Operating Systems

Project Report - Phase 3

Date: April 17, 2025

Instructor: Dena Ahmed

Group Members:

Haad Mehboob (hm2957)
Ameen Vadakkekara (av2851)

Contents

1 Architecture and Design

1.1 System Architecture Overview

Building upon our Phase 2 implementation, Phase 3 enhances the server functionality to support multiple clients simultaneously using multithreading. The architecture now includes a thread management layer that enables concurrent command execution for multiple connected clients.

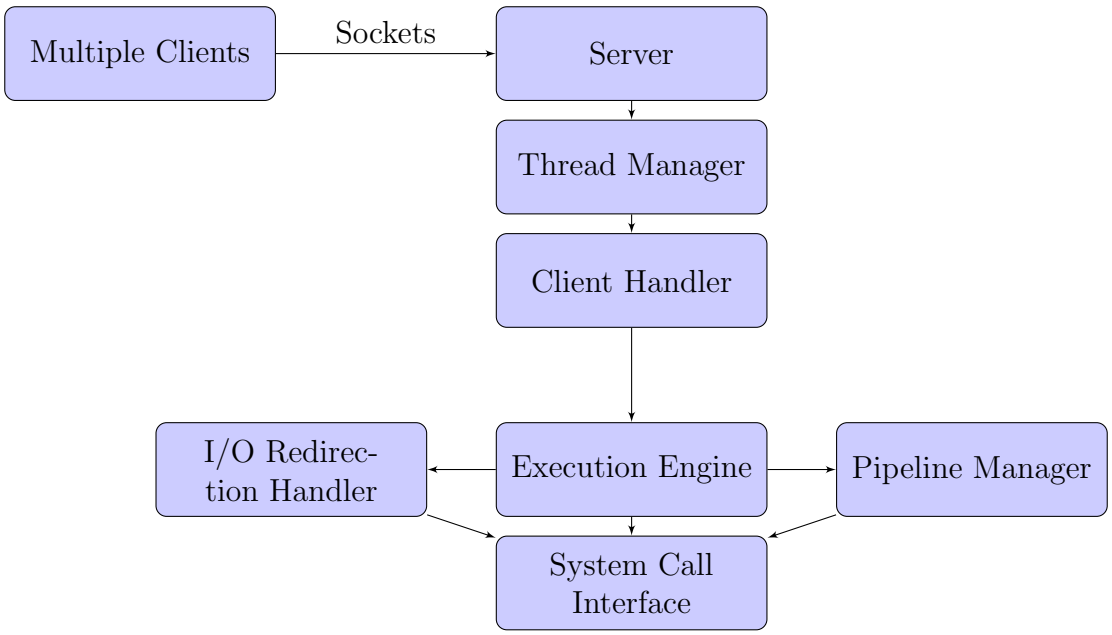

Figure 1: MyShell Phase 3 Architectural Overview with Multithreaded Server and Single Client Handler

1.2 Design Principles and Rationale

Our Phase 3 implementation builds on the design principles established in Phase 2, with additional considerations for concurrent client handling:

1. **Concurrency**: The server now employs multithreading to handle multiple clients simultaneously, allowing each client to have its own dedicated thread for command processing.

2. **Thread Safety**: Critical sections of code are protected to ensure thread safety when accessing shared resources, preventing race conditions and data corruption.

3. **Client Identification**: Each client connection is assigned a unique identifier to track and manage client interactions in the server logs.

4. **Resource Management**: Careful management of thread resources to prevent leaks, including proper thread creation, detachment, and cleanup.

5. **Structured Logging**: Enhanced logging that clearly identifies which client is sending commands and receiving responses, following the required format specified in the project requirements.

6. **Scalability**: The multithreaded design ensures the server can handle an increasing number of clients efficiently without significant performance degradation.

