Lab 4: Handling Client Data with `select()`

Objective:

Complete the provided `select()`-based TCP server to handle incoming data from multiple connected clients. The server should read data from any client that has sent something, print it, and echo it back to that client. It should also handle client disconnections gracefully.

Background:

You have a server that can accept multiple connections using `select()` and non-blocking sockets. The `select()` call tells you when the listening socket has a new connection OR when an existing client socket has data ready to be read. Your task is to implement the logic for the latter.

Starting Code:

Use the `server_select.cpp` code provided/developed in the lecture (the version you posted in the prompt).

Tasks:

Locate the TODO:

Find the comment at the end of the `while(true)` loop:

```
// Now, cycle through the read_fds and see where you can read
data.
```

This is where you will add your code.

Iterate Through Potential Client Sockets:

- After handling new connections (the `if (FD_ISSET(server_fd, &read_fds))` block), you need to check all other possible file descriptors that might have been set in `read_fds` by `select()`.

- You can loop from `0` (or `server_fd + 1`) up to `max_fd`.
- Inside the loop, for each file descriptor `i`:
  - Check if it's the listening socket (`server_fd`). If so, skip it (it was handled already).
  - Check if `FD_ISSET(i, &read_fds)`. If this is true, it means client socket `i` has data to read or has an event (like disconnection).

Handle Readable Client Socket `i`:

If `FD_ISSET(i, &read_fds)` is true for a client socket `i`:

Receive Data:

- Declare a `char buffer[BUFFER_SIZE]` and `memset` it to zero.
- Call `ssize_t bytes_read = recv(i, buffer, BUFFER_SIZE - 1, 0);`.

Process `recv()` Return Value:

- **`bytes_read > 0` (Data Received):**
  - Null-terminate the buffer: `buffer[bytes_read] = '\0';`.
  - Print what was received (e.g., "Received from socket X: YYY").
  - **Echo Data Back:** Call `send(i, buffer, bytes_read, 0);`.
    - **Error Check `send()`:** If `send()` returns `-1`:
      - If `errno` is `EAGAIN` or `EWOULDBLOCK`, it means the send buffer is full. For this lab, you can just print a message like "Send on socket X would block, data not sent this time." (A more advanced server would add this socket to a `write_fds` set for `select()` to know when it's writable again).
      - For other errors (`perror("send failed")`), assume the client connection is problematic. Close the socket `i`, and remove it from `master_fds` using `FD_CLR(i, &master_fds)`.
- **`bytes_read == 0` (Client Disconnected Gracefully):**
  - Print a message like "Client on socket X disconnected."
  - `close(i);`
  - Remove the socket from the `master_fds` set: `FD_CLR(i,`

`&master_fds);`.
- **`bytes_read < 0` (Error on `recv`):**
  - Check `errno`. If it's `EAGAIN` or `EWOULDBLOCK`, it means no data was actually ready (this can happen even if `select` indicated readiness, though it's less common for TCP data unless there was a race or spurious wakeup). You can usually just ignore this and `select` will notify you again.
  - For other errors (`perror("recv failed")`), assume the client connection is problematic.
    - Print an error message.
    - `close(i);`
    - Remove the socket from `master_fds`: `FD_CLR(i, &master_fds);`.
- **Updating `max_fd` (Important if you remove FDs):**
  - While not strictly necessary for this specific loop structure if you always iterate up to the current `max_fd`, if you were to remove a client socket that *was* `max_fd`, you would ideally re-calculate `max_fd` by finding the new highest FD in `master_fds`. For this lab, simply clearing it from `master_fds` is the primary goal. A simple server might not shrink `max_fd` and just continue iterating up to the old `max_fd`, which is slightly inefficient but often works.

Testing Your Completed Server:

1. Compile your `server_select.cpp`.
2. Run `./server_select`.
3. Open multiple `netcat` (or your Day 3 client) instances in separate terminals:

   ```
   netcat localhost 8080
   ```
4. From each `netcat` client:
   - Type messages and press Enter. Your server should print what it received and echo it back to the correct `netcat` window.
   - Try sending messages from different clients in an interleaved fashion.
   - Close one of the `netcat` clients (e.g., with `Ctrl+D` or `Ctrl+C`). Your server should detect the disconnection and print a message. The other clients should remain connected and functional.

5. Observe the server's console output for connection messages, received data, and disconnection messages.

Bonus Challenges (Optional):

1. **Manage `max_fd` more accurately:** When a client whose `fd` was `max_fd` disconnects, recalculate `max_fd` by iterating through `master_fds` to find the new highest active `fd`.
2. **Handle Partial Sends:** If `send()` returns a value less than `bytes_read` (but greater than 0), implement a loop to send the remaining data. This would involve adding the socket to a `write_fds` set for `select()` if the send would block. (This is more advanced).