📚 COP4610 Assignment 2: Multi-Threaded TCP Client-Server

📑 Table of Contents

📢 Important Note!

Please ensure that your implementation works on the course-designated ocelot server: ocelot-bbhatkal.aul.fiu.edu . Claims such as **"it works on my machine"** will not be considered valid, as all testing and grading will be conducted in that environment.

⚠️ **Important:** You are provided with an autograder to support consistent testing and maintain the highest level of grading transparency. The same autograder will be used by the instructor for final evaluation.

If your implementation passes all autograder tests with proper multi-client handling, timeout enforcement, and signal handling, you are likely to receive full credit for your submission.

You must use these strict gcc compilation flags while building your application — gcc -std=c17 -Wall -Werror -Wextra -O0 -g -lpthread

---

1. 📧 Grader Contact Information

For any questions or concerns related to assignment grading, please reach out to the TA listed below.

Communication Guidelines:

- Ensure that all communication is polite and respectful
- **You must contact the Graders first for any grading issues**
- If the matter remains unresolved, feel free to reach out to the instructor

Name: Instructor — bbhatkal@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U01 (84957)

Name: Angie Martinez — amart1070@fiu.edu (⚠️ via the Canvas inbox only)
Section: COP 4610-U02 (85031)

Name: Srushti Visweswaraiah — svisw003@fiu.edu(⚠️ via the Canvas inbox only)
Sections: COP 4610-UHA (85236)

---

2. 📖 Assignment Overview

This isn't just an assignment—**it's your gateway to understanding how the internet works!** The TCP socket programming, multithreading, and process control techniques you're mastering here power everything from SSH servers and web applications to video streaming platforms and online gaming. Major companies use these technologies to handle millions of users simultaneously. Every connection you make online relies on these concepts—now you're building them yourself!

This assignment provides **hands-on experience** with fundamental **TCP socket programming**, **multi-threaded server design**, and **UNIX process control**. You will implement a complete **multi-threaded shell server** capable of handling multiple concurrent client connections, executing remote shell commands, and managing system resources such as timeouts and file descriptors.

Through this implementation, you will explore how **network services handle concurrent requests**, how **servers manage client sessions**, and how **operating systems control process execution and resource allocation**. Your implementation must exhibit **concurrent execution, proper resource management, and reliable client-server communication**.Mastering these techniques will deepen your understanding of how modern network services achieve reliability, performance, and scalability.

---

3. 🛠️ Skills You'll Develop

In this assignment, you'll **gain hands-on experience** with both the **technical skills** and **operating system concepts** that are essential for professional systems programming and network service development such as:

Technical Skills:

- **Implement** TCP socket-based client-server architecture with proper connection handling
- **Manage** concurrent client sessions using POSIX multithreading (pthread)
- **Control** process execution with fork(), exec(), and process monitoring
- **Enforce** resource limits with timeout mechanisms and signal handling
- **Handle** inter-process communication using file descriptors and temporary files

- **Using** dup2() for output capture, temporary file management
- **Handling** SIGINT for graceful shutdown, SIGPIPE for broken connections
- **Enforcing** execution time limits using sleep() and process monitoring
- **Debug** network applications with proper error handling and resource cleanup

---

4. 🎯 Learning Objectives

- **Understand** and implement TCP socket-based client-server communication
- **Apply** multi-threading concepts to handle concurrent client connections
- **Use** process control mechanisms for command execution and resource management
- **Design** timeout enforcement systems to prevent runaway processes
- **Implement** signal handlers for graceful shutdown and cleanup
- **Optimize** network communication with proper buffering
- **Develop** scalable server architecture

---

5. ⚙️ Assignment Tasks

What You Must Implement

You will build a **multi-threaded TCP shell server and client application** that demonstrates the network service design:

1) Server Requirements:

A multi-threaded TCP server that manages concurrent client sessions and executes remote shell commands

Core Operations:

- Accept and handle **multiple concurrent client connections** on IP/port `127.0.0.1:8888` by creating a separate thread for every new client request
- Receive shell commands from each client and execute each of them by creating a new child process using `fork()` and `execlp()`.