1.3 Core Components and Their Interactions

1.3.1 Thread Manager

The thread manager is a new component responsible for creating and managing threads for each client connection:

Key responsibilities:

- Creating a new thread for each incoming client connection
- Assigning a unique client ID to each connection
- Managing thread resources and ensuring proper cleanup
- Coordinating thread execution and termination

1.3.2 Client Handler

Each client connection is managed by a dedicated client handler thread:

Key responsibilities:

- Maintaining the socket connection with a specific client
- Receiving commands from the assigned client
- Processing commands using the shell infrastructure
- Capturing command output and sending it back to the client
- Logging client interactions with proper identification
- Handling client disconnection and thread termination

1.3.3 Integration with Previous Components

The multithreaded server leverages the existing components from Phase 2:

- Client module (unchanged)
- Command parsing and execution
- I/O redirection
- Pipeline handling

The main enhancement is the addition of thread management and client-specific handlers to enable concurrent processing.

1.4 Data Flow Architecture

The data flow in the multithreaded client-server architecture follows this pattern:

1. Server initializes and begins listening for client connections
2. When a client connects, the server:
    - Accepts the connection
    - Creates a new thread to handle the client
    - Assigns a unique client ID
    - Logs the new connection with client information
3. Within each client handler thread:
    - The thread receives commands from its assigned client
    - The thread logs the received command with client identification
    - The thread executes the command using the shell infrastructure
    - The thread captures the command output
    - The thread logs the execution result with client identification
    - The thread sends the output back to the client
4. When a client disconnects:
    - The thread logs the disconnection with client identification
    - The thread cleans up resources and terminates
5. The server continues accepting new connections while existing threads handle their clients independently

1.5 File Organization and Code Structure

The project's file organization has been extended to include the multithreading components:

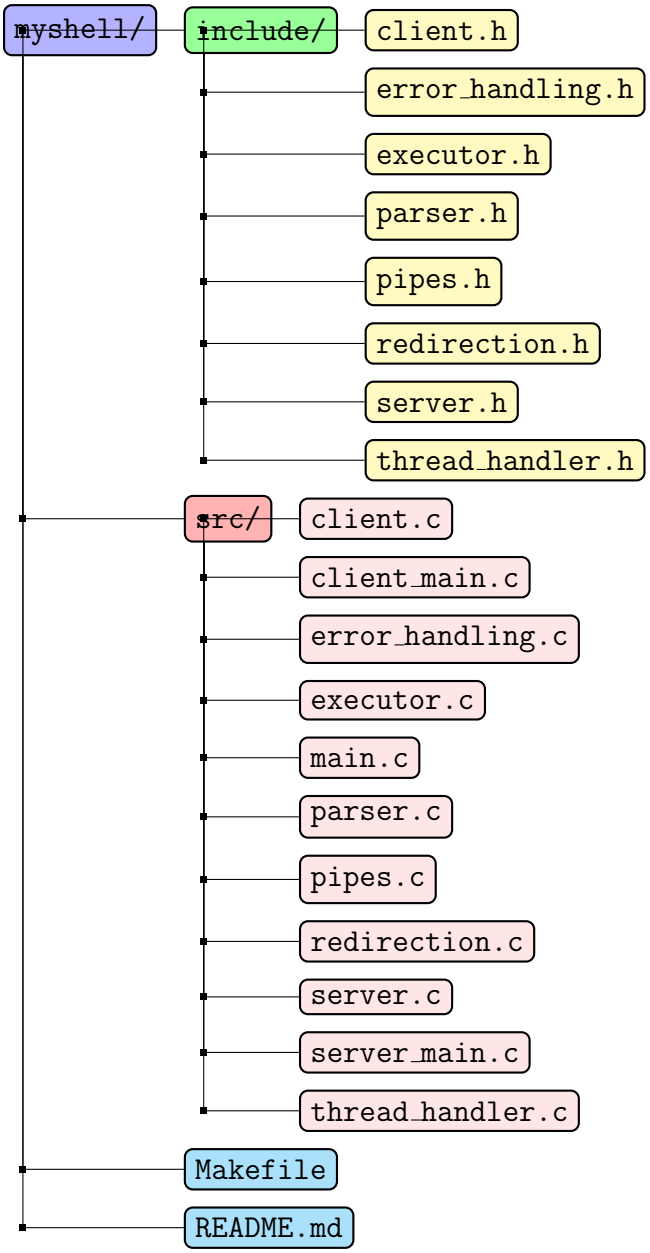

Figure 2: Shell Project Directory Structure for Phase 3

The new files added for Phase 3 are:

- `thread_handler.h` and `thread_handler.c`: Thread management and client handler implementation

The server.c file has been updated to support multithreaded operation, delegating client handling to the thread manager.

