CSCI 4220 Assignment 2

Concurrent Chatroom Server (select() + Worker Threads)

In this assignment, you will implement a **multi-client chatroom server** that combines **I/O multiplexing (`select()`)** with **multithreading (`pthread`)** to support concurrent messaging among multiple users.
This design mirrors real-world chat backends such as **IRC**, **Slack**, or **Discord**, which rely on asynchronous I/O and worker threads to handle simultaneous message traffic efficiently.
For instance, Discord's backend services use async frameworks (e.g., Node.js, Go, Rust tokio) that conceptually function like `select()` combined with threads, while Slack uses asynchronous I/O (`epoll`/`kqueue`) plus worker pools for message routing. The file skeleton.c is provided as a code skeleton to help you start your implementation.

Learning Goals

By completing this assignment, you will:

- Implement a TCP server that supports multiple simultaneous clients.
- Use `select()` for non-blocking, event-driven I/O.
- Employ threads and synchronization primitives (`pthread_mutex_t`, `pthread_cond_t`) for concurrency.
- Design and manage thread-safe producer/consumer queues.
- Handle client disconnects and server shutdown gracefully.

Program Invocation

```
./chatroom_server.out [port] [num_workers] [max_clients]
```

- `port` — TCP port number for the server to listen on.
- `num_workers` — number of worker threads in the processing pool (e.g., 2–4).
- `max_clients` — maximum number of simultaneous clients (e.g., <= 10).

Example:

```
./chatroom_server.out 12000 3 5
```

Protocol & Behavior

1. Connection and Username

When a client connects, the server sends:

```
Welcome to Chatroom! Please enter your username:
```

The client responds with a single-line username (letters, digits, or underscores only).
Usernames are **case-insensitive** and must be **unique** among connected users.

If accepted:

```
Let's start chatting, alice!
```

If rejected:

```
Username "alice" is already in use. Try another:
```

When users join or leave, the server notifies all connected clients:

```
bob joined the chat.
...
bob has left the chat.
```

---

2. Regular Messages and Processing Pipeline

After login, any non-empty line (<= 1024 bytes) is treated as a chat message.

Message Flow

1. **Main thread:**
   - Uses `select()` to receive messages from clients.
   - Packages each message as a **Job** and enqueues it to the **job queue**:
     `struct Job { int sender_fd; char username[32]; char msg[1024]; };`
2. **Worker threads:**
   - Dequeue jobs from the `job_queue`.
   - Sanitize and format them (remove trailing newlines, handle special commands).
   - Push processed messages into a **broadcast queue**.

   Detailed steps:
   1. **Dequeue job:**
      Wait (via condition variable) until a new message appears in the queue.
   2. **Format message:**
      - Regular message → `username: message`
      - `/me <text>` → `*username text*`
   3. **Enqueue for broadcast:**
      Push formatted messages into the broadcast queue.
      Because multiple worker threads process jobs concurrently, the final order of broadcast messages may differ slightly from the original arrival order. This behavior is **normal** for concurrent systems and will not affect grading.
3. **Main thread:**
   - Continuously checks the broadcast queue in its main loop and sends all pending messages to connected clients (except the sender).

---

3. Commands

Commands start with `/` and are case-insensitive:

Command	Description
`/who`	Lists all currently connected users (reply sent only to requester).
`/me <text>`	Broadcasts an action message formatted as `*username text*`.
`/quit`	Disconnects the client gracefully.

Example:
If **alice** types `/me waves at bob`, all users receive:

```
*alice waves at bob*
```

Error Handling Invalid commands should result in a private error message:

```
Invalid command. Type /who, /me, or /quit.
```

---

4. Message Input and Robustness

- Each message is a **single line ending with a newline** (`'\n'`).
- Maximum line length: 1024 bytes (truncate if necessary).
- Empty lines are ignored.
- Because TCP delivers a continuous stream of bytes rather than discrete messages, your server must handle cases where a `recv()` call may return only part of a line or multiple lines at once.
  - Your implementation should therefore:
    1. Maintain a per-client input buffer (e.g., `inbuf`) to accumulate bytes received from each client.
    2. Detect complete messages terminated by a newline character (`'\n'`).
    3. Process each complete line as a chat message.
    4. Leave any incomplete fragment in the buffer to be completed by future `recv()` calls.
- If a client disconnects or a `send()`/`recv()` call fails, remove that client without crashing the server.
- Username matching is **case-insensitive** (`Bob`, `boB`, and `BOB` refer to the same user).

