PROJECT PHASE 3

Ayyub Shaffy (am12321), Matija Susic (ms14012)

1. Architecture and Design

High-Level Structure

Our program has been upgraded from an iterative, single-client server (Phase 2) to a concurrent, multi-client server using POSIX threads. The core architecture now handles numerous clients simultaneously, with each client's session managed by a dedicated thread.

The components of the system are:

- **`server.c`:** This file has been significantly refactored. The main function is now a simple loop that `accept()`s new connections and spawns a new thread for each one. All per-client session logic (command receiving, execution, and response) has been moved into a thread function (`client_thread_func`).

- **`client.c`:** Unchanged from Phase 2. The client is unaware of the server's new architecture.

- **`net.c` and `net.h`:** Unchanged from Phase 2. The robust `readn`/`writen` and framing protocol functions are re-used. These functions are thread-safe as they only operate on the file descriptor passed to them.

- **`utils.c` and `utils.h`:** Unchanged from Phase 2. The core parsing (`parse_command`), pipeline building (`build_pipeline`), and execution (`exec_pipeline`) logic is re-used as-is by each thread.

- **`makefile`:** Updated to link the server against the `pthread` library.

Key Design Decisions

1. **Concurrent, One-Thread-Per-Client Model**
    - The Phase 2 architecture, where the main loop handled a single client, has been replaced. The `main` thread's *only* responsibility is to `accept()` new connections.

    - **Reason:** This is the core requirement of Phase 3. It allows the server to remain responsive and accept new clients while other, long-running commands (like `sleep` or complex pipelines) are executed by existing clients.

    - Upon a new connection, `pthread_create()` is called, passing the client's information to a new `client_thread_func`.

2. **Thread Detachment**
    - The first action in `client_thread_func` is to call `pthread_detach(pthread_self())`.

    - **Reason:** This makes the thread "detached," meaning its resources are automatically reclaimed by the system when it exits. This is a simple and effective resource management strategy that avoids the need for the main thread to `pthread_join()` every client, which would be complex and block the main `accept` loop.

3. **Thread Data Passing**
    - A new `client_info_t` struct was created to pass all necessary information to a new thread.

    - **Reason:** `pthread_create` only accepts a single `void*` argument. This struct bundles the client's file descriptor (`cfd`), unique ID (`client_id`), and their address information (`sockaddr_in`) into one `malloc`'d block that can be safely passed.

4. **Per-Thread Logging Prefix**
    - A new helper, `log_line_prefixed`, was created to handle the new logging format. Each thread first builds a unique prefix string (e.g., `[Client #1 - 127.0.0.1:56789]`).

    - **Reason:** This satisfies the project's logging requirement and makes the server's concurrent output readable. By building the prefix once, we avoid repeated `snprintf` calls for every log message within the loop.

Data Structures

`client_info_t`: The only new data structure in this phase. It holds all per-client state needed by a new thread.

```
typedef struct {
    int cfd;
    uint32_t client_id;
    struct sockaddr_in peer;
} client_info_t;
```

`g_client_counter`: A simple global `int` used to assign unique, incrementing IDs to clients. It is only ever accessed and incremented by the main `accept` thread, so it does not require a mutex.

2. Implementation Highlights

Core Flow

The server's new core flow is entirely event-driven and non-blocking from the main thread's perspective.

1. Server's `main()` calls `tcp_listen()` and enters its `for(;;)` accept loop.
2. A client connects. `accept()` returns a new `cfd`.
3. The main thread `malloc`s a `client_info_t` struct and populates it with the `cfd`, `peer` info, and a new `client_id` from `g_client_counter`.
4. `pthread_create()` is called, passing a pointer to this struct to the `client_thread_func`.
5. The main thread *immediately* loops back to step 2, ready to accept another client.
6. Simultaneously, the new `client_thread_func` begins execution.

Client Thread Function (`client_thread_func`)

This function contains the entire session logic that was previously in `main` in Phase 2.

