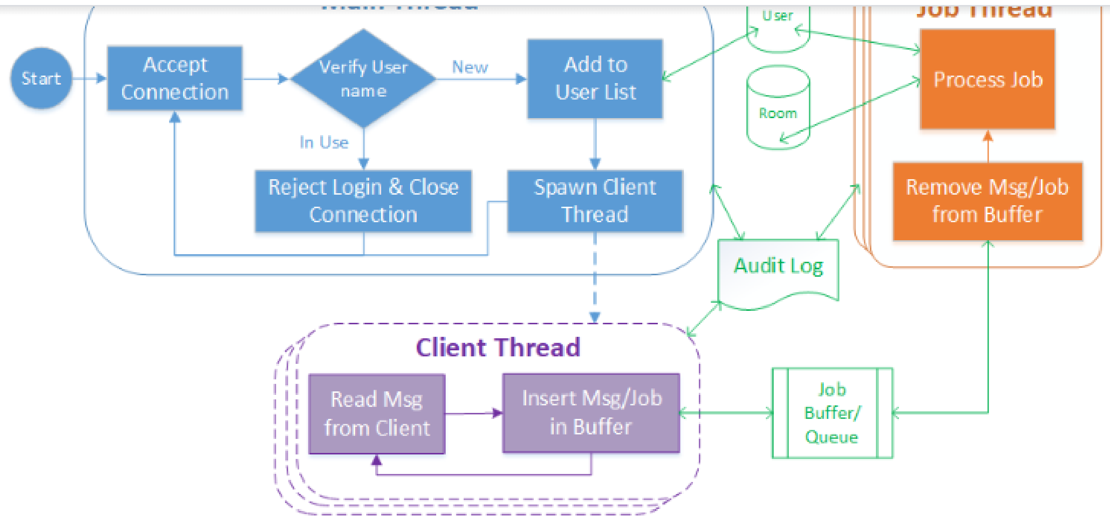
As shown in the previous figure, the server has 3 types of threads running each with a specific role.

- A main thread accepts new connections from clients. The thread communicates with the client initially to log the user into the server. Login checks the client's username to ensure it is unique among the connected clients (i.e. can't have 2 jwongma's at the same time!)
- A set of client threads (producer), one per client. These threads are spawned by the main thread upon successful login. Each client thread is short-lived (dotted-line). It terminates when the client terminates the connection, after performing any required clean up. The thread will receive protocol messages via the socket from the client and create job(s) per protocol message to be handled by the job threads.
- A set of N job threads (consumer). When the server is started, the main thread creates N threads to handle job requests from the client threads (producer). These threads never terminate (solid-line) and will block when there are no jobs to process. As jobs arrive, the job threads will process them in FIFO (first-in-first-out order).

The client threads (producers) are acting as a middle man for communication between the connected clients and the job threads (consumers). We acknowledge their role is minimal and possibly unnecessary. Remember that this project is an EDUCATIONAL exercise. We have designed the server using this architecture for a few reasons:

- Experience with the multiple producer, multiple consumer problem
- Experience with long-lived threads (job threads) and detached event-spawned short-lived threads (client threads)
- Ability to control the contention for locks on the shared data structures (explained later)
- Facilitates an EC option using multiplexing

Below is a diagram representing the general control flow of the server program which you are to implement.



When the server is initially started, N detached job threads (orange) are created and all shared resources (green) are initialized/created. The shared resources are:

- **User** - a shared data structure to maintain information (eg. username, file descriptor, etc) about the current connected users. As this structure is accessed (read) and modified (write) by various threads, consider a reader/writer locking model.
- **Room** - a shared data structure to maintain information (eg. room name, creator, current users in room, etc) about the chat rooms currently on the server. As this structure is accessed (read) and modified (write) by job threads, consider a reader/writer locking model.
- **Job Buffer/Queue** - a shared data structure to maintain all jobs (eg. send a message to the user, send a message to a room, close a room, add a room, etc) requested by the connected clients via the client thread to be performed by a job thread, when available. This structure should follow the producer/consumer access model.
- **Audit Log** - a shared output file to print logs of all events which occur/handled by the main, client and job threads. This file can be used to help debug your programs. Log events with date/time of the occurrence. Threads may require mutual exclusive access to write to this file to ensure correct logging.
  The log should include events such as:
  - User added/denied with name, file descriptor, etc
  - Client thread created/terminated
  - All messages received from a client, sent to a client
  - When a job is inserted/removed from in the job buffer and by which thread

Thread-safe operation is required for these data structures.

