# CO327 : Operating Systems

# Lab 01 : Multiprocessing

E/20/420 WANASINGHE J.K.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[CO327 : Operating Systems 1](#__RefHeading___Toc1010_2112274691)

[Lab 01 : Multiprocessing 1](#__RefHeading___Toc1012_2112274691)

[1. Processes 1](#__RefHeading___Toc1014_2112274691)

[Exercise 1: 1](#__RefHeading___Toc1016_2112274691)

[1.1. Creating a new process 3](#__RefHeading___Toc1018_2112274691)

[Exercise 2: 3](#__RefHeading___Toc1020_2112274691)

[1.2. Waiting for children 5](#__RefHeading___Toc1022_2112274691)

[Exercise 3: 5](#__RefHeading___Toc1024_2112274691)

[1.3. Replacing the process image 6](#__RefHeading___Toc1026_2112274691)

[Exercise 4: 7](#__RefHeading___Toc1028_2112274691)

[2. Multiprocess servers 9](#__RefHeading___Toc1030_2112274691)

[Exercise 5: 10](#__RefHeading___Toc1032_2112274691)

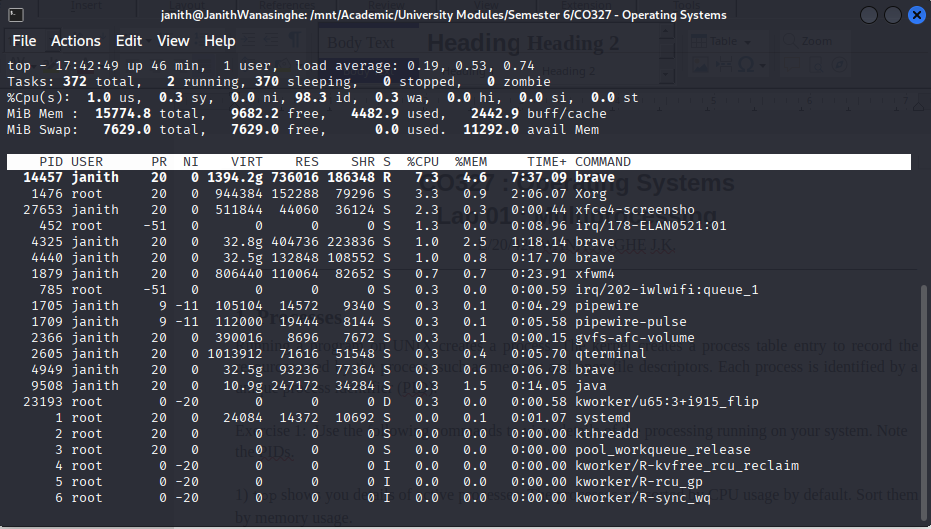
## 1. Processes

Running a program on UNIX creates a process. The kernel creates a process table entry to record the resources used by the process, such as memory and open file descriptors. Each process is identified by a unique process identifier (PID).

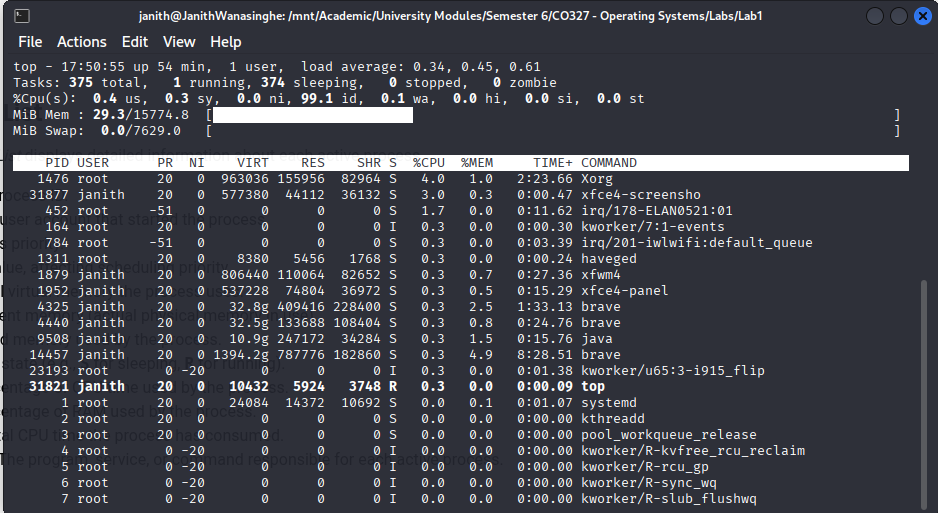
#### Exercise 1:

**Use the following commands to view details of the processing running on your system. Note the PIDs.**

1) top shows you details of active processes. The processes are sorted by CPU usage by default. Sort them by memory usage.

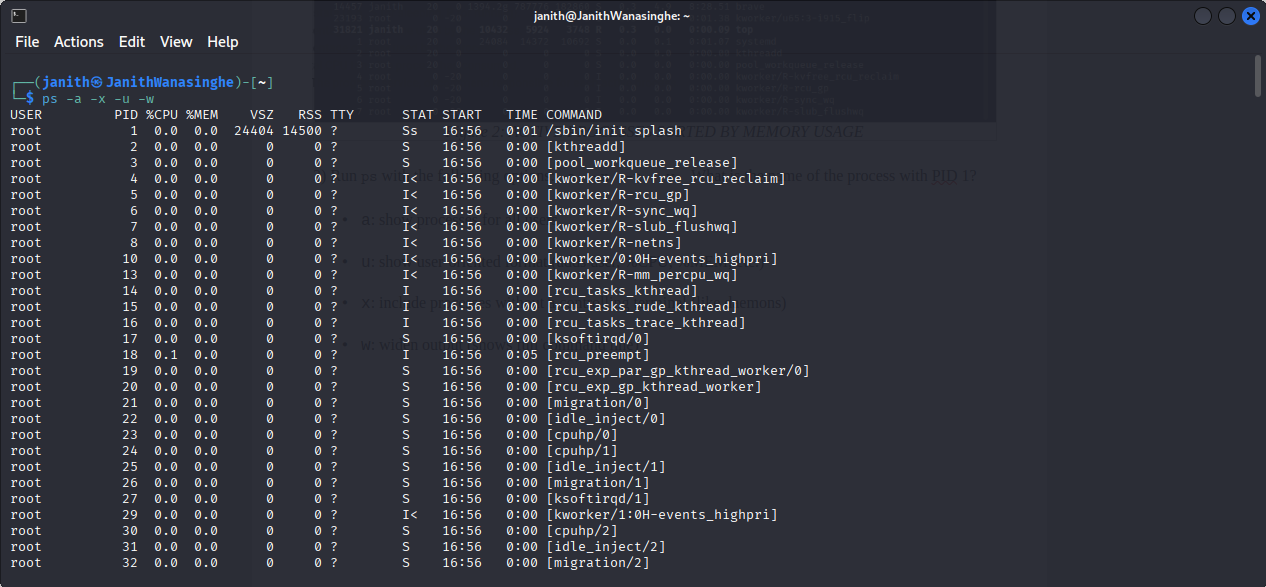
Figure 1: ACTIVE PROCESSES SORTED BY CPU USAGE

When Sorted by Memory Usage using Hotkey ‘M’ in keyboard

Figure 2: ACTIVE PROCESSES SORTED BY MEMORY USAGE

2) Run ps with the following options: -a, -x, -u, -w. What is the name of the process with PID 1?