1. **Detach:** Calls `pthread_detach(pthread_self())` for automatic resource cleanup.
2. **Unpack Info:** Casts the `void* arg` back to `client_info_t*`.
3. **Create Log Prefix:** Creates the `client_prefix` string (e.g., `[Client #1 - 127.0.0.1:56789]`) using `snprintf`.
4. **Log Connection:** Prints the required `[INFO] Client #... connected...` log.
5. **Enter Session Loop:** Enters the *exact same* `for(;;)` loop from Phase 2:
   - `recv_frame()`: Wait for a command.
   - Handle `exit`: Send the `0xFFFFFFFF` control frame and `break`.
   - **Log & Execute:** Use `log_line_prefixed` to log `[RECEIVED]` and `[EXECUTING]`.
   - `run_command_capture()`: Call the *same* Phase 1/2 execution logic. This function is re-entrant and safe to be called by multiple threads.
   - **Log & Respond:** Use `log_line_prefixed` to log `[OUTPUT]` or `[ERROR]` based on the result, and `send_frame()` the payload back to the client.
6. **Cleanup:** After the loop (on `exit` or error), the thread `close(cfd)`, logs the disconnect, `free(info)`, and returns `NULL`, terminating the thread.

New Logging Implementation

The new logging format required two small additions.

1. A new function to handle the prefix:

```
static void log_line_prefixed(const char *tag, const char *prefix,
const char *fmt, ...) {
    va_list ap; va_start(ap, fmt);
    fprintf(stderr, "[%s] %s ", tag, prefix); // [TAG] [Client #1 -
...]
    vfprintf(stderr, fmt, ap);
    fputc('\n', stderr);
    va_end(ap);
}
```

2. The prefix creation inside the thread function:

```
// Inside client_thread_func:
char ip[INET_ADDRSTRLEN];
inet_ntop(AF_INET, &info->peer.sin_addr, ip, sizeof ip);
uint16_t port = ntohs(info->peer.sin_port);

char client_prefix[128];
snprintf(client_prefix, sizeof client_prefix, "[Client #%d - %s:%u]",
client_id, ip, port);

// Example usage:
log_line_prefixed("RECEIVED", client_prefix, "Received command:
\"%s\"", cmd);
```

3. Execution Instructions

The `makefile` has been updated to link the `pthread` library.