You are free to choose the type of data structures for each of these items. Suggestions include the generic linked list (HW#3) and/or sbuf (lecture/textbook). Additionally, we encourage you to define your own structures for each of these

Dynamically sized data structure should not be a MAJOR roadblock to the completion of your project. If you are struggling, you may choose to use fixed size data structures that support at least the following:

- 5 simultaneous chat rooms
- a minimum chat room capacity of 5
- 10 simultaneous connected clients

Be prepared to discuss and explain in detail your challenges, attempted approaches with other data structures, and why your implementation is limited during your project review meeting with the TA.

Additionally, when designing your server you should consider the roles of the client thread and the job thread. One approach is to have the client thread take EVERY message as received from the client and insert it in the job queue for the job thread to decipher and handle all steps. Alternatively, the client thread could take a message from the client and determine a set of jobs to complete the request and insert each of them into the job queue to be handled. Yet another approach is to have the client thread handle the error checking cases and only queue valid messages to be handled by the job threads.

Given the flexibility you have in your approach you are allowed to slightly deviate from the above figure, eg. adding additional connections between components. However, the main core functionality of each thread type, the shared data structures, and the required locking mechanisms must remain.

Petr Protocol

The clients and server will communicate using the **Petr Protocol** described in this section. The concept of a protocol is an important one. When implementing the protocol the requirements and description must be followed **TO THE LETTER**. The idea is to create a standard so that anyone implementing the **Petr Protocol** will be able to connect and operate with any other program implementing the same protocol. Any client and chat should work with any server if the protocol is used correctly.

Each message sent between the client and server will consist of 2 parts: a message header and a message body. The message header consists of 8 bytes: 1 byte to represent the type of the message, 4 bytes to represent the length of the message body (in bytes), and 3 bytes of struct padding. The bytes for the message body directly precedes the header on the socket. This simple structure for the protocol specifies for your program exactly how many bytes to read from the socket for each read and allows for messages to be variable length. This approach is standard in network protocols (although the headers are typically larger and contain more information/fields).

The header structure is defined in `protocol.h` as

```
typedef struct {
    uint32_t msg_len; // includes the null terminator
    uint8_t msg_type;
} petr_header;
```

To assist you with reading the `petr_header` correctly from a socket we have created the helper function:

```
// Read a petr_header from the socket_fd. Places contents into the memory
// referenced by h. Returns 0 upon success, -1 on error.
int rd_msgheader(int socket_fd, petr_header* h);
```

This helper function reads ONLY the message header from the socket. The length of the message body can then be obtained from the structure to determine how many bytes to read from the socket.

One of the main reasons we are providing this function is due to endianness. When the structures are written to a socket, they are written byte-by-byte based on address. When interpreting the data from the socket, the bytes must be reassembled from the socket in the correct ordering before the data is read. While this does not impact char data types such as strings, it does impact larger primitive datatypes such as ints and floats.

To assist you with writing a petr protocol message (header and body) correctly to a socket we have created the helper function:

```
// Write petr_header reference by h and the msgbuf to the socket_fd.
// If the msg_len is 0, the msgbuf is ignored. [5/25 @4:32pm Added for clarity. Piazza @890]
// Returns 0 upon success, -1 on error.
int wr_msg(int socket_fd, petr_header* h, char* msgbuf);
```

This helper function writes the FULL message to the socket. The length of the message body to be sent is obtained from the structure argument.

When sending protocol messages on the socket, the header and the message body must be sent as ONE `write()` call to the socket. A single call is thread-safe, but thread interleaving can occur between subsequent `write()` calls.

It is not a requirement to use these functions within your code. Heed the warnings and information provided

Protocol Descriptions

The following is a description of each of the protocol message types and their meanings with respect to the client and server. A separate printable reference sheet with protocol diagrams of the message interaction is available.

