CSE 344 System Programming
Hw4 Report

Mehmet Hüseyin YILDIZ

Client Side

This part represents a client application that connects to a server, sends requests, receives responses, and performs file upload and download operations.

Overall Structure

The `biboClient` code is written in C and utilizes various system calls, library functions, and pthreads. Here is an overview of the main components and their functionality:

1. `client` Structure: This structure holds the necessary information and resources related to a client, including file descriptors, paths, semaphores, and mutexes.

2. Connection Establishment: The `connect` function is responsible for establishing a connection with the server. It opens the server connection semaphore and server write mutex, opens the server FIFO for writing, and waits for the connection semaphore to be available.

3. Signal Handling: The `sigintHandler` function is a signal handler for the `SIGINT` signal (Ctrl+C). It sets the `exit_now` and `send_quit` flags, indicating that the client should exit gracefully and send a quit request to the server.

4. Initialization: The `init_client` function initializes the client structure with the necessary paths and names based on the provided server PID. It also sets up the signal handler.

5. Argument Checking: The `arg_check` function validates the command-line arguments and returns a boolean value indicating whether to try connecting or connect immediately.

6. Request Sending: The `send_request` function sends a request to the server. It acquires the write mutex, writes the process ID, message length, and message content to the server FIFO, and releases the write mutex.

7. Response Printing: The `print_response` function reads the response from the client's FIFO and prints it to the specified file descriptor (stdout or a file). It opens the client FIFO for reading, reads the response in chunks, and writes it to the output.

8. File Downloading: The `download_file` function handles the downloading of a file from the server. It checks for the availability of the file, opens a new file for writing (with a unique name if necessary), and calls `print_response` to write the file content to the output file.

9. File Uploading: The `upload` function handles the uploading of a file to the server. It reads the file content from the specified file, creates a FIFO for communication with the server, and writes the file content to the FIFO using `write`.

10. Main Loop: The main function contains the main loop of the client program. It reads user commands, sends requests to the server, handles special commands (such as upload, download, and quit), and prints the server's responses.

Error Handling

The code includes basic error handling using `perror` to print error messages in case of failure. It checks the return values of system calls and library functions and exits the program with an appropriate error code when necessary.

Thread Safety

The code does not explicitly utilize pthreads for parallel execution. It runs in a single thread and uses synchronization mechanisms such as semaphores and mutexes to ensure thread safety when accessing shared resources, such as the server FIFO and client FIFOs.

Limitations and Possible Improvements

1. Error Handling: The code could benefit from more comprehensive error handling and recovery mechanisms. Currently, it relies on basic error checking, but it could be enhanced to handle various error scenarios more gracefully and provide better error messages to the user.

2. Robustness: The code assumes that the server and client will always be executed in a specific order and that the necessary directories and resources are already created. It would be helpful to add checks for the existence of directories, create them if needed, and handle scenarios where the server is not available or terminates unexpectedly.

3. Command Parsing: The code currently assumes that user commands are provided in a specific format. Enhancements could be made to handle different command formats, validate command parameters, and provide more informative error messages for invalid commands.

4. Code Organization: The code could benefit from improved code organization and modularization. Breaking down the functionality into smaller, well-defined functions could improve readability and maintainability.

5. Testing: More extensive testing could be conducted to verify the behavior and correctness of the client application in various scenarios, such as concurrent client connections, handling large file transfers, and stress testing.

6. Documentation: Inline comments and function-level documentation could be added to clarify the purpose and usage of various functions and data structures, making the code more readable and maintainable.

7. Code Optimization: Depending on the specific requirements and constraints of the system, there may be opportunities for optimizing the code in terms of resource usage, performance, and scalability. Profiling and analyzing the code could help identify potential areas for optimization.

Conclusion

The `biboClient` provides a basic client implementation for interacting with a server. It establishes a connection, sends requests, receives responses, and handles file upload and download operations. The code demonstrates the usage of various system calls, library functions, and synchronization mechanisms to achieve the desired functionality.

Server Side

This is a detailed design report for the pthread FIFO filesystem server code. The code represents a server application that handles client requests using pthreads, FIFOs, and a queue.

Overall Structure

The code is written in C and utilizes pthreads, FIFOs, and a queue implementation. Here is an overview of the main components and their functionality:

1. `server` Structure: This structure holds the necessary information and resources related to the server, including the maximum client capacity, the directory the server runs on, file descriptors, paths, semaphores, and mutexes.

