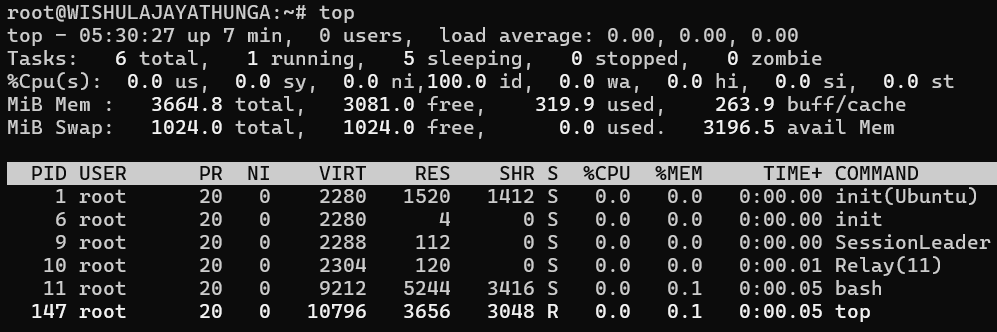
CO327 – Lab 01

# E/19/166 Jayathunga W.W.K.

1. **Processes**

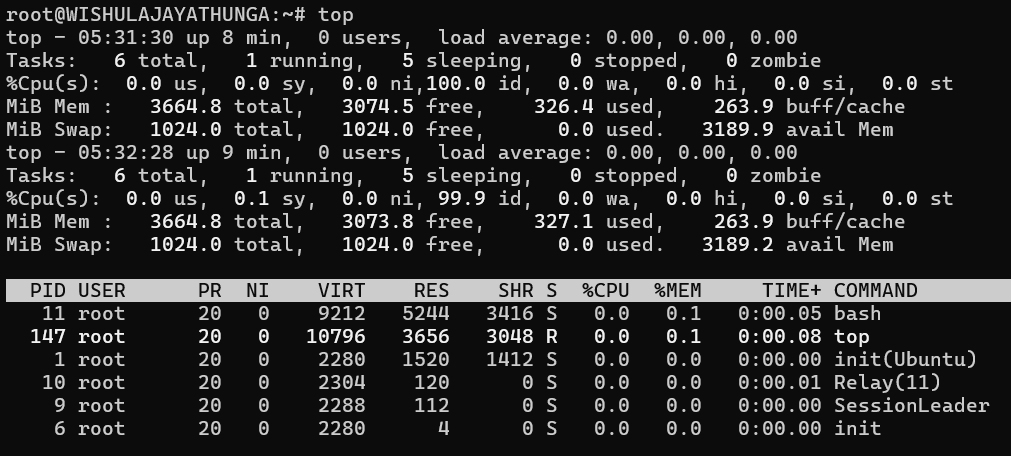
* **Exercise 1**

1. Sorted by CPU usage



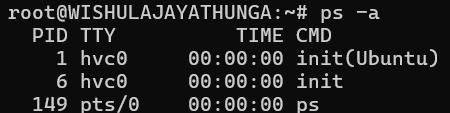
Sorted by memory usage

* To sort the processes by memory usage, while the “top” is running, we can press “Shift + M”



1. “ps -a” command

* Display information about all the active processes.
* “-a” option stands for “all”
* This shows the processes for all users on the system, not just those of the current user.



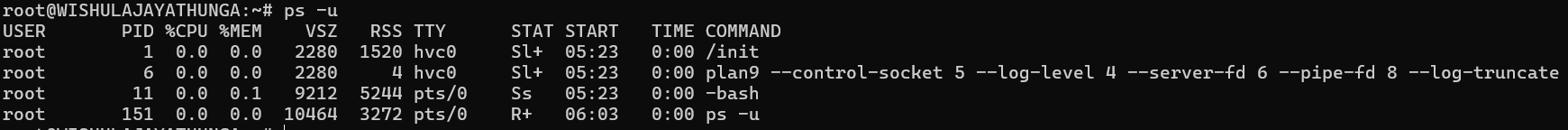
“ps -x” command

* Display information about all the processes.
* The “-x” optiong stands for “all”, but it also includes processes that do not have a controlling terminal, such as daemon processes.

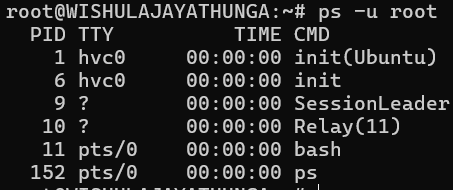


“ps -u” command

* Display information about processes started by a specific user.
* The “-u” option stands for “user”.

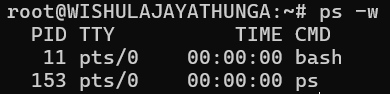


* It shows the processes started by the user specified after the “-u” option.



“ps -w” command

* Display information about the currently running processes.
* The “-w” option stands for “wide”.
* It is used to provide a wide output, which means it will display the full width of the output, including all text.

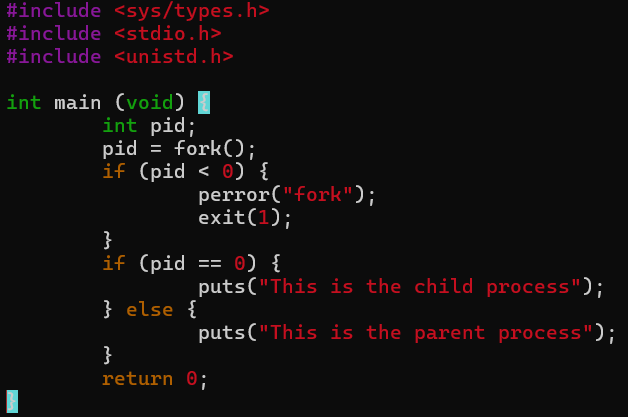


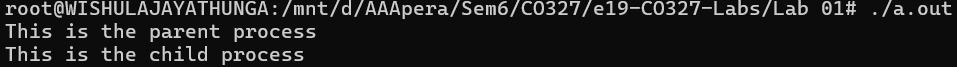
Name of the process with PID 1

* We can use “ps -p 1 -o comm=” command
* Process = init(Ubuntu)



* 1. **Creating a new process**



****

**There is no getcpid() call. Why?**

In Unix-like operating systems, including Linux, the fork() system call is used to create a new process. When a process calls fork(), it creates a new process known as the child process. The fork() system call returns a value that allows us to distinguish between the parent process and the child process:

