SAPIENZA
UNIVERSITÀ DI ROMA

Master's Degree in Cybersecurity

Computer Systems and Programming

Project Report

Author:
Riccardo Tuzzolino

University ID:
1954109

Academic Year 2023/24

Contents

Chapter 1

Introduction

The objective of the assigned project was to implement a "**log server**", i.e. a server that writes to a log file the inputs it receives from clients.

The goal of this report is to explain how I solved the main problems I encountered in the implementation of such server and why I made certain choices.

The remainder of this document is organized in three sections. Chapter 2 discusses the idea behind the architecture of the client-server application I implemented. Chapter 3 describes the method I used to coordinate the access of multiple processes to the same log file. Chapter 4 is concerned with how I handled the termination of the server. Finally, in Chapter 5 I discuss my implementation of the logs rotation mechanism.

Chapter 2

Architecture Idea

By the specifications of the project, the server should be able to manage an unlimited number of clients. It means that we are asked to implement a "concurrent server": a server that is able to handle multiple clients at the same time.

To implement such server, as suggested, I used the **fork()** system call to create a dedicated child process to handle each client. The idea behind this mechanism is the same one that is used by **inetd** (internet service daemon).

In order to provide a network service, a permanently available server application must be kept running to receive client requests through the protocol and port number established for the purpose. This type of configuration makes perfect sense for high-utilization network services for which the shortest possible response time is desirable. For low-utilization network services for which response time is not critical, inetd offers an alternative that avoids the need for a running process for each network service, thus reducing the load on the system.

The inetd network service allows to replace multiple daemons, which are idle waiting for a contact from a client, with a single running process. To achieve this goal, once contacted by a client, inetd creates a child process (fork()) which is then replaced (exec()) by the actual server application process. By doing so, it is the server application that will effectively dialogue with the client and not inetd.

Similarly, in my log server, whenever a client contacts the server, it forks a child process to handle each client request.

The advantage of using fork() call in a client server application with multiple clients is that each client connection can be handled in a separate process. As multiple clients can be served concurrently, this enhances the application's responsiveness and scalability. Each forked process operates independently, avoiding interference between clients.

However, proper synchronization mechanisms are necessary to manage shared resources and ensure data consistency. In fact, as multiple child processes are going to access the same log file (it is unique for all clients) to write the message received by the clients, we have to implement a mechanism to synchronize access to the file to avoid race conditions and ensure data integrity.

Chapter 3

Advisory File Locking

What happens if one process writes a file while others read it? Processes that read can read partially or incompletely written information. What happens if multiple processes write to the same file at the same time? The output to the file is mixed unpredictably: a process can overwrite the output of another process.

In order to solve this problem I decided to implement a File Locking mechanism: it allows to block the access of other processes to a file (or parts of it), so as to avoid overlaps and guarantee the atomicity of the writing operation (only one child process at a time will have exclusive access to the log file). The lock is released after the writing operation is completed.

In particular, there are two types of locking mechanisms: Mandatory and Advisory.

Mandatory locking (disabled by default) is enforced by the kernel, so it is the system that prevents a process from accessing the file (or parts of it) if another process holds the lock. So, unlike advisory locking, mandatory locking doesn't require any cooperation between the participating processes.

In the Advisory Locking mechanism, the one I implemented, the access control is performed by the individual processes, i.e. processes must explicitly check the state of the file before accessing it. It means that the processes could still read and write from a file even if there is a lock on it, so they must cooperate with each other, by implementing a common protocol, to ensure that locks are properly observed: each process has to check for the existence of a lock before a read or write operation.

So, the common protocol I implemented for all the processes is the following:

```
1 Lock request; (blocking)
2 If the process receives the lock:
3     Perform file operations (read or write);
4     Release the acquired lock;
```

It means that, every time a process wants to write on the log file, it has to request the lock for the entire file first. This request is blocking, i.e. if another process holds the lock, then the process will block until the lock can be acquired (instead of immediately return an error). This can be achieved using the flag F_SETLKW in the fcntl() system call.