2. `request` Structure: This structure represents a client request, containing the process ID and the request message.

3. Signal Handling: The `sigintHandler` function is a signal handler for the `SIGINT` signal (Ctrl+C). It sets the `terminate` flag, indicating that the server should exit gracefully.

4. Request Handling Threads: The `handle_request` function is the entry point for the threads that handle client requests. Each thread dequeues a request from the request queue, processes it, and sends the response back to the client.

5. Initialization: The `init_server` function initializes the server structure and performs the necessary initial operations. It creates a directory for temporary files, creates a FIFO for incoming requests, opens the FIFO for reading, creates or opens the named connection semaphore and write mutex, and sets up the signal handler.

6. Request Handling: The `get_request` function reads a request from the server FIFO. It allocates memory for the request, reads the process ID and the message length, and reads the message content.

7. Request Queue: The `enqueue` and `dequeue` functions are used to add requests to the queue and retrieve requests from the queue, respectively. They handle the synchronization using a mutex and a semaphore.

8. Thread Initialization: The `init_threads` function initializes the server threads that handle requests. It creates the specified number of threads and assigns the `handle_request` function as the thread function.

9. Main Loop: The `main` function contains the main loop of the server program. It reads requests from the server FIFO, logs the requests, checks for termination signals, and adds the requests to the request queue.

Error Handling

The code includes basic error handling using `perror` to print error messages in case of failure. It checks the return values of system calls and library functions and exits the program with an appropriate error code when necessary.

Thread Safety

The code utilizes pthreads to achieve parallel execution for handling client requests. It uses synchronization mechanisms such as a mutex and a semaphore to ensure thread safety when accessing shared resources, such as the request queue and the server FIFO.

Limitations and Possible Improvements

1. Error Handling: The code could benefit from more comprehensive error handling and recovery mechanisms. Currently, it relies on basic error checking, but it could be enhanced to handle various error scenarios more gracefully and provide better error messages to the user.

2. Scalability: The code sets a maximum limit on the number of simultaneous clients and the number of threads. It could be improved to handle a dynamic number of clients and dynamically adjust the number of threads based on the workload.

3. Robustness: The code assumes that the necessary directories and resources are already created and available. It would be helpful to add checks for the existence of directories, create them if needed, and handle scenarios where the required resources are not available.

4. Command Handling: The code currently focuses on handling a limited set of commands, such as processing client requests and terminating the server. Enhancements could be made to handle a broader range of commands, allowing for more complex interactions with the server.

5. Code Organization

The code could benefit from better code organization and modularization. Breaking down the functionality into smaller, reusable functions and separating concerns could improve readability and maintainability.

6. Documentation: Inline comments and function-level documentation could be added to clarify the purpose, usage, and limitations of various functions and data structures, making the code more understandable for future development and maintenance.

7. Testing: More extensive testing could be conducted to verify the behavior and correctness of the server application in various scenarios, such as concurrent client connections, stress testing, and handling large file transfers.

8. Performance Optimization: Depending on the specific requirements and constraints of the system, there may be opportunities for optimizing the code in terms of resource usage, performance, and scalability. Profiling and analyzing the code could help identify potential areas for optimization.

Conclusion

The pthread FIFO filesystem server code provides a basic implementation for handling client requests using threads, FIFOs, and a queue. It demonstrates the usage of various system calls, library functions, synchronization mechanisms, and signal handling to achieve the desired functionality. The code can serve as a starting point for building a more robust and scalable server application, with improvements in error handling, scalability, robustness, and code organization.

Thread Safe Queue

Mutex Locking/Unlocking: Before accessing the queue, the enqueue and dequeue functions lock the queue's mutex using pthread_mutex_lock to ensure that only one thread can modify the queue at a time. After the modification is complete, the mutex is unlocked using pthread_mutex_unlock.

Semaphore Signaling: The queue uses a semaphore to handle synchronization between the producer and consumer threads. The enqueue function calls sem_post after adding an element to the queue, which signals to any waiting consumer thread that an element is available for dequeueing. The dequeue function calls sem_wait to wait for the semaphore to be signaled before dequeuing an element. This ensures that the consumer thread waits until an element is available before attempting to dequeue.

Tests

All tests screen images are in the tests directory.