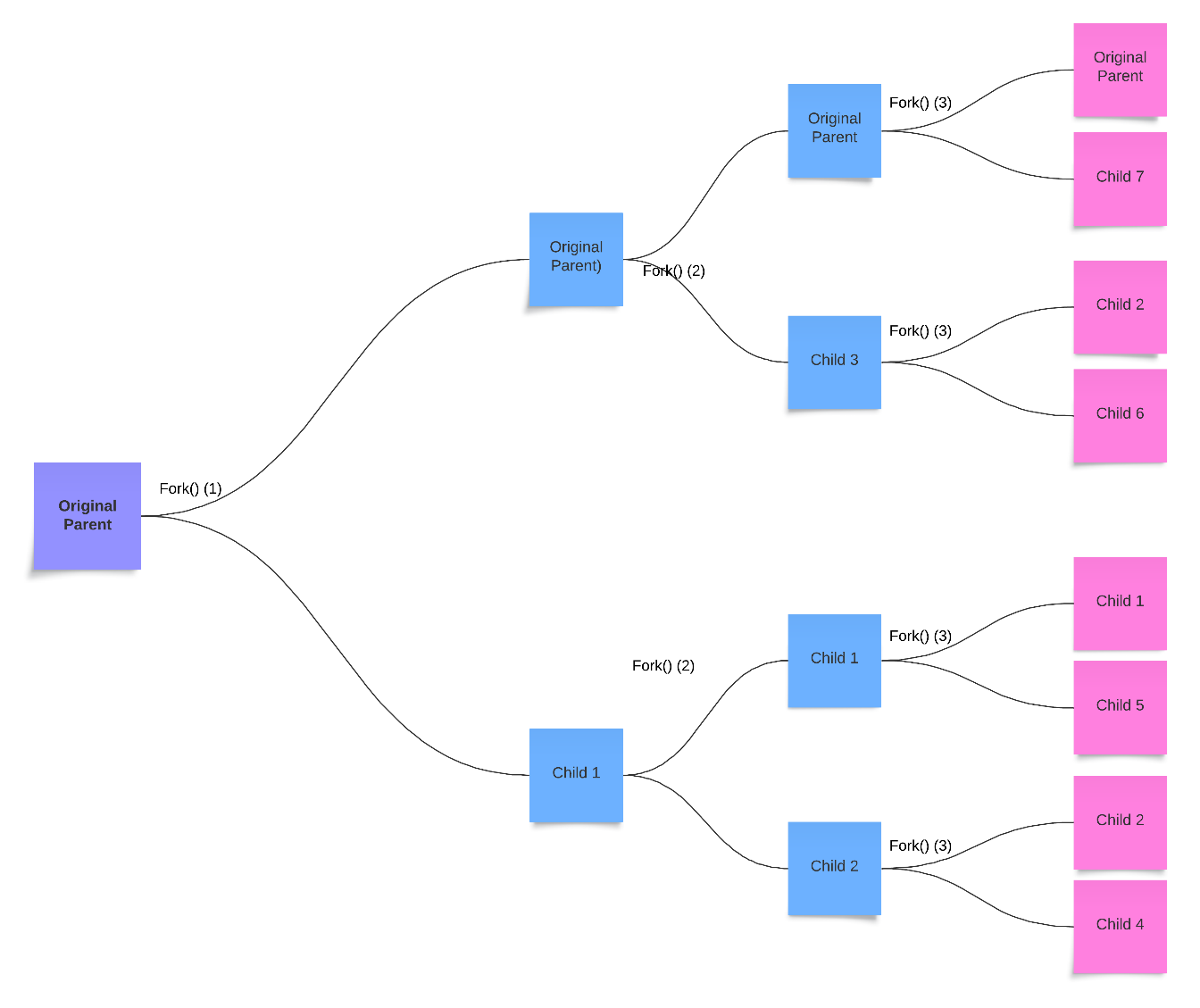
* + If fork() returns a zero, it means the code is being executed by the child process.
  + If fork() returns a positive value, it means the code is being executed by the parent process, and the value returned is the PID of the child process.

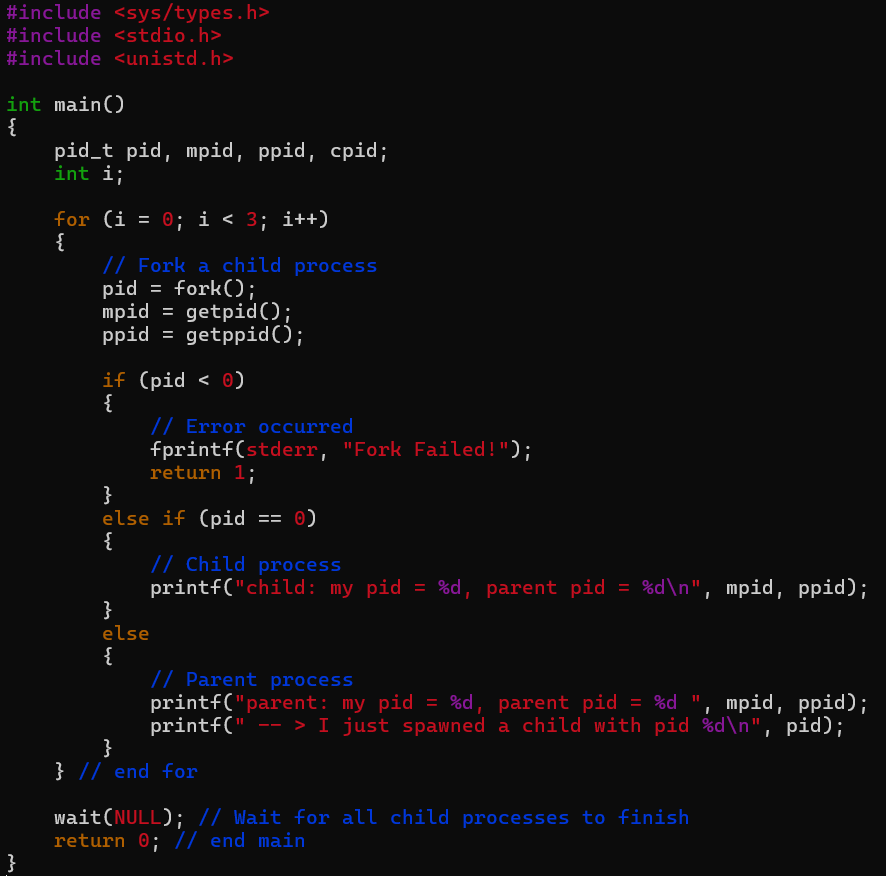
This design makes a getcpid() function unnecessary because the parent process can always know the PID of its child processes from the return value of fork(). There’s no need for a separate system call to get the child PID.

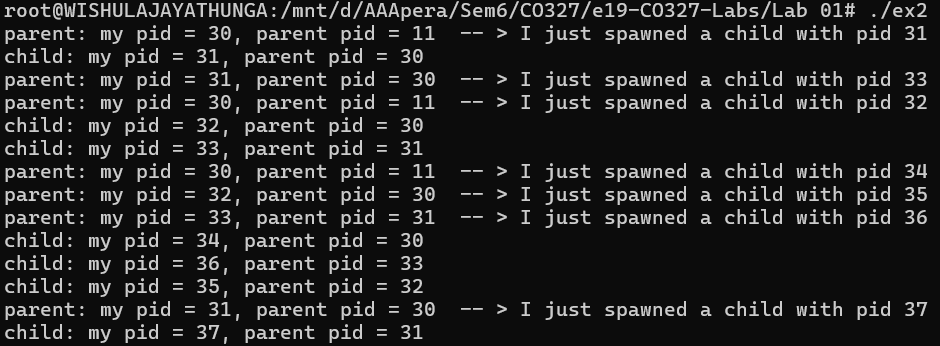
Moreover, a process in Unix-like systems can have multiple child processes, so a getcpid() function without arguments wouldn’t make much sense. If a process needs to keep track of its child processes, it must store the PIDs returned by fork() in its own data structures.

