Concurrent File Access System

Tugay Talha İçen

May 5, 2024

1 Introduction

This report outlines the design and implementation of a concurrent file access system, which enables multiple clients to connect, access, modify, and archive files in a specific directory located on the server side. The project is implemented as a server-side program and a client-side program, with the server managing the file operations and the clients issuing various commands to interact with the files.

2 System Architecture

The system architecture consists of the following components: (TO see the structure look at the png file in same directory with this report)

2.1 Server-side Program (neHosServer)

The server-side program, `neHosServer`, is responsible for the following tasks:

- Creating and managing the directory specified by the user, as well as a log file for client activities.
- Handling client connection requests and managing a queue of connected clients.
- Forking a child process for each connected client to handle their file operations.
- Implementing mutual exclusion and synchronization mechanisms to ensure data consistency and prevent race conditions.
- Handling signal interrupts (e.g., Ctrl-C) and gracefully terminating all child processes and the server itself.

2.2 Client-side Program (neHosClient)

The client-side program, `neHosClient`, allows users to interact with the server and perform the following operations:

- Requesting a connection to the server and waiting in the queue if the maximum number of clients is reached.
- Issuing various commands to the server, such as listing files, reading from files, writing to files, uploading files, downloading files, archiving the server directory, and terminating the server.
- Receiving and displaying the responses from the server for the issued commands.

3 Design Decisions and Implementation Details

3.1 File I/O Module

The file I/O module is responsible for handling read and write operations for different file formats (e.g., text, binary). It uses the standard file I/O functions provided by the C standard library, such as `open()`, `read()`, `write()`, and `close()`, to perform these operations.

3.2 Synchronization Module

The synchronization module implements mutual exclusion and prevents race conditions using file locking mechanisms. Specifically, it uses the `fcntl()` system call to acquire and release locks on files before performing read and write operations. This ensures that only one client can access a file at a time, preventing data corruption and maintaining data consistency.

3.3 Concurrent File Access

The server-side program uses the `fork()` system call to create a child process for each connected client. Each child process is responsible for handling the file operations requested by its corresponding client. This approach allows multiple clients to access files concurrently, with the server managing the synchronization and coordination between the child processes.

3.4 Large File Handling

To handle large files (i.e., greater than 10 MB), the system uses a chunked approach for file transfers. When a client requests to upload or download a file, the server and client exchange the file content in smaller chunks, with the server sending or receiving the data in multiple iterations. This approach ensures that the system can handle files of any size without running into memory constraints.

3.5 Signal Handling

Both the server-side and client-side programs handle signal interrupts, such as Ctrl-C, using the `signal()` and `sigaction()` functions. When a signal is received, the programs perform the necessary cleanup tasks, such as closing file descriptors, removing temporary files, and terminating child processes, before exiting.

4 Testing and Evaluation

The system was thoroughly tested to ensure that it meets the requirements. The test plan included the following scenarios:

- Multiple clients connecting to the server and performing various file operations concurrently.
- Handling of large files (both text and binary) during upload, download, and read/write operations.
- Verifying data consistency and the absence of race conditions by performing simultaneous file access.
- Testing the server's ability to handle signal interrupts and gracefully terminate all child processes.
- Validating the correctness of the file archiving and server termination functionalities.

The test results demonstrated that the system successfully meets all the requirements, with no data corruption or inconsistencies observed during the concurrent file access operations.

5 Conclusion

The Concurrent File Access System provides a robust and efficient solution for enabling multiple clients to access, modify, and archive files on a server-side directory. The system's design, which includes a file I/O module, a synchronization module, and a concurrent file access mechanism, ensures data consistency and prevents race conditions. The system's ability to handle large files and signal interrupts further enhances its reliability and usability. The thorough testing and evaluation process has validated the system's compliance with the project requirements, making it a reliable and practical solution for file management in a concurrent environment.