OK (0x00)	Sent by the Server upon successful completion of a Client's request, except in the case of RMLIST & USRLIST.
LOGIN (0x10)	The first message sent by the Client to the Server to request the use of `<username>` on the server. If `<username>` is not already in use, server replies with OK. If it is in use, the server replies with EUSREXISTS.
LOGOUT (0x11)	The last message sent by the Client to the Server to request removal from all chat rooms, closure of all chat rooms created by user, and removal of the user from the user list. The server replies with OK when the action is completed. Closure of a chat room results in the server sending RMCLOSED `<roomname>` to all users in the room.
RMCREATE (0x20)	Sent by the Client to the Server to request the creation of a chat room with `<roomname>`. If `<roomname>` is not already in use, server replies with OK. If it is in use, the server replies with

	ERMEXISTS. The creator of the room is automatically placed in the room on the server. Only the creator of a room can delete the room from the server. This action results in the server sending RMCLOSED to all users in the room.
RMDELETE (0x21)	Sent by the Client to the Server to request deletion of the chat room with `<roomname>`. If `<roomname>` does not exist, the server replies with ERMNOTFOUND. If the user is not the creator of the room, the server replies with ERMDENIED. If the room can be deleted, this action results in the server sending RMCLOSED to all other users in the room and OK sent to the creator.
RMCLOSED (0x22)	Sent by the Server to client's who are in a chat room that is closed by the creator. No response is sent by the Client upon receipt.
RMLIST (0x23)	When sent by the Client, the message is a request for the current list of chat rooms on the server. If there are rooms on the server, the server responds with RMLIST. The message body contains a string of text stating the roomname and the list of users in each room (1 line per room, usernames separated from room name by a colon, usernames separated by commas). If there are no rooms on the server, the server responds with RMLIST with message length 0. [5/27 @5:43pm Clarification: The rooms and users within the room can be in any order. Gradescope will reorder the message body before comparing the output for correctness. Format must be correct! Note: A newline is appended at the end of each userlist, including the last one. The end of the message body should be `"\n\0"`] The server uses this message type in reply to the client's RMLIST request, as stated above.
RMJOIN (0x24)	Sent by the Client to the Server to join an existing chat room on the server. If the chat room with `<roomname>` does not exist, the server responds with ERMNOTFOUND. If the chat rooms has reached the server defined limit (min 5) on the number of users will respond with ERMFULL. Upon successful addition to the room, the server responds with OK.
RMLEAVE (0x25)	Sent by the Client to the Server to leave an existing chat room on the server. If the chat room with `<roomname>` does not exist, the server responds with ERMNOTFOUND. If the chat room with `<roomname>` exists but the user is not in the room, the server responds with OK. (This should never happen using our client, but if it does the server just ignores it) If the chat room with `<roomname>` exists and the user is the creator, the server responds with ERMDENIED. The creator of the room can not leave. The creator must delete the room to leave, RMDELETE. Upon successful removal from the room, the server responds with OK.
RMSEND	Sent by the Client to the Server to send `<message>` to `<chatroom>`.

(0x26)	If the chat room with <roomname> does not exist, the server responds with ERMNOTFOUND. If the chat room with <roomname> exists but the user is not in the room, the server responds with ERMDENIED. (This should never happen using our client) Upon successful send of the message to all users (not the sender) in <roomname>, server responds with OK.
RMRECV (0x27)	Sent by the Server to the Client as a result of the RMSEND received from <from_username>. No response is sent by the Client upon receipt.
USRSEND (0x30)	Sent by the Client to the Server to send <message> to <username>. If <username> does not exist, the server responds with EUSRNOTFOUND. Upon successful send of the message to <to_username>, the server responds with OK.
USRRECV (0x31)	Sent by the Server to the Client as a result of the USRSEND received from <from_username>. No response is sent by the Client upon receipt.
USRLIST (0x32)	When sent by the Client, the message is a request for the current list of users on the server. The message body contains a string of text of the usernames separated by newlines (eg. 1 line per username), not including the requesting user.[5/26 @8:08pm Added Piazza @908] If there are no other users on the server, the server responds with USRLIST with message length 0. [5/27 @5:43pm Clarification: The list of users can be sent in any order. Gradescope will reorder the message body contents before comparing the output for correctness. Format must be correct! Note: A newline is appended at the end of each user, including the last one. The end of the message body should be "\n\0"] The server uses this message type in reply to the client's USRLIST request, as stated above.

The following table summarizes the Petr Protocol message types, their meaning, who sends the message, and the contents/format of the message body (if needed).

The <> are not sent in the body of the message. All message body's are character strings terminated with '\0', which is included in the message length. If a message body format is not specified, the message length sent is 0.