2 Implementation Highlights

2.1 Multithreaded Server Implementation

The core enhancement in Phase 3 is the implementation of a multithreaded server that can handle multiple clients simultaneously:

```c
void start_server(int port) {
    // Socket setup code...

    // Client connection counter for assigning IDs
    int client_count = 0;

    printf("Server started. Listening on port %d...\n", port);

    // Main server loop
    while (1) {
        // Accept client connection
        client_addr_len = sizeof(client_addr);
        client_socket = accept(server_socket,
                               (struct sockaddr *)&client_addr,
                               &client_addr_len);

        if (client_socket < 0) {
            perror("accept");
            continue;
        }

        // Create client info structure
        client_info *info = malloc(sizeof(client_info));
        info->client_socket = client_socket;
        info->client_addr = client_addr;
        info->client_id = ++client_count;

        // Create a new thread to handle this client
        pthread_t thread_id;
        if (pthread_create(&thread_id, NULL, handle_client, (void *)
    info) != 0) {
            perror("pthread_create");
            close(client_socket);
            free(info);
            continue;
        }

        // Detach thread to allow it to clean up automatically
        pthread_detach(thread_id);

        // Log new connection
        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, sizeof(
    client_ip));
        printf("[CONNECTED] Client #%d connected from %s:%d\n",
               client_count, client_ip, ntohs(client_addr.sin_port));
    }

    // Cleanup code...
}
```

Listing 1: Server with Thread Creation

2.2 Client Handler Thread Implementation

Each client connection is managed by a dedicated thread that handles the communication and command execution:

```c
void *handle_client(void *arg) {
    client_info *info = (client_info *)arg;
    int client_socket = info->client_socket;
    struct sockaddr_in client_addr = info->client_addr;
    int client_id = info->client_id;

    // Get client IP and port
    char client_ip[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &client_addr.sin_addr, client_ip, sizeof(client_ip));
    int client_port = ntohs(client_addr.sin_port);

    // Handle client communication
    char input[MAX_INPUT_SIZE];
    ssize_t bytes_received;

    // Receive data from client
    while ((bytes_received = recv(client_socket, input, sizeof(input) - 1, 0)) > 0) {
        input[bytes_received] = '\0';

        // Check if client wants to exit
        if (strcmp(input, "exit") == 0) {
            printf("[RECEIVED] [Client #%d - %s:%d] Received command: \"exit\"\n",
                   client_id, client_ip, client_port);
            char goodbye[] = "Disconnected from Server.\n";
            printf("[OUTPUT] [Client #%d - %s:%d] Sending response: \"Disconnected from Server.\"\n",
                   client_id, client_ip, client_port);
            send(client_socket, goodbye, strlen(goodbye), 0);
            break;
        }

        // Execute the command
        execute_shell_command(client_socket, input, client_ip, client_port, client_id);
    }

    // Check if there was an error receiving data
    if (bytes_received < 0) {
        perror("recv");
        printf("[ERROR] [Client #%d - %s:%d] Error receiving data\n",
               client_id, client_ip, client_port);
    }

    printf("[INFO] Client #%d disconnected from %s:%d\n", client_id, client_ip, client_port);

    // Close the client socket
    close(client_socket);
    free(info);

    return NULL;
}
```