* **Exercise 2**

1. The order in which messages from the parent and child processes are printed can vary because processes in Linux are scheduled independently by the operating system’s scheduler. The order is not guaranteed to be the same every time, as it depends on various factors such as system load and process priorities. In concurrent programming, unless there is explicit synchronization, the execution order of processes or threads is non-deterministic.
2. 7 children will be spawn (8 processes including the original parent)



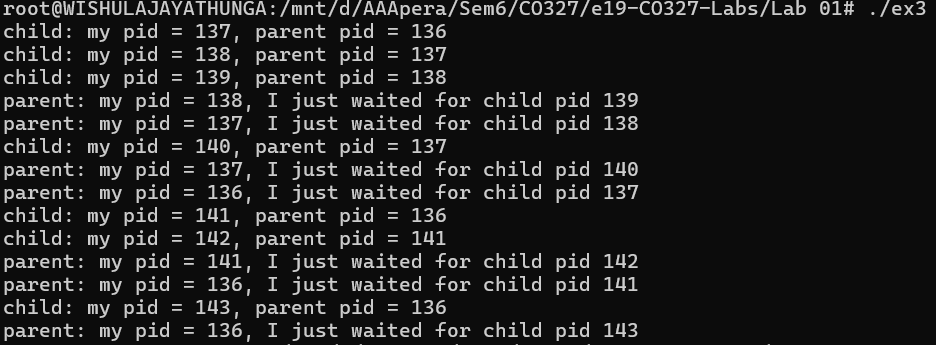




* 1. **Waiting for Children**

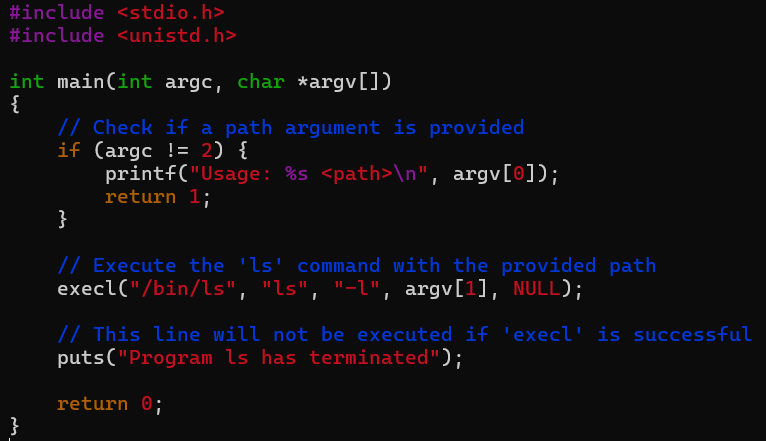
**Exercise 3**

****

****

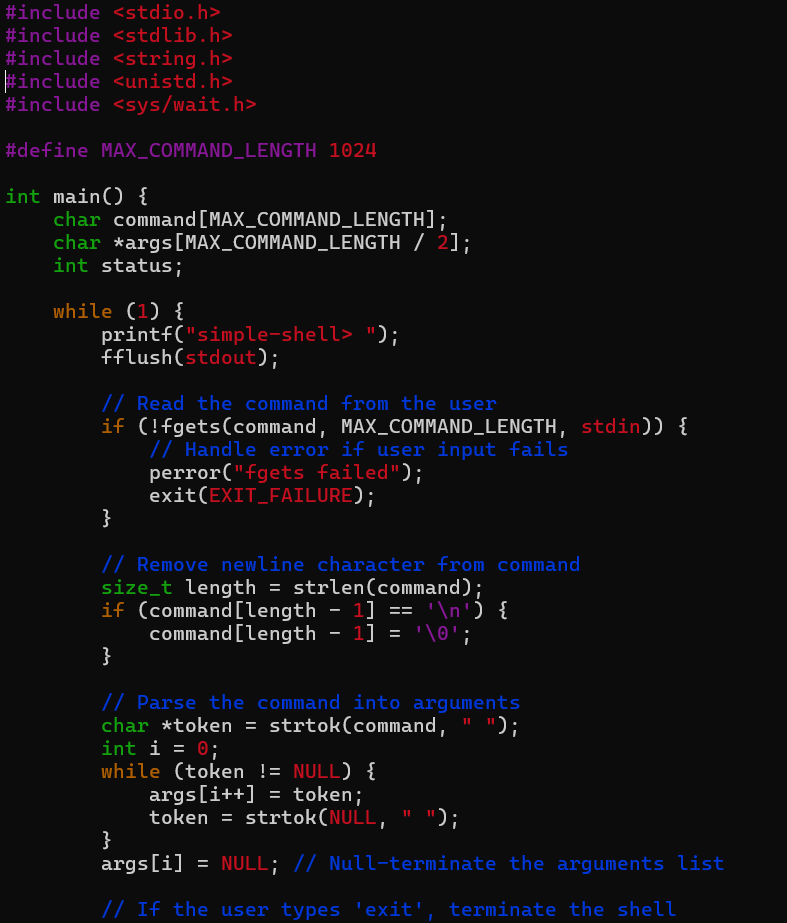
* 1. **Replacing the process image**

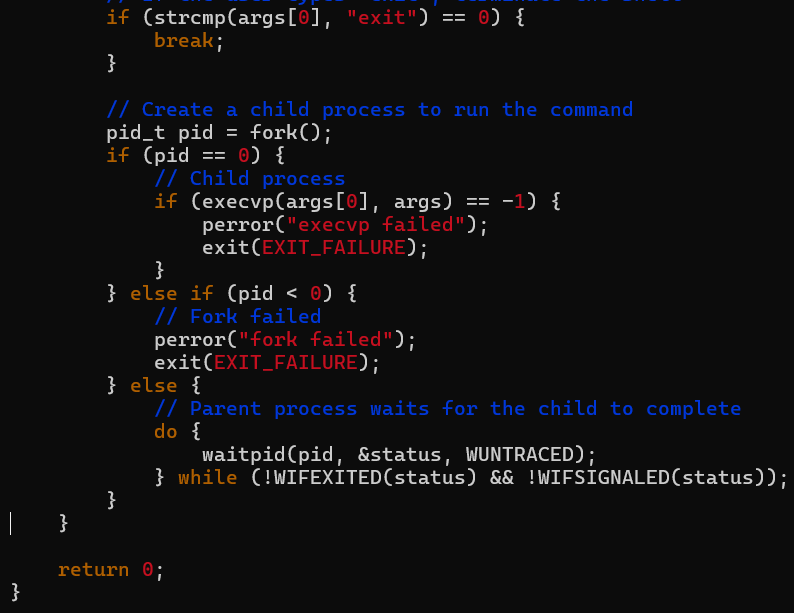
**Exercise 4:**

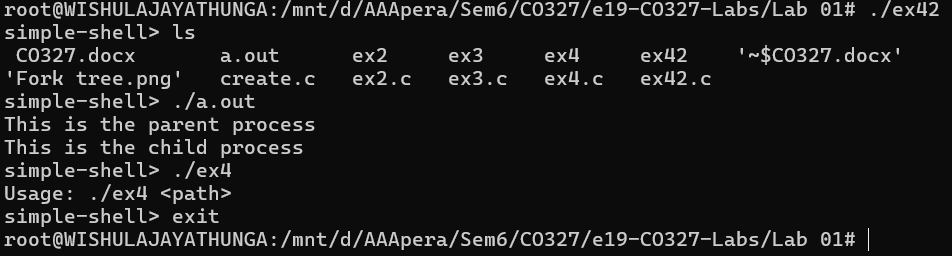
****

****

The message “Program l has terminated” is printed zero times. This is because once execl() is called, the current program (which includes the puts() statement) is replaced by the /bin/ls program. Since execl() does not return unless there’s an error, the puts() statement is never reached, and the message is not printed.