Type	Name	Meaning	Sent By	Message Body Format
0x00	OK	Success	Server	
0x10	LOGIN	User login w/ username	Client	<username>
0x11	LOGOUT	User logout	Client	
0x20	RMCREATE	Create new room	Client	<roomname>
0x21	RMDELETE	Delete room	Client	<roomname>
0x22	RMCLOSED	Room was deleted	Server	<roomname>
0x23	RMLIST	Request list of rooms	Client	
		Return list of rooms w/ users per room	Server	<roomname>:<username>,...,<username>\n…
0x24	RMJOIN	Join room	Client	<roomname>
0x25	RMLEAVE	Leave room	Client	<roomname>
0x26	RMSEND	Send message to room	Client	<roomname>\r\n<message>
0x27	RMRECV	Receive message from a user in room	Server	<roomname>\r\n<from_username>\r\n<message>
0x30	USRSEND	Sent message to user	Client	<to_username>\r\n<message>
0x31	USRRECV	Receive message from user	Server	<from_username>\r\n<message>
0x32	USRLIST	Request user list	Client	
		Return list of users	Server	<username>\n<username>\n<username>\n...

RMSEND, RMRECV, USRSEND, and USRRECV use “\r\n” to separate the fields of the message body. The use of a single set of Windows-style line ends is common in networking protocols to signify the termination of a data field. Newline ‘\n’ alone is not used as this would prevent the ability to have a newline within the message text.

Many network protocols will end their messages with “\r\n\r\n”. As we are sending the length of each message to reduce complexity, this is not required.

As described in the protocol message descriptions, for each message sent by the client, the server will send a response message. If the requested action by the client is successfully completed by the server, the OK message is sent (except for RMLIST, USRLIST, RMRECV, USRRECV). RMLIST and USRLIST message types are used by the server in response to the client's request. The body of the message then includes the requested information.

If the requested action by the client fails the server replies with an error message. All error messages have a message length of 0. The following table specifies the error type and the set of message types which can receive the error.

Type	Name	Meaning	Error in Response to
0x1A	EUSREXISTS	Username already in use	LOGIN
0x2A	ERMEXISTS	Room exists already	RMCREATE
0x2B	ERMFULL	Room capacity reached	RMJOIN
0x2C	ERMNOTFOUND	Room does not exist on server	RMDELETE, RMSEND, RMJOIN, RMLEAVE
0x2D	ERMDENIED	Can't delete room, not creator or Creator can't leave room	RMDELETE, RMLEAVE
0x3A	EUSRNOTFOUND	User does not exist on server	USRSEND
0xFF	ESERV	Generic error	*

* While we have done our best to identify all possible errors which can occur, as humans we know we might have missed one (or more). Since the client is provided to you, it is difficult to add support for any new errors which could arise. Therefore, we have added ESERV as a catch for any of these unforeseen cases.

Petr Server - Implementation

The following is the usage statement for your server program.

```
./bin/petr_server [-h][-j N] PORT_NUMBER AUDIT_FILENAME

-h               Displays this help menu, and returns EXIT_SUCCESS.
-j N             Number of job threads. Default to 2.
AUDIT_FILENAME   File to output Audit Log messages to.
PORT_NUMBER      Port number to listen on.
```

Remember that options in **[ ]** are considered optional, and arguments which are not assigned a flag are positional.

To implement this, consider using getopts() as illustrated in the HW#2 base code.

When the server is first started it should print out which port it is currently listening on.

```
$ ./bin/petr_server 3200 audit.txt
Currently listening on port 3200.
```

Create a signal handler for the server so that if `ctrl-c` is pressed, the socket is closed correctly. Failure to close the socket correctly, you may have to wait for the OS to clean it up which sometimes happens instantly, and other times you have to wait up to 5 minutes. This can be annoying during development and grading.

The multi-threaded server code from Lab #8 is provided in `src/server.c` as a starting point for your implementation. Feel free to start fresh and/or to include any of the provided code from the labs/homeworks/book this term. You may also use any helper functions which either partner wrote during the term and any standard C libraries.

The provided Makefile will compile all *.c files within the src/server directory to create your server implementation.

Petr Client - Usage

A working client is provided with the base code. Only a subset of the code is accessible for viewing/modification (for EC). The base project requires no modifications to the client code. The Makefile will build the client executable.