Listing 2: Client Handler Thread Function

2.3 Command Execution Implementation

The implementation includes a robust command execution function that captures command output and handles different scenarios:

```
void execute_shell_command(int client_socket, const char *input, char *client_ip, int client_port, int client_id) {
    // Print received message
    printf("[RECEIVED] [Client #%d - %s:%d] Received command: \"%s\"\n",
           client_id, client_ip, client_port, input);

    // Print executing message
    printf("[EXECUTING] [Client #%d - %s:%d] Executing command: \"%s\"\n",
           client_id, client_ip, client_port, input);

    // Redirect stdout and stderr to capture the output
    int stdout_backup = dup(STDOUT_FILENO);
    int stderr_backup = dup(STDERR_FILENO);

    // Create pipes for capturing stdout and stderr
    int pipefd[2];
    if (pipe(pipefd) < 0) {
        perror("pipe");
        return;
    }

    // Redirect stdout and stderr to the pipe
    dup2(pipefd[1], STDOUT_FILENO);
    dup2(pipefd[1], STDERR_FILENO);
    close(pipefd[1]);

    // Execute the command using existing shell implementation
    if (strchr(input, '|')) {
        if (strstr(input, "||") != NULL) {
            fprintf(stderr, "Error: Empty command between pipes.\n");
        } else {
            execute_pipeline(input);
        }
    } else {
        // Parse and execute a single command
        Command *cmd = parse_command(input);
        if (cmd) {
            execute_command(cmd);
            free_command(cmd);
        } else {
            fprintf(stderr, "Parsing error.\n");
        }
    }

    // Flush the streams
    fflush(stdout);
    fflush(stderr);

    // Restore original stdout and stderr
    dup2(stdout_backup, STDOUT_FILENO);
    dup2(stderr_backup, STDERR_FILENO);
```

```
    close(stdout_backup);
    close(stderr_backup);

    // Read the output from the pipe
    char buffer[MAX_OUTPUT_SIZE];
    ssize_t bytes_read = read(pipefd[0], buffer, sizeof(buffer) - 1);
    close(pipefd[0]);

    // If there's output, send it to the client
    if (bytes_read > 0) {
        buffer[bytes_read] = '\0'; // Null-terminate the buffer

        // Check if the output contains an error message
        int is_error = (strstr(buffer, "Error:") != NULL ||
                        strstr(buffer, "not found") != NULL ||
                        strstr(buffer, ": missing operand") != NULL ||
                        strstr(buffer, "Parsing error") != NULL);

        if (is_error) {
            // Print error message
            printf("[ERROR] [Client #%d - %s:%d] %s",
                   client_id, client_ip, client_port, buffer);

            // Print output message
            printf("[OUTPUT] [Client #%d - %s:%d] Sending error message
     to client:\n\"%s\"\n",
                   client_id, client_ip, client_port, buffer);

            send(client_socket, buffer, bytes_read, 0);
        }
        else if (strcmp(input, "ls") == 0) {
            // For 'ls' command, perform special formatting to match
    Linux terminal
            char processed_buffer[MAX_OUTPUT_SIZE];
            int j = 0;

            // Replace newlines with spaces (except for the last one)
            for (int i = 0; i < bytes_read; i++) {
                if (buffer[i] == '\n' && i < bytes_read - 1) {
                    processed_buffer[j++] = ' ';
                } else {
                    processed_buffer[j++] = buffer[i];
                }
            }

            // Ensure we end with a newline
            if (j > 0 && processed_buffer[j-1] != '\n') {
                processed_buffer[j++] = '\n';
            }

            processed_buffer[j] = '\0';

            // Print output message
            printf("[OUTPUT] [Client #%d - %s:%d] Sending output to
    client:\n%s",
                   client_id, client_ip, client_port, processed_buffer);

            // Send the processed output to client
```

```
106             send(client_socket, processed_buffer, j, 0);
107         }
108         else {
109             // Normal output
110             // Print output message
111             printf("[OUTPUT] [Client #%d - %s:%d] Sending output to
    client:\n%s",
112                 client_id, client_ip, client_port,
113                 buffer[bytes_read-1] == '\n' ? buffer : strcat(buffer
    , "\n"));
114 
115             // Ensure output ends with a newline for consistency
116             int needs_newline = (bytes_read > 0 && buffer[bytes_read-1]
     != '\n');
117 
118             // Send the output to client
119             send(client_socket, buffer, bytes_read, 0);
120 
121             // Add a newline if needed
122             if (needs_newline) {
123                 send(client_socket, "\n", 1, 0);
124             }
125         }
126     } else {
127         // If there's no output, send an empty response with just a
    newline
128         printf("[OUTPUT] [Client #%d - %s:%d] Sending empty response (
    command had no output)\n",
129             client_id, client_ip, client_port);
130         send(client_socket, "\n", 1, 0);
131     }
132 }
```