  🦹 **HINT:**

  ```
  execlp("/bin/bash", "bash", "-c", command, NULL);
  ```

- For a valid shell command, the output of the command will be the response to the client. For an invalid (illegitimate) shell command, the error output of the shell will be sent as a response to the client.
- Enforce a ⏰ **3-second execution timeout** for each client command using `sleep(4)` and process monitoring. If a command runs for ⏰ **3 seconds or more**, the child process forked to execute that command will be **terminated**, and a **timeout response** will be sent to the client.
  - 🦹 **HINT:**

    You can test the timeout feature using the `sleep` command with a duration longer than 3 seconds. For example:

```
sleep 5
```

- ⚠️ **Important - Timeout Message Format:** When a command exceeds the **3-second** time limit, the server must send the following exact message format to the client:

```
Command Timeout (exceeded 3 seconds)
```

🕵️**HINT:**

```
const char* timeout_msg = "Command Timeout (exceeded 3 seconds)\n" RESPONSE_END_MARKER "\n";
send(client_socket, timeout_msg, strlen(timeout_msg), 0);
```

- Gracefully handle **SIGINT** (`Ctrl+C`) signal to allow **clean, graceful server termination**.
- Respond correctly to special commands: `HELP` and `EXIT`.
- **All server responses must end with** `<END_OF_RESPONSE>` marker for proper client-side parsing

2) Client Requirements:

A robust TCP client that communicates with the server following the specified protocol

Core Operations:

- Connect to server via TCP socket on `127.0.0.1:8888`
- Send shell commands to server and receive responses
- Handle user input with proper validation and formatting
- Gracefully disconnect on `EXIT` command or **SIGINT** (`Ctrl+C`)
- Display server responses with proper formatting

Implementation Standards

- Server must handle **2+ concurrent clients** without blocking
- Each command execution must enforce **exactly 3-second timeout** (not before, not significantly after)
- Implement **robust error handling** for all system calls (socket operations, process control, file operations)
- Follow **exact function signatures** provided in skeleton files for compatibility
- Ensure **memory safety** with proper resource cleanup (no leaks, no zombie processes, no orphaned files)
- Use **proper thread isolation** where each client thread operates independently with its own client socket and local variables. Command output is captured using unique temporary files (named by child PID) to prevent file conflicts between concurrent command executions.
- Handle **all edge cases**: connection failures, invalid commands, signal interruptions, resource exhaustion

---

6. 🗂️ Project Framework

This assignment includes a comprehensive development framework comprising **skeleton code files** (*templates for your implementation*), **reference executables**, an **autograder**s, and supporting infrastructure to guide your implementation process.

1) Provided Framework Infrastructure (*Do Not Modify*)

- `shell_server.h`: Complete header file with constants, data structures, and function prototypes.
- `autograder_socket.sh`: Automated testing script.
- `batchgrader_socket.sh`: Batch processing script for grading multiple student submissions.

2) Student Implementation Files

- `skeleton_server.c`: Server implementation template with guided TODO sections. Rename it to → `server.c`.
- `skeleton_client.c`: Client implementation template with guided TODO sections. Rename it to → `client.c`

3) Instructor's Reference Implementation

- `sample_server`: The instructor-provided reference executable that demonstrates the **complete, correct server implementation** of this assignment. 💡 **This executable serves as your primary reference for understanding expected server behavior.**
- `sample_client`: The instructor-provided reference executable that demonstrates the **complete, correct client implementation** of this assignment. 💡 **This executable serves as your primary reference for understanding expected client behavior.**

⚠️ **Important**: Your program's behavior must **exactly match** that of the instructor-provided executables for equivalent scenarios. Any deviation in behavior, protocol compliance, or output formatting may be failed by the autograder while evaluating your submission.