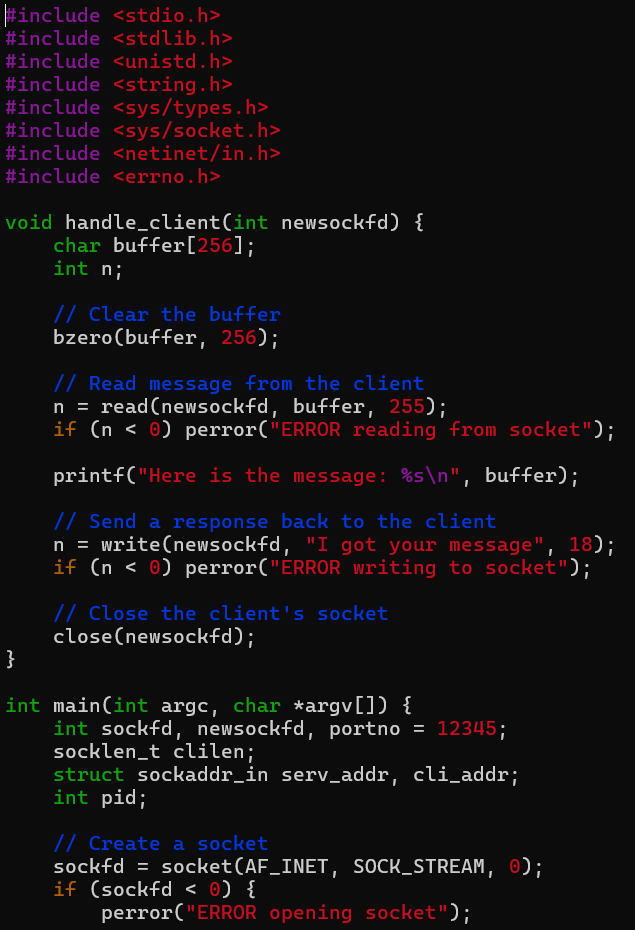


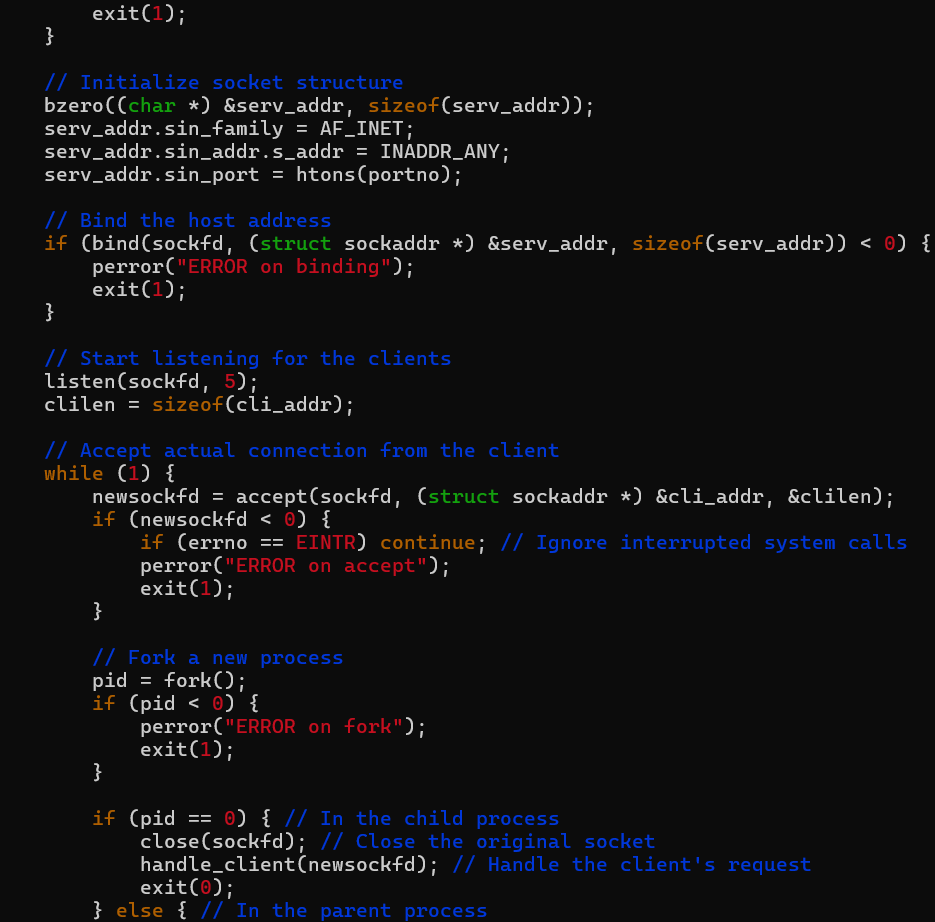


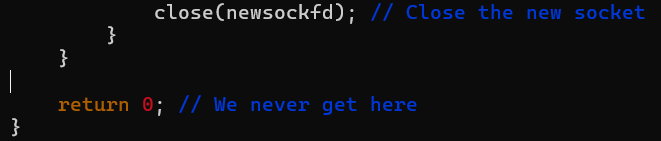


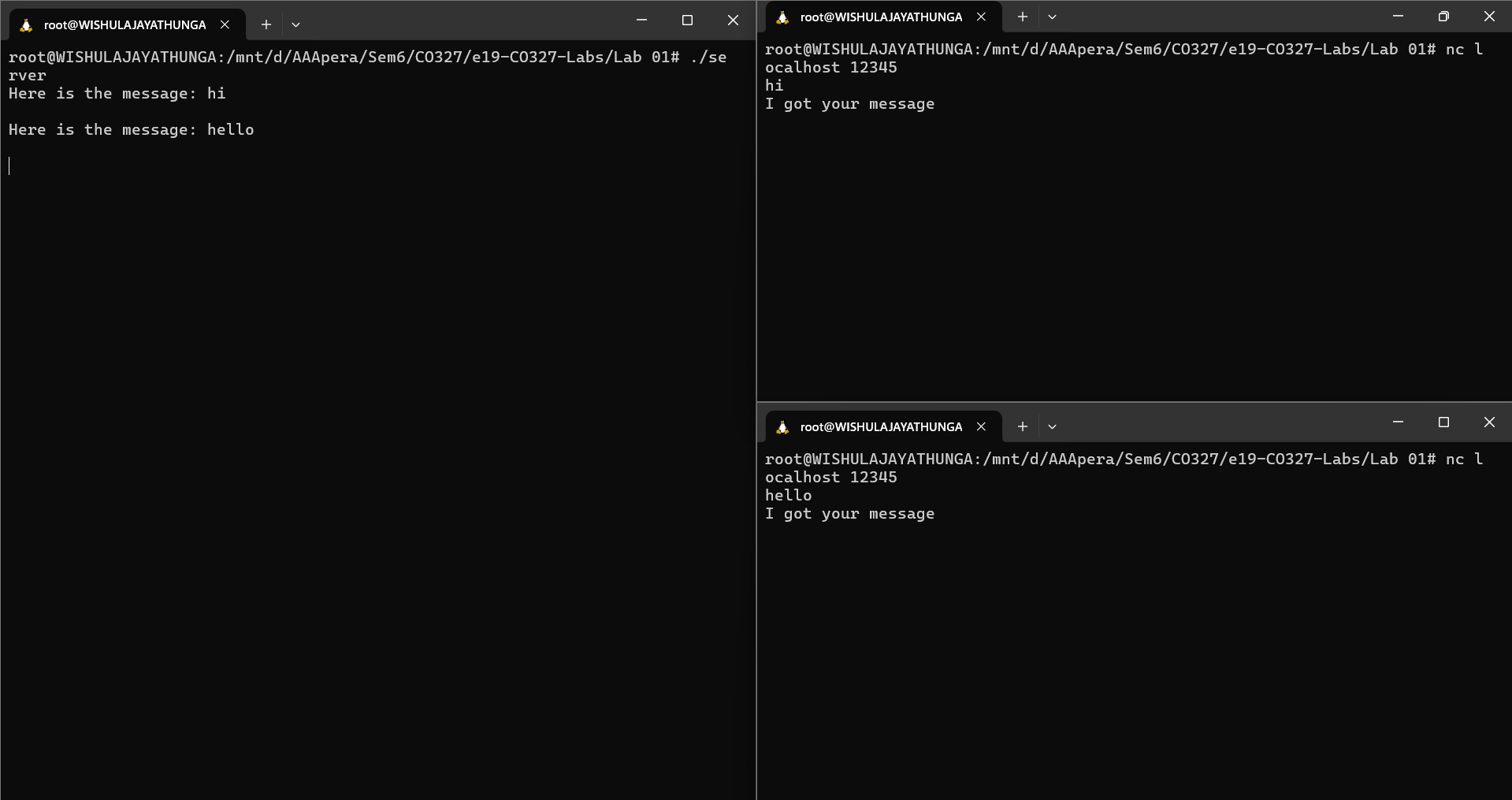