1. Compile the server and client:

```
make
```

2. Run the server: Open a terminal and run the server.

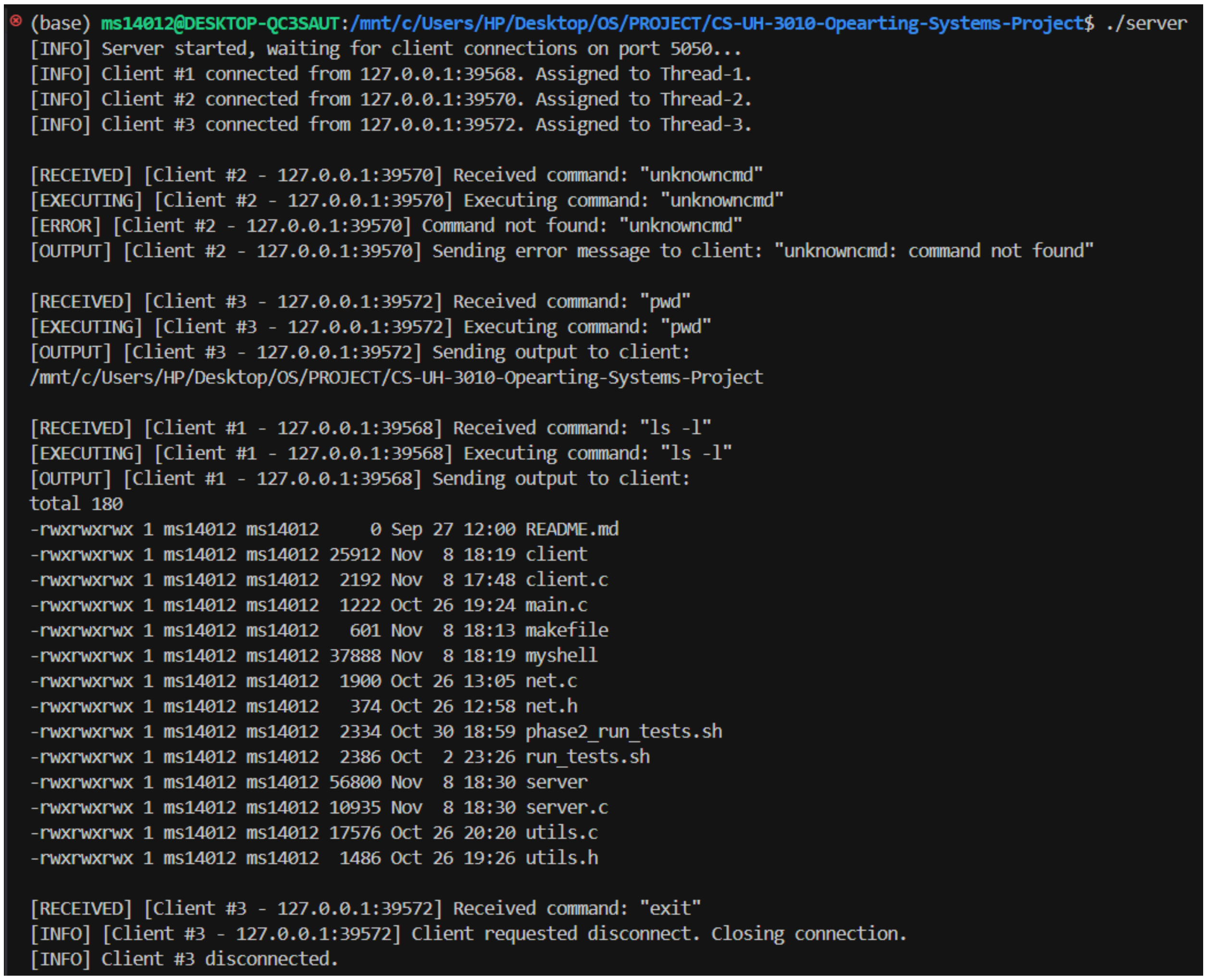
```
./server
```

You should see: `[INFO] Server started, waiting for client connections on port 5050...`

3. Run clients: Open multiple *new* terminal windows. In each one, you can connect to the server.

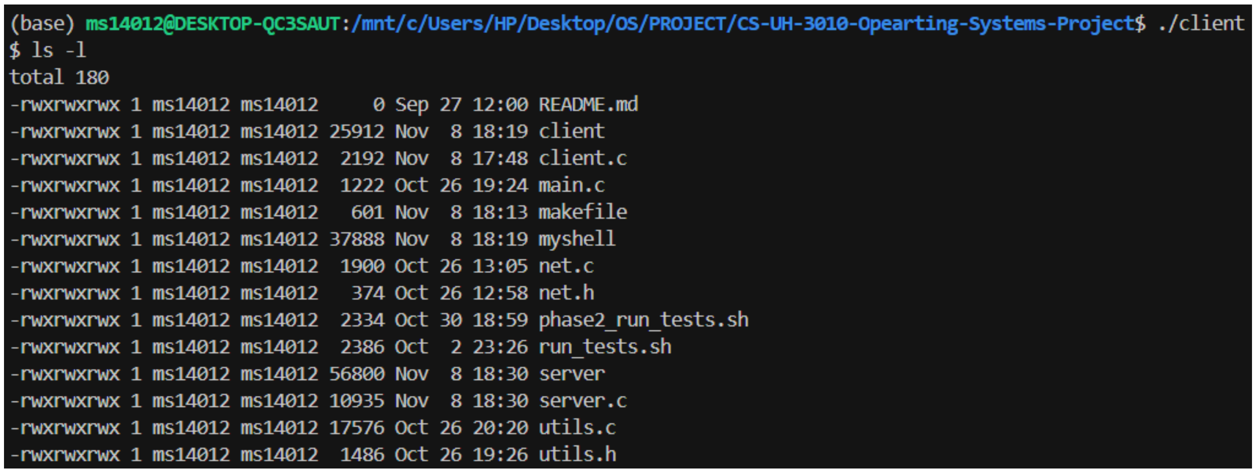
```
# In Terminal 2
./client

# In Terminal 3
./client
```