The usage statement of the program is:

```
./bin/petr_client [-h] NAME SERVER_ADDR SERVER_PORT

-h                 Displays this help menu, and returns EXIT_SUCCESS.
NAME               This is the username to display when chatting.
SERVER_ADDR        The IP address of the server to connect to.
SERVER_PORT        The port to connect to.
```

When the client is run, a connection with the server will be initiated based on the command line arguments. The client will request `LOGIN` with the server. If LOGIN fails, the client will terminate. If the LOGIN is successful, a message will print and the terminal will present a prompt (>) to take commands from the user.

```
$ ./bin/petr_client "petr anteater" 127.0.0.1 8080
INFO: petr anteater
INFO: 127.0.0.1
INFO: 8080
Socket successfully created
Try connecting to server @ 127.0.0.1 on port 8080
Successfully connected to the server
>
```

Below is a list of the available commands and a description of their operation.

Command(s)	**Description**
`/help` `/h`	Lists available commands to run

`/createroom roomname`
`/create roomname`
`/c roomname`
	Sends a request to create a room on the server. `roomname` can include spaces.
	
	An xterm window will appear if the room is successfully created on the server.
	
	If an xterm window for a room with `roomname` is already open, no action is taken.
	
	Petr Protocol command:`RMCREATE`

`/deleteroom roomname`
`/delete roomname`
`/d roomname`
	Sends a request to delete a room on the server. `roomname` can include spaces.
	
	If successful and the associated xterm window is open, it will be closed.

`/joinroom roomname`
`/join roomname`
`/j roomname`
	Sends a request to join an existing room on the server. `roomname` can include spaces.
	
	An xterm window will appear upon successfully joining the room.
	
	If an xterm window for a room with `roomname` is already open, no action is taken.

`/leaveroom roomname`
`/leave roomname`
`/l roomname`
	Sends a request to leave a room on the server. `roomname` can include spaces.
	
	If successful and the associated xterm window is open, it will be closed.

`/roomlist`
`/rlist`
`/r`
	Sends a request to the server for a list of all current rooms and their associated users. Results are displayed on the terminal.

`/userlist`
`/ulist`
`/u`
	Sends a request to the server for a list of all current users. Results are displayed on the terminal.

`/chat username`
`/ch username`
	Sends a request to directly message `username` on the server. `username` can include spaces.
	
	An xterm window will appear upon successfully sending the message to the user.
	
	If an xterm window for the user is already open, no action is taken.

`/logout`
`/quit`
`/q`
	Sends a request to the server to logout the user. Closes all currently open xterm windows for the client.

When an xterm window opens for a chat room or direct message to another user, the user can type within the xterm window to send messages. When incoming messages arrive, they will appear on the appropriate xterm window. If an error occurs a message will print on the terminal.

When an xterm window is terminated (X in corner), the window will close. No command is sent to the server to correspond to this action. The next time a user in the room sends a message, a new xterm window will appear.

Client commands can be entered into the terminal at any time.

If you start to type in the xterm window or the terminal and before pressing enter, incoming data is printed to `stdout`, the cursor will move and the output will be interleaved on the terminal. This visual interleaving has NO IMPACT on the input typed on `stdin`.

`Ctrl-C` is not handled on the client. By terminating with `SIGINT` via the client, you are able to test how your server will handle a client crash/disconnection.

Make sure to clean up your data state on crash/disconnection!

We have done our best to catch as many possible errors from the server and print out warning messages to help you determine the issue. However, there are cases which may not be covered. It is possible that your server implementation will send data to the client that causes a segfault.

Extra Credit (up to 15 points towards homework)

A separate submission will not be used for these extra credit options. Add them directly into the main implementation. These items will be evaluated during your grading meeting and/or by the teaching staff via a Canvas quiz.

- (10 pts) Instead of spawning a client thread per connected client, utilize I/O multiplexing via select or epoll
  - There are multiple approaches to this each with different design considerations. For example, you can multiplex on both the accept socket and client sockets in the main thread. Alternatively, a single client thread can multiplex on all clients, which requires IPC to communicate new clients from main thread to client thread.
- (up to 5pts) Customize your xterm client chat windows. Impress us, nothing trivial.
  - Code associated with the way the messages are displayed on the xterm is in `src/chat/chat.c`
  - The formatting/colors of the xterm can be modified Refer to: https://debian-administration.org/article/66/Customizing_your_xterm