📋 Homework-4: TCP Command Server with Process Management

⚠️ **Important:** This assignment focuses on TCP socket programming combined with process management using fork(). Students will create a server that forks a new child process for each command received from a client.

You must use these strict gcc compilation flags while building your application:
-std=c17
-D_POSIX_C_SOURCE=200809L
-Wall
-Wextra
-Werror
-pedantic
-g
-O0
-pthread

⚠️ **There is only one submission - RESUBMISSION IS NOT ALLOWED. You must test your implementation with the provided testing framework on the Ocelot server before the final submission.**

📧 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)

✅ Section 1: Homework Overview

To create a TCP command server that accepts commands from a client and executes them in separate child processes, demonstrating both network programming and process management concepts.

✅ Section 2: Learning Objectives

- Implement TCP socket programming for client-server communication.
- Use `fork()` system call to create child processes for each command.
- Execute system commands using `system()` function.
- Understand socket communication patterns and process lifecycle management.
- Apply proper error handling in network and process contexts.

✅ Section 3: Problem Description

You will implement a **TCP command server** that accepts connections from clients and executes commands sent by those clients. The key requirement is that the server must **fork a new child process for each command received**, not just for each client connection.

Architecture Overview

1. **Server**: Listens on port 8080 and accepts one client connection
2. **Client**: Connects to server and sends commands from an input file
3. **Process Management**: Server forks a new child process for each command
4. **Command Execution**: Child processes execute commands using `system()` and report status back

Key Features

1. **TCP Socket Communication** – Server-client communication using TCP sockets
2. **Fork Per Command** – Create a new child process for each command (not per client)
3. **Command Execution** – Use `system()` to execute commands in child processes
4. **Status Reporting** – Send command success/failure status back to client
5. **Sequential Processing** – Handle commands one by one from the client
6. **Process Management** – Proper child process creation, execution, and cleanup

✅ Section 4: Provided Files Framework

📖 **Refer to these files** for understanding the exact requirements.

⚠️ **Please do not make any modifications to the framework files provided.** You are required to implement **only** the file `submission_HW4.c`. Any changes to the framework files may result in errors during evaluation or loss of credit.

Folder Structure

```
PROVIDED_FILES/
├── submission_HW4.c      # Template file (implement your solution here)
├── tcp_server.h          # Header file (Do not modify!)
├── driver.c              # Server startup framework (Do not modify!)
├── tcp_client.c         # Client program (Do not modify!)
├── Makefile              # Makefile (Do not modify!)
├── autograder_HW4.sh     # Testing script (Do not modify!)
├── batchgrader_HW4.sh    # Testing script (Do not modify!)
├── Sample_Executable/
|   ├── server            # TCP server sample executable
|   ├── client            # TCP client sample executable
└── Testing/              # Testcases (Do not modify!)
    ├── Testcases/
    |   ├── input1.txt    # Valid commands only
    |   ├── input2.txt    # Invalid commands only
    |   └── input3.txt    # Mixed valid/invalid commands
    └── Expected_Output/
        ├── output1.txt   # Expected output for test 1
        ├── output2.txt   # Expected output for test 2
        └── output3.txt   # Expected output for test 3
```

☑ Section 5: Submission Requirements

File Requirements (⚠ exact names required)

- **submission_HW4.c**: Your complete implementation
- **README.txt or README.pdf**: Student information and documentation
    - Your README ⚠ **MUST include**:

```
# Student Information
- Full Name: [Your Full Name]
- PID: [Your FIU Panther ID]
- Section: [Your Course Section]
- FIU Email: [Your @fiu.edu email]

# Homework: TCP Command Server
[Brief description of your implementation approach]
[Mention how you implemented the fork-per-command architecture]
[Describe any challenges faced and how you solved them]
```

Deliverable Folder (`ZIP file`) 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: submission_HW4.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**.
- ⚠️ **Filename Format**: your `Firstname_Lastname.zip` (**exact format required**).
- **You will be held responsible for receiving a ZERO grade if the submission guidelines are not followed**.

```
Harry_Potter.zip
├── submission_HW4.c  # Your server implementation
└── README.txt        # Your details and implementation approach
```

☑️ Section 6: Developing and Testing Your Implementation

⚠️ **DO NOT modify the following provided framework files:**
autograder_HW4.sh, batchgrader_HW4.sh, tcp_server.h, driver.c, test_client.c, Makefile, input.txt, and output.txt.