4. Testing



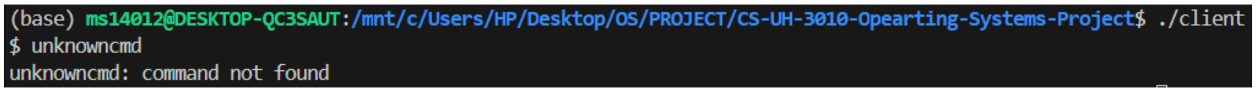
```
(base) ms14012@DESKTOP-QC3SAUT:/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project$ ./server
[INFO] Server started, waiting for client connections on port 5050...
[INFO] Client #1 connected from 127.0.0.1:39568. Assigned to Thread-1.
[INFO] Client #2 connected from 127.0.0.1:39570. Assigned to Thread-2.
[INFO] Client #3 connected from 127.0.0.1:39572. Assigned to Thread-3.

[RECEIVED] [Client #2 - 127.0.0.1:39570] Received command: "unknowncmd"
[EXECUTING] [Client #2 - 127.0.0.1:39570] Executing command: "unknowncmd"
[ERROR] [Client #2 - 127.0.0.1:39570] Command not found: "unknowncmd"
[OUTPUT] [Client #2 - 127.0.0.1:39570] Sending error message to client: "unknowncmd: command not found"

[RECEIVED] [Client #3 - 127.0.0.1:39572] Received command: "pwd"
[EXECUTING] [Client #3 - 127.0.0.1:39572] Executing command: "pwd"
[OUTPUT] [Client #3 - 127.0.0.1:39572] Sending output to client:
/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project

[RECEIVED] [Client #1 - 127.0.0.1:39568] Received command: "ls -l"
[EXECUTING] [Client #1 - 127.0.0.1:39568] Executing command: "ls -l"
[OUTPUT] [Client #1 - 127.0.0.1:39568] Sending output to client:
total 180
-rwxrwxrwx 1 ms14012 ms14012     0 Sep 27 12:00 README.md
-rwxrwxrwx 1 ms14012 ms14012 25912 Nov  8 18:19 client
-rwxrwxrwx 1 ms14012 ms14012  2192 Nov  8 17:48 client.c
-rwxrwxrwx 1 ms14012 ms14012  1222 Oct 26 19:24 main.c
-rwxrwxrwx 1 ms14012 ms14012   601 Nov  8 18:13 makefile
-rwxrwxrwx 1 ms14012 ms14012 37888 Nov  8 18:19 myshell
-rwxrwxrwx 1 ms14012 ms14012  1900 Oct 26 13:05 net.c
-rwxrwxrwx 1 ms14012 ms14012   374 Oct 26 12:58 net.h
-rwxrwxrwx 1 ms14012 ms14012  2334 Oct 30 18:59 phase2_run_tests.sh
-rwxrwxrwx 1 ms14012 ms14012  2386 Oct  2 23:26 run_tests.sh
-rwxrwxrwx 1 ms14012 ms14012 56800 Nov  8 18:30 server
-rwxrwxrwx 1 ms14012 ms14012 10935 Nov  8 18:30 server.c
-rwxrwxrwx 1 ms14012 ms14012 17576 Oct 26 20:20 utils.c
-rwxrwxrwx 1 ms14012 ms14012  1486 Oct 26 19:26 utils.h

[RECEIVED] [Client #3 - 127.0.0.1:39572] Received command: "exit"
[INFO] [Client #3 - 127.0.0.1:39572] Client requested disconnect. Closing connection.
[INFO] Client #3 disconnected.
```