After the process has acquired the lock, it can append a line to the log file and finally release the lock.

The Advisory Locking mechanism is implemented in the logReceivedMessage() function I wrote in the logServer.c source code. This function is called every time a child process wants to append the message received by the client to the log file.

Chapter 4

Signal Handling

As we can read from the specifications: "The server should shutdown when it receives a user-selected signal or a quit command from the command line", after recording the order of shutdown in the log file.

To implement this specific I chose to define a signal handler for the SIGINT signal. SIGINT is a signal that is generated when we press Ctrl+C from the command line and it terminates the program. So, the idea is that when we will press Ctrl+C from the command line of the running server, the registered signal handler will first write inside the log file that the server was shut down and then shut down the server.

However, as the child process inherits the signal handler for SIGINT from the parent process, this means that when we press Ctrl+C then also the child process will try to write on the log file that the server was shut down.
To solve this issue I used a global variable called 'is_main_process' that can be only 0 or 1. By default, this variable is set to 1, but when we fork a child process it will set the variable to 0 (a separate copy, so it does not affect the value in the parent process).

So, in the signal handler I defined an if-else statement such that: whenever a process executes the signal handler, if it is a child process (is_main_process == 0) then it just terminates, otherwise it means it is the main process so it writes on the log file and then terminates.

Chapter 5

Logs Rotation

Among the specifications there was the following one for the extra bonus:

"When the log file size exceed a given threshold, the server should cancel the oldest log file in the log directory and create a new log file. In this case, the server should not create a new log file at start-up, but rather append to the most recent log file in the directory."

In order to implement such requirement I thought to name the log files in an ascending order. For example, supposing that the maximum number of possible log files inside the directory (MAXLOGFILE) is 3, then we will have the following names: 'server_1.log', 'server_2.log', 'server_3.log'.

This means that the file with the highest number will always be the most recent, while 'server_1.log' will always be the oldest one.

This is how I broke down the problem:

- When the server is started, it will write to the most recent log file that it finds inside the given directory:
  - If the given directory doesn't exist, it is created and the first log file 'server_1.log' is created;
  - If the directory exists but it is empty, the first log file 'server_1.log' is created;
  - If the directory is not empty, the most recent file is searched.

  This is done by the function findMostRecentFile().
- When the most recent file is found, before writing into it, we have to check if its size will exceed the threshold (THRESHOLD). To do that I wrote a function that checks if the sum between the number of bytes we want to write and the current size of the file we want to write on is greater than the threshold.

  This is done by the function sizeExceedThreshold().
- If the sum is less or equal, we just write on the most recent file we found.
- If the sum is greater (the returned value by previous function is 1), this means that we have to create a new log file. However, if the directory already contains the

maximum possible number of log files, we have to remove the oldest log file and create a new one. Otherwise, as there is space in the directory, we just create a new log file.

- To remove the oldest file and create a new one maintaining the nomenclature (1, 2, 3, ...) I wrote a function called rotateLogs() that is only executed when the directory contains MAXLOGFILE log files. It removes the oldest one ('server_1.log'), renames the logs in the proper order (e.g., _3 becomes _2, _2 becomes _1, _1 was deleted before), and finally creates a new log file named 'server_MAXLOGFILE.log'.

This logs rotation mechanism is implemented in the source file "logsRotation.c". To check its correctness I suggest the following test:

1. Run the executable giving the name of a non-existing directory (e.g., logs):
   /.logsRotation logs
   It will create the directory and the first log file inside it.

2. Repeat several times the previous command: you will see that the first log file will be populated by log messages ('test').

3. After 4 log messages ('test' is 4 bytes + the new line character = 5 bytes) we reach the default threshold (20 bytes = 4 x 5) so a new log file is created.

4. Continuing to repeat always the same command you will see that after 4 log files (MAXLOGFILE is set to 4 by default) no other files will be created but every time the oldest file is removed, the logs are rotated and the most recent file will be empty (contains only one line).