1. **Multiprocess Servers**

**Exercise 5**

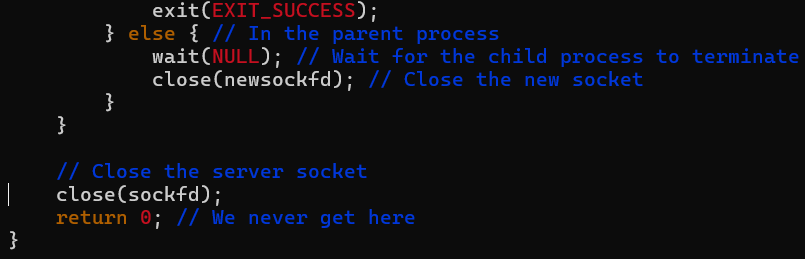
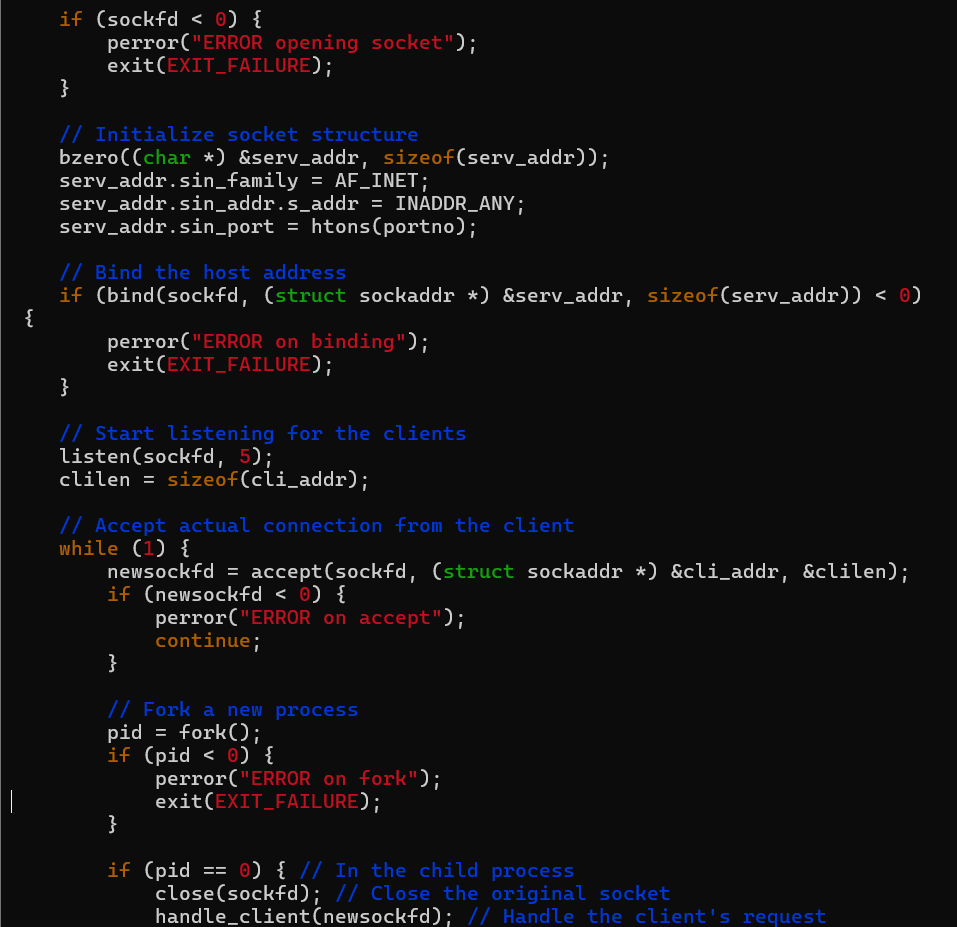
****