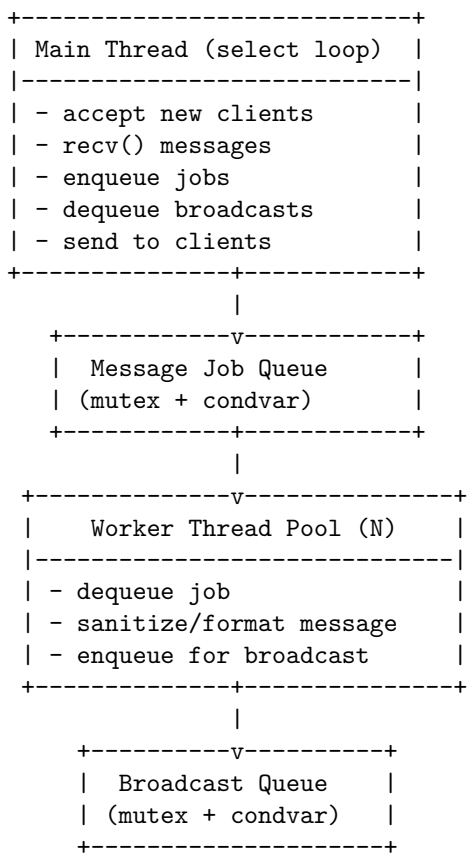
Concurrency Requirements

- **Main thread (I/O loop):**
  Uses `select()` to handle all client sockets and new connections.
- **Worker pool:**
  Multiple threads wait on a condition variable.
  When new jobs arrive, workers wake up, process messages, and enqueue formatted results.
- **Queues:**
  Two thread-safe queues are required:
  - `job_queue`: messages awaiting processing (main thread → workers).
  - `bcast_queue`: processed messages ready for broadcast (workers → main thread).
    Each queue must be protected by a mutex and condition variable to ensure correct synchronization.
- **Graceful shutdown:**
  On `SIGINT` (Ctrl+C):
  - Stop accepting new clients.
  - Close all sockets.
  - Mark queues as closed.
  - Signal all threads to exit and call `pthread_join()`.

  Hint:
  - The main thread signals worker threads using `pthread_cond_signal()` whenever a new job is added.

- Worker threads block on `pthread_cond_wait()` until work becomes available.
- On shutdown, use `pthread_cond_broadcast()` to wake all waiting threads so they can exit cleanly.

Suggested Architecture



Build and Run

Provide a `Makefile` that builds your server with `clang` and `-pthread`.

Example Makefile:

```
chatroom_server.out: chatroom_server.c
    clang -Wall -Wextra -O2 -pthread chatroom_server.c -o chatroom_server.out
```

Testing: Use `netcat` (or multiple terminals) to simulate clients:

```
nc localhost 12000
```

Example Session (2 clients):

```
Welcome to Chatroom! Please enter your username:
bob
Let's start chatting, bob!
alice joined the chat.
bob: hello everyone
```

```
*alice waves*
bob has left the chat.
```

Submission

Upload to Submitty:

```
chatroom_server.c
Makefile
README.txt
```

Academic Integrity and AI Use Policy

You must implement all networking, threading, and synchronization logic yourself.
All submissions will be automatically checked for **code similarity** and **AI-generated content**.
Using code from another student or online source is considered plagiarism.

If you use AI tools (such as ChatGPT or Copilot) to help debug or explain concepts, you must **document this clearly in your README**. Only submit code that reflects your **own understanding and work**.