Required Functions

You must implement these functions in `submission_HW4.c`:

`void handle_client_request(int client_socket, struct sockaddr_in client_addr)`

- **Input**: Client socket descriptor and client address structure
- **Process**:
  1. Accept commands from client one by one in a loop
  2. For each command received, fork a new child process
  3. Child process executes command and sends status back
  4. Parent process waits for child completion and continues
  5. Handle client disconnection gracefully
- **Output Format:**

```
Server: Connected to client [IP]:[PORT]
Server: Received command: [command]
Server: Child <pid> started for command: [command]
Server: Child <pid> completed
Server: Client [IP]:[PORT] disconnected
```

`void execute_command_in_child(int client_socket, const char *command)`

- **Input**: Client socket and command string to execute
- **Process**:
  1. Print child process information
  2. Execute command using `system()`
  3. Check return status and create appropriate message
  4. Send status message to client via socket
  5. Close socket and exit child process
- **Output Format:**

```
Child <pid>: Executing command: [command]
```

Command Execution Requirements

- **Use `system()` function** to execute commands in child processes
- ⚠️ **IMPORTANT NOTE:** Your TCP server implementation must use `system()` to execute commands received from the client but should NOT capture or send the actual command output back to the client - instead, it must only send status messages indicating whether the command completed successfully or failed. When commands like `echo hello` are executed, their output (e.g., "hello") will appear on the server terminal but will NOT be sent to the client
- **Status Messages**:
  - Success (system() returns 0): `"Command 'command' completed successfully"`
  - Failure (system() returns non-zero): `"Command 'command' failed"`
- **Process Flow**:
  - Child: Execute command → Send status → Exit
  - Parent: Wait for child → Continue accepting commands

Example Input/Output

Sample Input (`input1.txt`):

```
# Valid commands
true
echo hello
false
```

Expected Output Pattern:

⚠️ **Note: The server terminal output is followed by the client terminal output, for all three expected outputs.**

```
# Server terminal output
Starting TCP Command Server...
Server: Listening on port 8080...
Server: Waiting for client connection...
Server: Connected to client 127.0.0.1:XXXXX
```

```
Server: Received command: true
Server: Child <PID> started for command: true
Child <PID>: Executing command: true
Server: Child <PID> completed
Server: Received command: echo hello
Server: Child <PID> started for command: echo hello
Child <PID>: Executing command: echo hello
hello
Server: Child <PID> completed
Server: Received command: false
Server: Child <PID> started for command: false
Child <PID>: Executing command: false
Server: Child <PID> completed
Server: Client 127.0.0.1:XXXXX disconnected
Server: Shutting down

# Client terminal output
Client: Connected to server
Client: Sending command: true
Client: Received: Command 'true' completed successfully
Client: Sending command: echo hello
Client: Received: Command 'echo hello' completed successfully
Client: Sending command: false
Client: Received: Command 'false' failed
Client: Disconnected from server
```

📝 **Note about Command Output:**

- The line `hello` appears in the server terminal because the `echo hello` command executed successfully
- This command output is NOT sent to the client - only the status message is sent
- The client receives only: `"Command 'echo hello' completed successfully"`

STEP 1 - Sample Executable Testing:

💡 To better understand the client–server implementation requirements, you are **highly encouraged** to test the instructor-provided executables for both the server and client:

```
# Provide execute permissions to both server and client executables:
chmod 777 server client

# Run the server in one terminal
./server

# Run the client in another terminal with an input test case
./client Testing/Testcases/input1.txt

# Try with the other two test cases in the same way
./client Testing/Testcases/input2.txt
./client Testing/Testcases/input3.txt
```

STEP 2 - Manual testing:

Ensure the following files are located in the **same directory**:

- **Your implementation**: `submission_HW4.c`
- **All the provided framework files**
- **The provided testcase folder:** `Testing/`

```
# Explore the Makefile first to understand build targets and compilation settings
cat Makefile

# Build both server and client executables
make rebuild

# Run server in one terminal
./server

# Run the client in another terminal
# Test with:
#   - Valid commands:    input1.txt
#   - Invalid commands:  input2.txt
#   - Mixed commands:    input3.txt
./client Testing/Testcases/input1.txt
./client Testing/Testcases/input2.txt
./client Testing/Testcases/input3.txt

# Prepare the student output files by copy-pasting the outputs from both
# the server terminal and the client terminal into the following files:
student_output1.txt
student_output2.txt
student_output3.txt

⚠️ Normalizing the process IDs and PORT numbers in student_output.txt.
# (The autograder replaces all actual PIDs with <PID> before comparison.)
# For example, your output:
#  - Process IDs: <1234> becomes <PID>
# will match the expected pattern:
#  - Port numbers: 127.0.0.1:54321 becomes 127.0.0.1:XXXXX

# Compare your output with the expected output:
diff student_output1.txt Testing/Expected_Output/output1.txt
diff student_output2.txt Testing/Expected_Output/output2.txt
diff student_output3.txt Testing/Expected_Output/output3.txt
```

STEP 3 - Autograder testing:

> ⚠️ **Important:** ⚠️ The autograder may take longer to test your application due to client–server interactions. Please be patient!