💡 **Help**: **Study the reference executables thoroughly before beginning implementation. Test your code incrementally against the sample executables.**

4) Implementation Guidance Documentation

- `A2_Specification.pdf`: Comprehensive specification document (this file)
- `A2_Testing_Guide.pdf`: Detailed testing procedures and validation strategies
- `README.txt`: Template for team information submission

5) Data Structure and Protocol Requirements

All implementations must follow the specifications provided in `shell_server.h`. Refer to this header file **thoroughly** to understand the constants (SERVER_PORT, BUFFER_SIZE, RESPONSE_END_MARKER, etc.), function prototypes, and protocol requirements.

6) Provided Files Layout

```
A2_PROVIDED_FILES/
├── Framework/
|   ├── shell_server.h              # Header file (provided - DO NOT MODIFY)|
|   ├── autograder_socket.sh        # Autograder (provided - DO NOT MODIFY)
|   └── batchgrader_socket.sh       # Batch grader (provided - DO NOT MODIFY)
├── Sample_Executable/
|   ├── sample_client               # Reference client implementation (study this thoroughly!)
|   └── sample_server               # Reference server implementation (study this thoroughly!)
├── A2_skeleton_codes/
```

```
|   ├── skeleton_server.c          # Server template (rename to server.c for submission)
|   └── skeleton_client.c          # Client template (rename to client.c for submission)
├── A2_Specification.pdf           # This detailed specification & requirement document
├── A2_Testing_Guide.pdf           # A comprehensive testing guide
└── README.txt                     # You must submit team details in the same format
```

7) ✅ Execution Permissions Notice

If you **do not** have execute permissions for the instructor-provided `sample_server`, `sample_client`, `autograder_socket.sh`, `batchgrader_socket.sh` and any other files, you can manually grant the required permissions using the following command on a **Linux system**.

> **Note:** When files are uploaded or transferred to Ocelot server (`ocelot-bbhatkal.aul.fiu.edu`), execute permissions **may be stripped** due to security restrictions. In such cases, you must explicitly grant **read (r)**, **write (w)**, and **execute (x)** permissions.

Granting Permissions to All Items in the Current Directory

```bash
# Check current permissions
ls -l

# Grant read, write, and execute permissions (safe and preferred – applies only what's missing)
chmod +rwx *

# Assign full permissions (read, write, execute) to all users for all files in the current directory
# ⚠️ Not preferred – use only as a last resort!
chmod 777 *

# Verify that permissions were applied
ls -l
```

⚠️ **Caution: Use `chmod 777` with care!** Granting full permissions to everyone can:

- Expose sensitive data
- Allow unintended modifications or deletions
- Introduce security risks, especially in shared or multi-user environments

8) ☑️ YOU MUST!

- **Test your implementation thoroughly** on the **Ocelot** server (`ocelot-bbhatkal.aul.fiu.edu`)
- **Ensure exact behavior matching** with the instructor sample executables `sample_server` and `sample_client`
- **Validate concurrent client handling** by testing with 3+ simultaneous client connections
- **Test edge cases** including timeout enforcement, signal handling, and error conditions
- **Verify compilation** with the specified compiler flags including pthread support

---

7. 🔧 Development & Testing

Required Files for Development

All the following files must be in the same directory:

1. **Framework files**: `shell_server.h`
2. **Your implementation files**: `server.c`, `client.c`
3. **Testing tools**: `autograder_socket.sh`, reference executables (`sample_server`, `sample_client`)

Comprehensive Development Workflow

> ⚠️ **Important**: **Please refer to the document A2_Testing_Guide.pdf for the detailed testing instructions.**

Step 1: Study Reference Implementations

```
# Study the expected server behavior
./sample_server

# In another terminal, test with the reference client
./sample_client

# Test with multiple concurrent clients to observe threading behavior
```

Step 2: Implement with Incremental Testing

Recommended Implementation Order:

1. **Basic Socket Setup (Server)**
   - Implement TODO 1: `setup_tcp_server()`
   - Test: Server should start and listen on port 8888
   - Verify: Use `netstat -tuln | grep 8888` to confirm
2. **Basic Connection (Client)**
   - Implement: `connect_to_server()`
   - Test: Client should connect to sample_server successfully
   - Verify: Connection establishment and welcome message
3. **Command Transmission (Client)**
   - Implement: `send_command()`
   - Implement: `receive_response()`
   - Test: Send simple commands and receive responses from sample_server
4. **Client Handling (Server)**
   - Implement: `handle_client_connection()`
   - Test: Your server with reference client

   - Verify: Single client connection and command execution
5. **Command Execution (Server)**
   - Implement: `execute_shell_command()`
   - Implement: `send_command_output()`
   - Test: Execute various shell commands (ls, pwd, date, echo)
   - Verify: Correct output and timeout enforcement
6. **Special Commands (Server)**
   - Implement: `handle_user_command()`
   - Test: HELP and EXIT commands
   - Verify: Proper responses and connection handling
7. **Signal Handling (Both)**
   - Implement (Server): `cleanup_and_exit()`
   - Implement (Client): `cleanup_and_exit()`
   - Test: Ctrl+C graceful shutdown on both server and client
   - Verify: Proper cleanup messages and resource deallocation
8. **Concurrent Testing**
   - Test: Open 3+ client terminals simultaneously
   - Verify: All clients can execute commands concurrently
   - Check: Thread IDs displayed for each client at server

Step 3: Use Autograder for Final Validation

```
# Run the autograders

./autograder_socket.sh

./batchgrader_socket.sh
```

⚠️ **Important**: **The instructor will use the same autograder for final grading.**

🤔Why No Complex Synchronization Techniques are Needed for this Assignment?

This assignment uses a carefully designed architecture that **minimizes the need** for complex synchronization mechanisms like **mutexes** or **semaphores**:

Design Principles:

- **Thread Isolation**: Each client connection handled by separate thread with independent resources
- **Process-Based Execution**: Shell commands run in separate child processes with own memory space
- **Minimal Shared Resources**: Only `server_running` flag and `tcp_server_socket` are shared
- **Simple Access Pattern**: Shared resources accessed in non-conflicting ways
- **Assignment Scope**: Focus on socket programming and process management, not advanced threading

💡 **Note**: Real-world multi-threaded servers often require more sophisticated synchronization for shared resources like connection pools and logging systems.

8. 📤 Assignment Submission Guidelines

- All assignments **must be submitted via Canvas** by the specified deadline as a **single compressed (.ZIP) file**. Submissions by any other means **WILL NOT** be accepted and graded!
- The filename must follow the format: `A2_FirstName_LastName.zip` (e.g., **A2_James_Bond.zip**)
- ⚠️ **Important:** Include the Firstname and Lastname of the **team's submitting member**, or your own name if **submitting solo**.
- ⚠️ **Important:** For **group submissions**, each group must designate one **corresponding member**. This member will be responsible for submitting the assignment and will serve as the primary point of contact for all communications related to that assignment.
- ⚠️ **Important Policy on Teamwork and Grading**