Listing 3: Command Execution Function

2.4 Thread Safety Considerations

To ensure thread safety in our multithreaded environment, we implemented several measures:

1. **Thread-Local Storage**: Each thread maintains its own execution context and buffers to prevent interference between threads.

2. **Command Execution Isolation**: Command execution is isolated within each thread, ensuring that one client's commands don't affect another client's execution.

3. **Proper Resource Management**: Each thread is responsible for managing its own resources, including socket connections and memory allocations.

4. **Structured Logging**: The logging system is designed to clearly identify which client and thread is generating each log entry, preventing confusion in the server output.

2.5 Client Identification and Tracking

Each client connection is assigned a unique identifier to track and manage client interactions:

```
typedef struct {
    int client_socket;           // Socket descriptor for this client
    struct sockaddr_in client_addr;  // Client address information
    int client_id;               // Unique client identifier
} client_info;
```

Listing 4: Client Information Structure

This client identification is used throughout the server logs to clearly indicate which client is sending commands and receiving responses, following the required format specified in the project requirements.

3 Execution Instructions

3.1 Compilation

To compile the project, use the provided Makefile:

```
make
```

This will generate two executable files: ‘server‘ and ‘client‘.

3.2 Running the Server

To start the server, use the following command:

```
./server
```

This will prompt you to enter the port number on which the server will listen for client connections.

You can also specify a different IP address:

```
./server [differentIpSpecified]
```

By default, the server uses 127.0.0.1 as the IP address.

3.3 Running the Client

To start a client and connect to the server, use the following command:

```
./client
```

By default, the client connects to 127.0.0.1 as the server IP address.

3.4 Using the Shell

Once connected, you can use the shell as in previous phases. Type commands at the prompt and press Enter to execute them. The server will process the commands and return the results to the client.

To exit the shell, type ‘exit‘ at the prompt.

3.5 Note

If you're using code from the repository, make sure to perform a git pull to get the latest updates:

```
git pull
```

4 Testing and Evaluation

4.1 Test Case

Sample Server Output:

```
hm2957@DCLAP-V1525-CSD:~/proj_p3/operating_systems$ ./myshell -s 3000
[INFO] Server started, waiting for client connections...
[INFO] Client #1 connected from 127.0.0.1:62856. Assigned to Thread-1.
[INFO] Client #2 connected from 127.0.0.1:30022. Assigned to Thread-2.
[RECEIVED] [Client #1 - 127.0.0.1:62856] Received command: "unknowncmd"
[EXECUTING] [Client #1 - 127.0.0.1:62856] Executing command: "unknowncmd"
[ERROR] [Client #1 - 127.0.0.1:62856] Command not found: "unknowncmd"
[OUTPUT] [Client #1 - 127.0.0.1:62856] Sending error message to client:
"Command not found: "unknowncmd"
"
```

Figure 3: Terminal output

Sample Server Output:

```
Connected to server at 127.0.0.1:3000
$ unknowncmd
Command not found: "unknowncmd"
$
```

Figure 4: Terminal output

Sample Server Output:

```
hm2957@DCLAP-V1525-CSD:~/proj_p3/
Connected to server at 127.0.0.1:
$ exit
Disconnected from Server.
```

Figure 5: Terminal output

Sample Server Output:

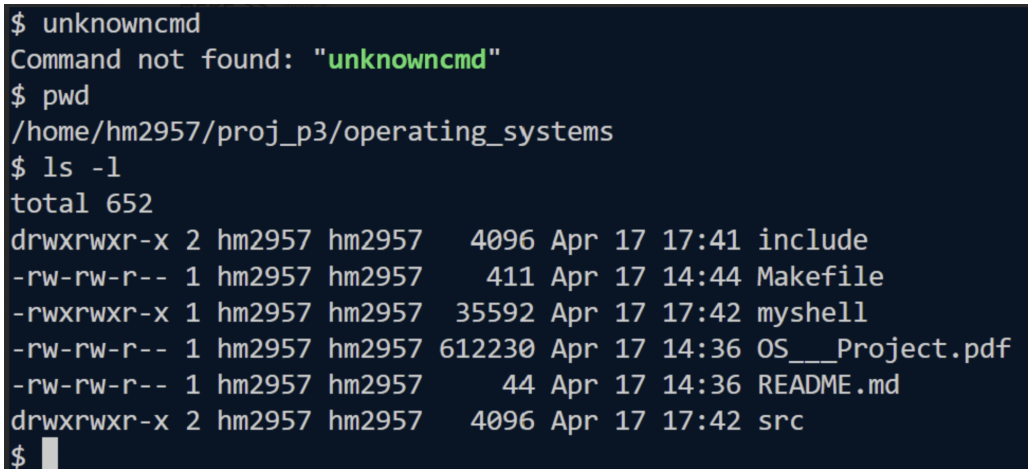

Figure 6: Terminal output

5 Challenges and Solutions

5.1 Thread Synchronization

Challenge: Ensuring thread safety when multiple threads are executing commands simultaneously.

Solution: We implemented thread-local storage for command execution contexts and ensured that each thread operates independently on its own resources. This eliminated the need for complex synchronization mechanisms while maintaining thread safety.

5.2 Client Disconnection Handling

Challenge: Properly detecting and handling client disconnections to prevent resource leaks.

Solution: We implemented robust error checking in the client handler thread to detect disconnections through socket read errors or zero-byte reads. Upon detecting a disconnection, the thread properly closes the socket, frees allocated memory, and terminates itself.

5.3 Server Output Formatting

Challenge: Ensuring that the server output follows the required format with proper client identification.

Solution: We designed a structured logging system that includes client ID, IP address, and port number in each log entry. The format follows the project requirements exactly, making it easy to track which client is sending commands and receiving responses.

5.4 Command Output Capture and Processing

Challenge: Capturing and processing command outputs, especially for special commands like 'ls' that require specific formatting.

Solution: We implemented a comprehensive output capture system using pipes to redirect both stdout and stderr, with special processing for certain commands. For the

'ls' command, we perform additional formatting to match the expected Linux terminal output format by replacing newlines with spaces.

5.5 Error Handling and Message Formatting

Challenge: Differentiating between normal outputs and error messages, and ensuring consistent formatting for both.

Solution: We implemented error detection by scanning for common error patterns in the command output. When an error is detected, it is logged with an [ERROR] tag and handled specifically to ensure proper formatting before sending it to the client.

6 Division of Tasks

- **Haad Mehboob (hm2957)**:
  - Implemented the thread management system
  - Developed the client handler thread function
  - Implemented thread safety measures
  - Conducted testing for concurrent command execution
  - Developed the structured logging system
  - Conducted testing for client connection and disconnection handling
- **Ameen Vadakkekara (av2851)**:
  - Modified the server to support multithreading
  - Implemented client identification and tracking
  - System architecture design
  - Integration testing
  - Documentation and report writing
  - Performance optimization

7 References

1. Operating Systems Concepts, 10th Edition, by Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne
2. Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). UNIX Network Programming, Volume 1: The Sockets Networking API (3rd ed.). Addison-Wesley Professional.
3. Butenhof, D. R. (1997). Programming with POSIX Threads. Addison-Wesley Professional.
4. Kerrisk, M. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press.
5. POSIX Threads Programming. (n.d.). Retrieved from https://hpc-tutorials.llnl.gov/posix/