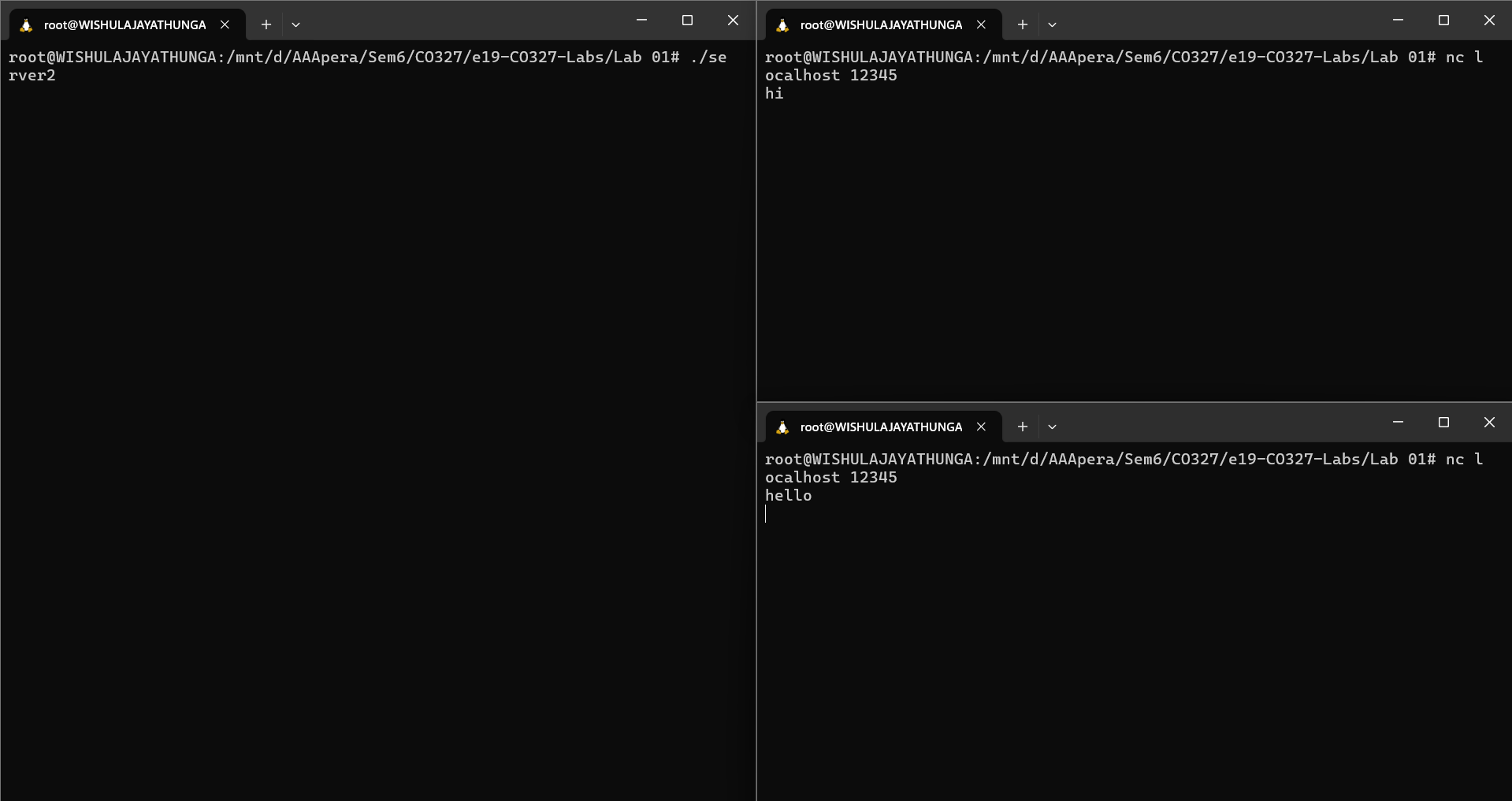
****

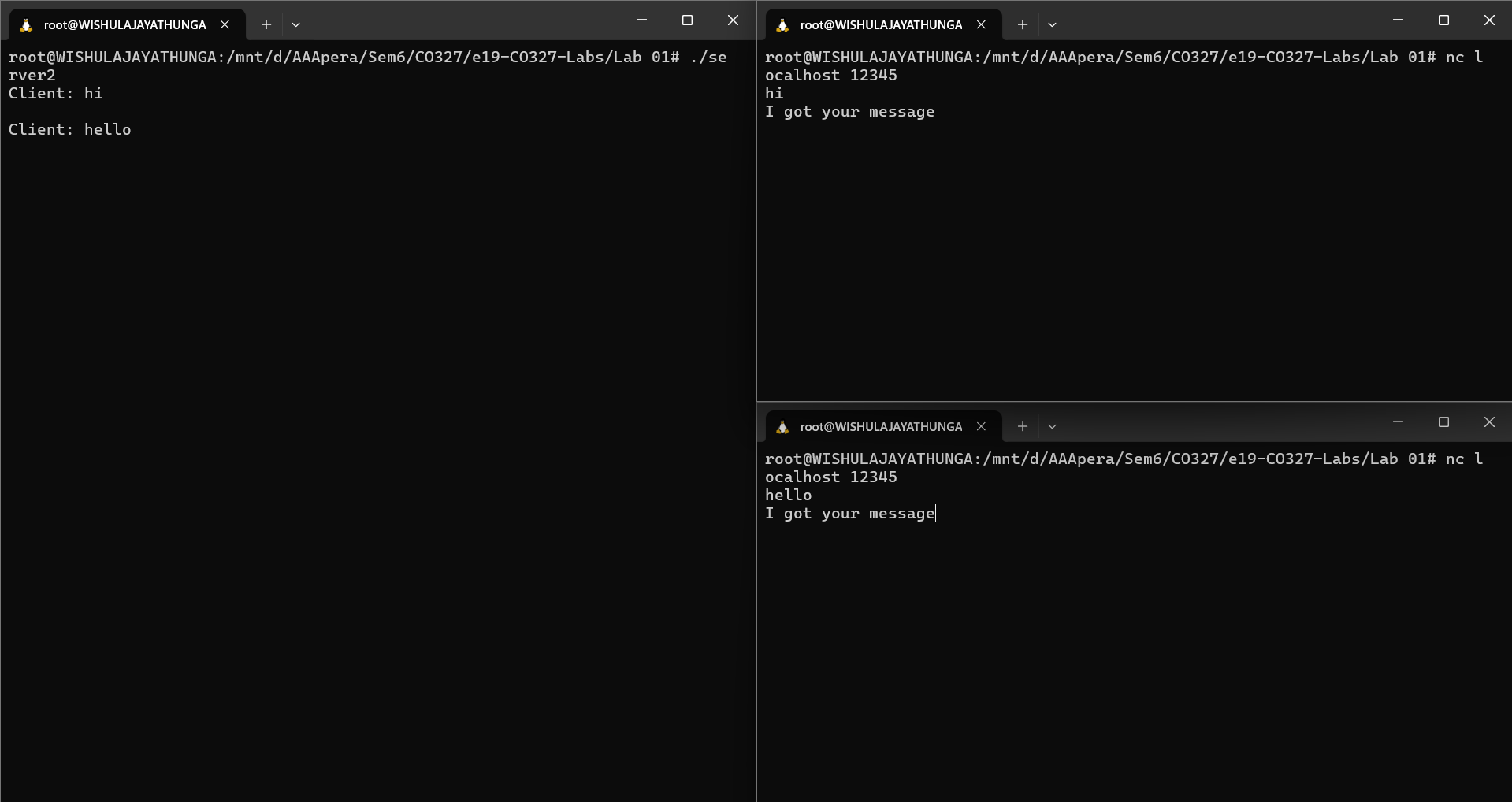




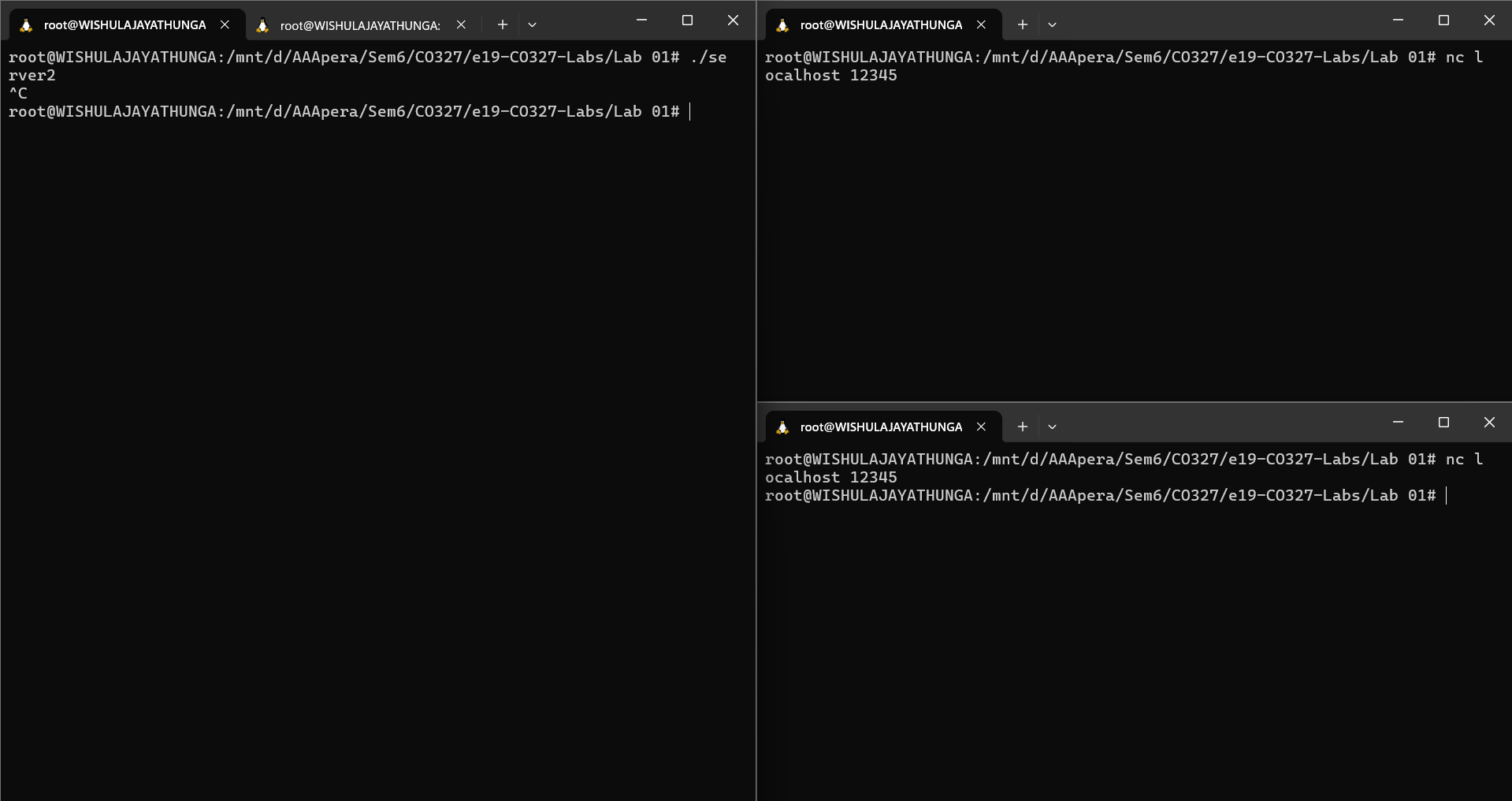


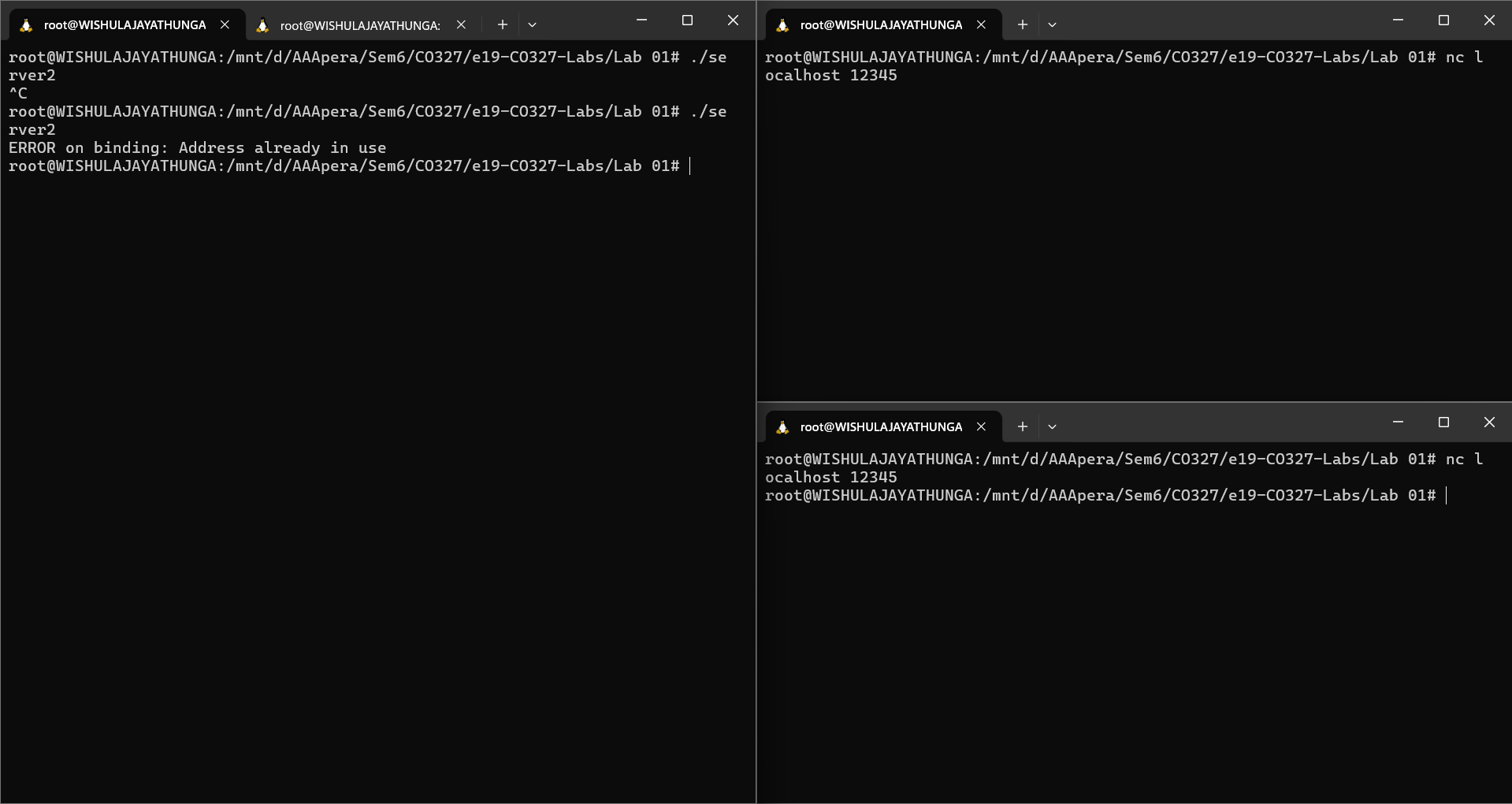






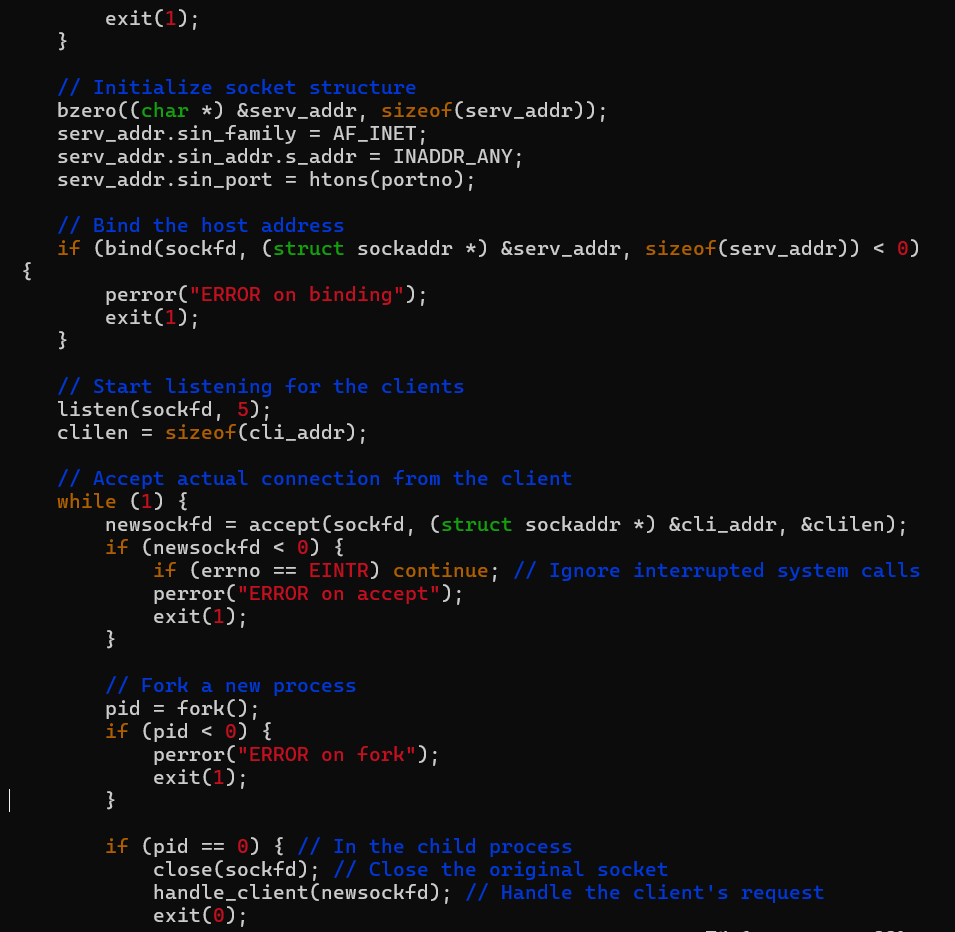
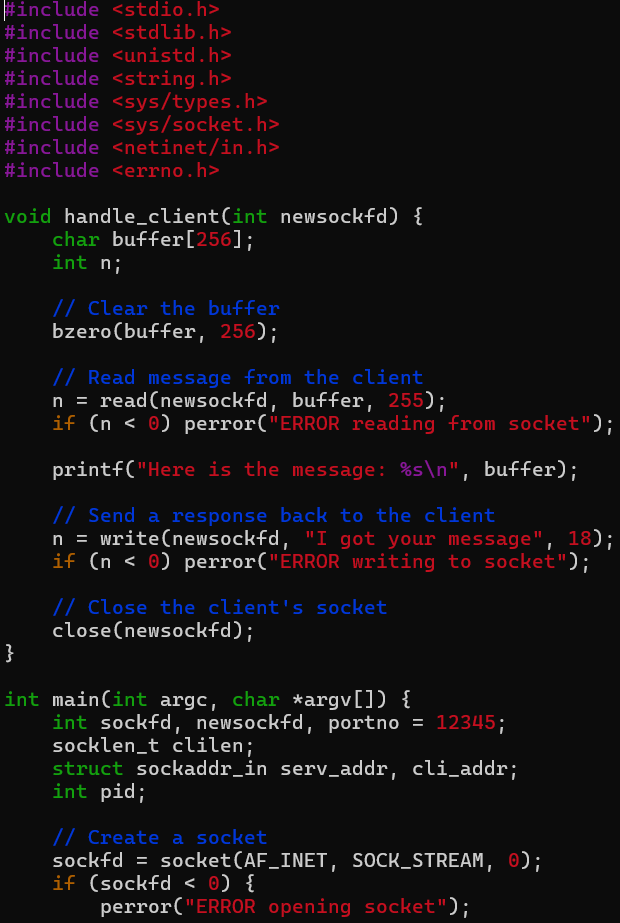
The server parent process calls wait() to wait until the child serving a client terminates, the server would handle one client at a time sequentially. It would not accept new connections until the current child process finishes, leading to a non-concurrent, single-client-at-a-time service model.

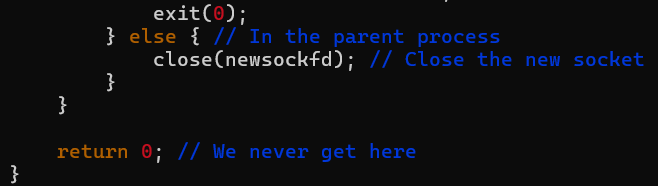




When we terminate a TCP server while a client is connected, the client experiences a sudden loss of connection and any ongoing data transfer will be interrupted. If we try to restart the server immediately, we encounter an issue where the server’s socket is still in the TIME\_WAIT state, which prevents the server from binding to the same port right away.

To resolve this issue, we can implement a signal handler in your server code that catches termination signals (such as SIGINT for Ctrl+C) and ensures that the server closes all open sockets properly before shutting down. Additionally, we can set the SO\_REUSEADDR socket option, which allows the server to bind to the port even if it is still in the TIME\_WAIT state.





**Verify that your new server can handle multiple concurrent connections by using nc()**

To handle multiple concurrent connections, the server must implement a mechanism to track and manage multiple client sockets. This can be achieved using one system call, which allows the server to monitor multiple file descriptors (sockets) to see if any of them is ready for reading, writing, or if an error occurred.

**Can two concurrent clients request the same file?**

Yes, two concurrent clients can request the same file. The server can handle this by creating a separate process or thread to deal with each client request. Each process or thread reads the requested file and sends its contents back to the requesting client. Since file reading is typically a non-destructive operation, multiple processes or threads can read the same file concurrently without any issues.