Figure 1: Server output



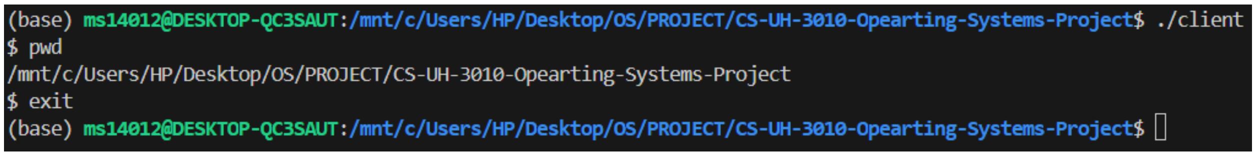
```
(base) ms14012@DESKTOP-QC3SAUT:/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project$ ./client
$ ls -l
total 180
-rwxrwxrwx 1 ms14012 ms14012     0 Sep 27 12:00 README.md
-rwxrwxrwx 1 ms14012 ms14012 25912 Nov  8 18:19 client
-rwxrwxrwx 1 ms14012 ms14012  2192 Nov  8 17:48 client.c
-rwxrwxrwx 1 ms14012 ms14012  1222 Oct 26 19:24 main.c
-rwxrwxrwx 1 ms14012 ms14012   601 Nov  8 18:13 makefile
-rwxrwxrwx 1 ms14012 ms14012 37888 Nov  8 18:19 myshell
-rwxrwxrwx 1 ms14012 ms14012  1900 Oct 26 13:05 net.c
-rwxrwxrwx 1 ms14012 ms14012   374 Oct 26 12:58 net.h
-rwxrwxrwx 1 ms14012 ms14012  2334 Oct 30 18:59 phase2_run_tests.sh
-rwxrwxrwx 1 ms14012 ms14012  2386 Oct  2 23:26 run_tests.sh
-rwxrwxrwx 1 ms14012 ms14012 56800 Nov  8 18:30 server
-rwxrwxrwx 1 ms14012 ms14012 10935 Nov  8 18:30 server.c
-rwxrwxrwx 1 ms14012 ms14012 17576 Oct 26 20:20 utils.c
-rwxrwxrwx 1 ms14012 ms14012  1486 Oct 26 19:26 utils.h
```

Figure 2: Client 1 output



```
(base) ms14012@DESKTOP-QC3SAUT:/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project$ ./client
$ unknowncmd
unknowncmd: command not found
```

Figure 3: Client 2 output



```
(base) ms14012@DESKTOP-QC3SAUT:/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project$ ./client
$ pwd
/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project
$ exit
(base) ms14012@DESKTOP-QC3SAUT:/mnt/c/Users/HP/Desktop/OS/PROJECT/CS-UH-3010-Opearting-Systems-Project$
```

Figure 4: Client 3 output

5. Challenges

The biggest challenge was refactoring the Phase-2 single-threaded server loop into a thread entry point without rewriting Phase-1 logic. We lifted the per-client session code into `client_thread_func`, moved connection metadata into a small `client_info_t`, and detached threads to avoid joins. Care had to be taken to pass only the client-specific state (fd, id, peer addr), avoid sharing mutable state across threads, and keep all shell execution in the child process exactly as before.

A second challenge was getting logging right. Every line now needs clear attribution (client ID plus ip:port) so we added a per-connection prefix and used a prefixed logger for all events (`RECEIVED`, `EXECUTING`, `OUTPUT`, `ERROR`). We also had to normalize presentation details: insert exactly one blank line before each command block, trim trailing newlines in previews to prevent double spacing, and distinguish clean EOF (Ctrl+C) from I/O/protocol errors so disconnects log as `INFO`, not `ERROR`.

6. Division of Tasks

Ayyub: Focused on the **server-side architecture**. This involved refactoring `server.c` to handle concurrency, implementing the main `accept()` loop, and moving all session logic into the new `client_thread_func`.

Matija: Focused on **stability and validation**. Did testing. Implemented feedback from phase 2. Wrote the lab report. He also managed the thread data (`client_info_t`) and implemented the new prefixed logging format.

7. References

- Linux man-pages (Linux Programmer's Manual): Used for `pthread_create`, `pthread_detach`, and `pthread_self`.