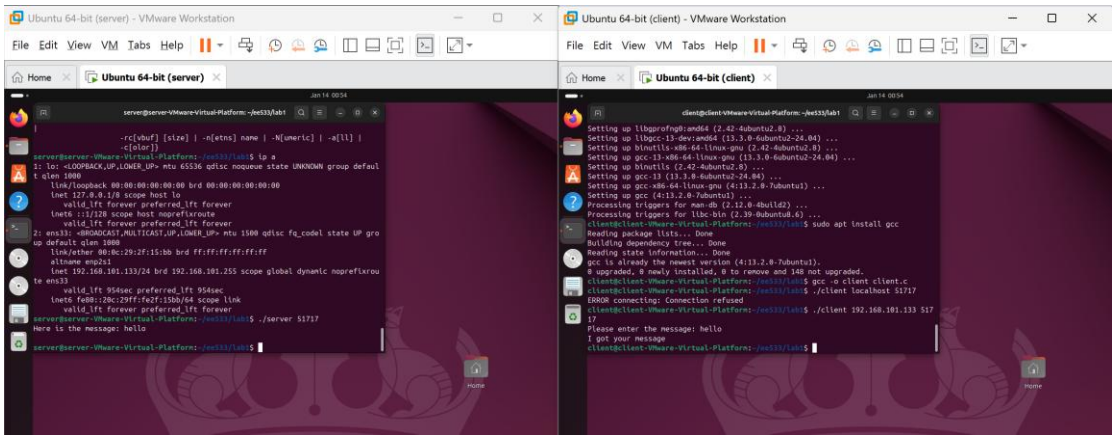
Lab1 Report

YaYi Lin

Github: https://github.com/yayi1213/EE533_lab1

Original Server



The original server program can handle only **one client connection**.

After accepting a connection and performing a single read/write operation, the server terminates.

Such a design is not practical in real-world applications, since a real server should run indefinitely and be able to handle multiple clients simultaneously.

Fork-Based Concurrent Server

To allow the server to handle multiple clients concurrently, the server calls fork() after each successful accept().

The fork() system call creates a child process.

The child process handles communication with the connected client, while the parent process continues waiting for new connections.

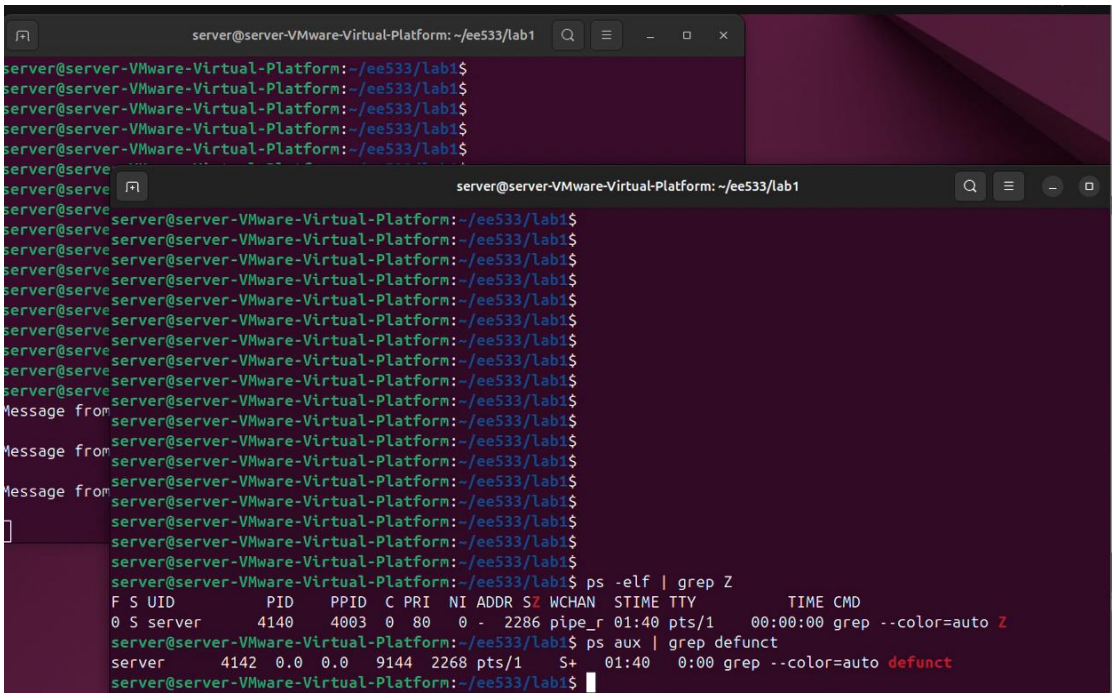
Zombie problem

When a child process terminates and the parent process does not call wait(), the child becomes a zombie process.

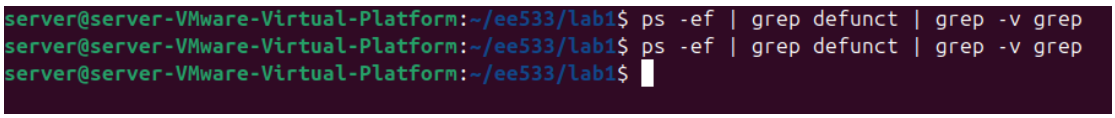
Zombie processes occupy entries in the process table and may eventually exhaust system resources.

Common solutions include:

- Ignoring the SIGCHLD signal
- Or using a signal handler that calls wait() with the WNOHANG option



The output of ps aux | grep defunct and ps -elf | grep Z only shows the grep command itself, indicating that no defunct (zombie) processes exist. This confirms that the server correctly reaps child processes after termination.



Zombie processes were verified using ps -ef | grep defunct | grep -v grep.
No zombie processes were observed because a SIGCHLD handler with waitpid(WNOHANG) was installed to properly reap terminated child processes.

No output from ps -ef | grep defunct | grep -v grep means there are no zombie processes.

Code Analysis – Fork-based Concurrent TCP Server

1. Overall Architecture

This program implements a fork-based concurrent TCP server. The server listens on a specified port and creates a new child process using fork() for each incoming client connection. Each child process independently handles communication with its assigned client, while the parent process continues to accept new connections.

2. Socket Creation and Initialization

The server begins by creating a TCP socket using socket (AF_INET, SOCK_STREAM, 0). The sockaddr_in structure is then initialized to specify IPv4, the chosen port number, and INADDR_ANY, which allows the server to accept connections on any local network interface.

3. Binding and Listening

The bind() system call associates the socket with a specific port on the local machine. After binding, the listen() call places the socket into a passive listening state, allowing the server to queue incoming connection requests.

4. Accepting Client Connections

The accept() system call blocks until a client initiates a connection. When a connection is accepted, a new socket descriptor (newsockfd) is created specifically for communication with that client.

5. Fork-based Concurrency

After accepting a connection, the server calls fork(). The child process handles client

communication by calling the dostuff() function, while the parent process immediately closes the connected socket and returns to accept new clients. This design allows multiple clients to be served concurrently.

6. Client Communication (dostuff Function)

The dostuff() function is responsible for handling communication with a single client. It reads data sent by the client using read(), prints the received message, and sends a response back using write(). This function executes only in the child process.

7. Zombie Process Prevention

To prevent zombie processes, the server installs a SIGCHLD signal handler. When a child process terminates, the SIGCHLD signal is delivered to the parent, which calls waitpid() with the WNOHANG option to reap all terminated child processes without blocking.

8. Resource Management

Proper file descriptor management is critical. The child process closes the listening socket since it is not needed, while the parent process closes the connected socket. This ensures no file descriptor leaks occur.