  All members of a team will receive the **same grade** for collaborative assignments. It is the responsibility of the team to ensure that work is **distributed equitably** among members. Any disagreements or conflicts should be addressed and resolved within the team in a **professional manner**. **If a resolution cannot be reached internally, the instructor may be requested to intervene** .
- ⚠️ **Important:** Your implementation **must be compiled and tested on the course designated Ocelot server** (`ocelot-bbhatkal.aul.fiu.edu`) **to ensure correctness and compatibility**. **Submissions that fail to compile or run as expected on Ocelot will receive a grade of ZERO.**
- Whether submitting as a **group or individually**, every submission must include a **README** file containing the following information:
  1. Full name(s) of all group member(s)
  2. **Prefix the name of the corresponding member with an asterisk** `*`
  3. Panther ID (PID) of each member
  4. FIU email address of each member
  5. Include any relevant notes about compilation, execution, or the programming environment needed to test your assignment. **Please keep it simple—avoid complex setup or testing requirements.**
- ⚠️ **Important - Late Submission Policy:** For each day the submission is delayed beyond the **Due date** and until the **Until date** (as listed on Canvas), a **10% penalty** will be applied to the total score. **Submissions after the Until date will not be accepted or considered for grading.**
  - **Due Date**: ANNOUNCED ON CANVAS
  - **Until Date**: ANNOUNCED ON CANVAS
- Please refer to the course syllabus for any other details.

---

9. 📦 Deliverable Folder Structure (⚠️ **exact structure required - no additional files / subfolders**)

> ⚠️ **DO NOT submit any other files other than these required files. The batch autograder may treat any additional items in the ZIP file as invalid, which will result in a grade of zero: server.c, client.c, README.txt**

> 📁 All other required framework files will be supplied by the instructor to the autograder during final grading.

- ⚠️ You are required to submit your work as **a single ZIP archive file** to Canvas.
- ⚠️ **Filename Format**: your `A2_Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.
- ⚠️ **For team submissions** – Include the first name and last name of the team's submitting member in the ZIP file name (e.g., `A2_James_Bond.zip`). **For solo submissions** – Use your own first and last name.

```
A2_FirstName_LastName.zip
├── server.c          # Your server implementation
├── client.c          # Your client implementation
└── README.txt        # Team information
```

---

10. 💯 Grading Rubric

Criteria	Points
⚠️ **Program fails to run or crashes during testing**	**0**
⚠️ **Partial Implementation** *(Evaluated based on completeness and at the instructor's discretion)*	**0 - 70**

Detailed Breakdown of 100 Points

Grading Component	Points	Description
1. Compilation	10	Must compile without errors using provided Makefile, handle all warnings with -Wall -Wextra flags, proper pthread linking, C99 standard compliance
2. Basic Functionality	25	TCP connection establishment on 127.0.0.1:8888, socket creation and binding, client-server communication, basic shell command execution (ls, pwd, date), welcome messages, HELP and EXIT command support, proper server response formatting with END_OF_RESPONSE markers
3. Threading	25	Handle multiple clients simultaneously using pthread_create(), proper thread detachment with pthread_detach(), isolated thread execution without data races, concurrent command processing, thread-safe resource management, scalable connection handling
4. Timeout Handling	10	Kill commands exceeding 3-second limit using SIGKILL, proper process monitoring with waitpid(WNOHANG), timeout message delivery to clients, child process cleanup after timeout, prevention of runaway processes, accurate timing enforcement
5. Signal Handling	10	Graceful shutdown on SIGINT (Ctrl+C) or EXIT command, proper cleanup of server socket and resources, clean termination of all client threads, removal of temporary files, orderly server shutdown with status messages

Grading Component	Points	Description
6. Error Handling	10	Invalid command handling with appropriate error messages, system call failure detection and recovery, socket disconnection handling, file access error management, memory allocation failure handling, signal interruption (EINTR) handling, graceful degradation on resource exhaustion
7. Submission & Code Quality	10	Clear code structure, Meaningful comments, well-written README, Overall readability
Total	**100**	

🚀 ***Success Strategy:*** The student autograder uses the **same grading mechanism** as the final evaluation, testing all aspects of your implementation including concurrent client handling, timeout enforcement, and signal handling. The final **10 points** for code quality will be based on:

- Proper comments
- Clean structure
- Adherence to guidelines
- Overall readability

Tips for success:

- Study the reference implementations thoroughly
- Implement features incrementally with continuous testing
- Ensure **exact behavior matching** with sample executables
- Test with multiple concurrent clients
- Maintain professional code quality throughout development

Good luck! 💪