⚠️ The autograder may take longer to test your application due to client–server interactions. Please be patient!

Ensure the following files are located in the **same directory**, and then run the **Autograder**:

- **Your implementation**: `submission_HW4.c`
- **All the provided framework files**
- **The provided testcase folder:** `Testing/`

```
# Run the autograder
./autograder_HW4.sh
```

- **Check your grade** and fix any issues
- **Repeat until** you achieve the desired score

STEP 4️⃣ - Batch Autograder testing:

> ⚠️ **Important:** ⚠️ The batch autograder may take even longer than the autograder to test your application due to additional testing steps. Please be patient!

The batch autograder, **used by the instructor**, processes all student submissions at once. It utilizes the provided framework files, extracts the required files from your submitted `.zip` file, and performs grading accordingly.

⚠️ **Backup Files Before Running Batch Grader - The script deletes existing files to rebuild the environment from scratch.**

Ensure the following files are located in the **same directory**, and then run the **Batch Autograder**:

- **All the provided framework files**
- **The provided testcase folder:** `Testing/`
- **Your final submission `ZIP` file** consisting the required files for the submission: example, `Harry_Potter.zip`

```
# Run the batch-autograder
./batchgrader_HW4.sh
```

Final Testing requirements:

- ⚠️ **MUST test on ocelot server**: `ocelot-bbhatkal.aul.fiu.edu`
- ⚠️ **Test thoroughly before submission** - no excuses accepted
- ⚠️ **"Works on my computer" is NOT accepted** as an excuse
- ✅ **If you pass all test cases on the server, you will likely pass instructor's final grading**
- ✅ **What you see is what you get** - autograder results predict your final grade

✅ Section 7: Grading Criteria

🤖 Autograder Testing

- ⚠️ Your implementation `submission_HW4.c` will be tested against the provided test case as well as additional instructor test cases using the autograder.
- ⚠️ **Exact output matching required** - any deviation results in point deduction
- ⚠️ **Program must compile** without errors or warnings

Test Case Distribution:

- **Test 1 (30 points)**: Valid commands only - tests successful process execution and status reporting
- **Test 2 (30 points)**: Invalid commands only - tests error handling for non-existent commands
- **Test 3 (40 points)**: Mixed valid/invalid commands - tests comprehensive functionality including:
  - Proper fork-per-command behavior
  - Correct client-server communication
  - Process management and cleanup
  - Recovery after command failures

⛔ Penalties

- ⚠️ **Missing README**: **-10 points**
- ⚠️ **Missing** `submission_HW4.c` **ZERO grade** (autograder compilation failure)
- ⚠️ **Incorrect ZIP filename**: **ZERO grade** (autograder compilation failure)
- ⚠️ **Wrong source filename**: **ZERO grade** (autograder compilation failure)
- 🔒 **Resubmission**: **NOT ALLOWED**

✅ Section 8: Technical Specifications

Compilation Requirements

- **Build System**: A complete Makefile is provided as part of the framework
- **Compiler Standards**: The Makefile uses strict C17 compliance with POSIX.1-2008 extensions
- **Important**: Explore the provided Makefile to understand the compilation flags and build targets
- **Cross-platform**: The build configuration ensures compatibility across WSL, Ubuntu, and macOS
- **Implementation Checks**: The Makefile includes automated checks to verify your implementation contains required functions

Socket Programming Requirements

- Server listens on **port 8080**
- Accept **one client connection** at a time
- Use **TCP sockets** (SOCK_STREAM)
- Handle client disconnection gracefully

Process Management Requirements

- **Fork for each command** received (not per client)
- Child processes use **system()** to execute commands
- Parent process **waits** for child completion using **waitpid()**
- Proper **zombie process** reaping

Communication Protocol

- Client sends commands as strings
- Server receives commands and forks for execution
- Child processes execute commands using system() - command output appears on server terminal
- Child sends **only status message** back to client (not command output)
- Client displays received status messages

Error Handling

- Check **fork()** return value for failure
- Handle **socket errors** appropriately
- Manage **command execution failures**
- Implement **timeout protection** in autograder

✅ Section 9: Academic Integrity

- 👤 **This is an individual assignment**
- 💬 **You may discuss concepts** but not share code
- 📝 **All submitted code** must be your original work
- 📚 **You are encouraged to adapt from instructor provided socket examples** but must understand and modify them appropriately
- ⚠️ **Plagiarism** is a serious offense and will result in penalties - **ZERO grade** (academic integrity violation).

🚀 **Good luck with your TCP command server implementation!**