* a: show processes for all users
* u: show user-oriented format (adds user, %CPU, %MEM, etc.)
* x: include processes without a controlling terminal (like daemons)
* w: widen output (shows full command line)

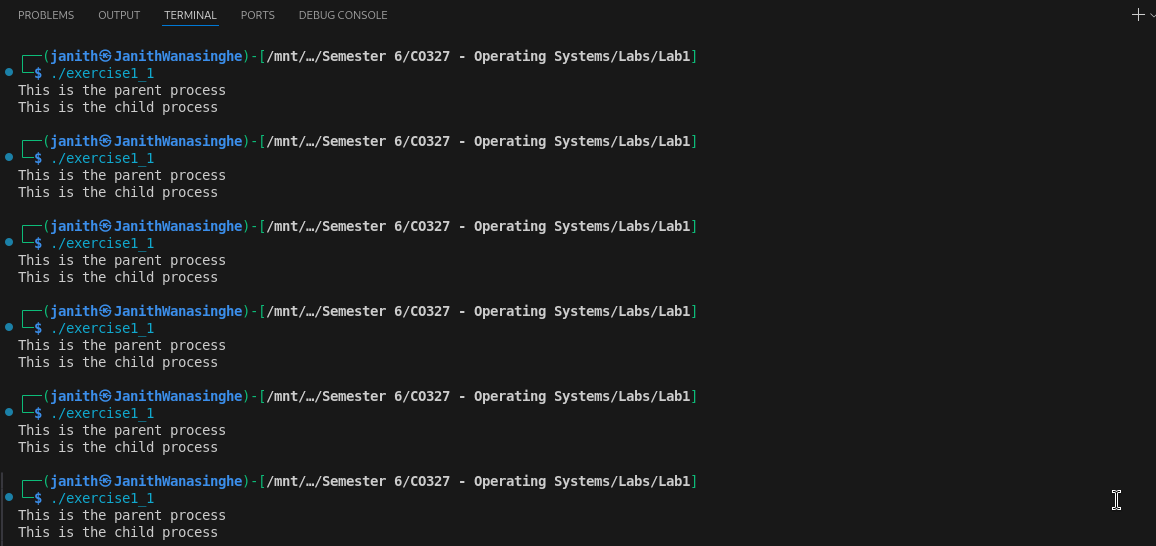
Figure 3: WHEN PS COMMAND IS RUN WITH GIVEN FLAGS

/sbin/init is the process with PID = 1

### 1.1. Creating a new process

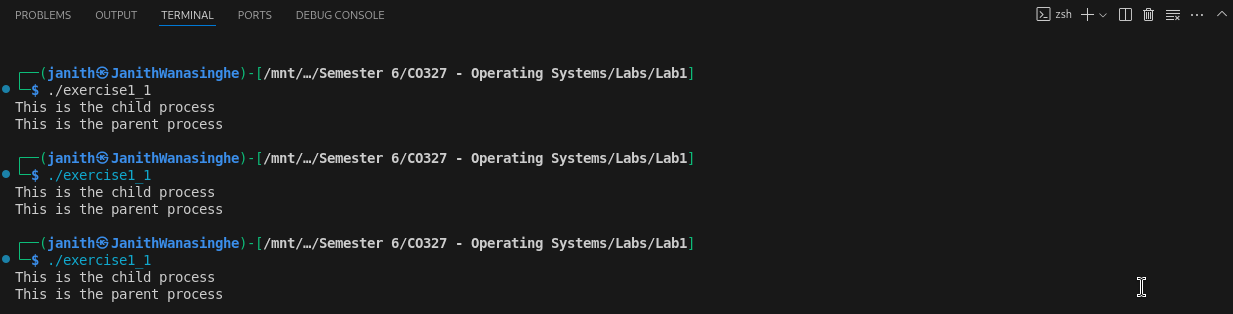
#### Exercise 2:

1. In what order are the messages from parent and child printed? Is the order always the same?

Figure 4: PRINT ORDER WHEN THE PROGRAM IS RUN SEVERAL TIMES

The order of messages from the parent and child processes is **not guaranteed**. It depends on the operating system scheduler. The parent might print first, or the child might. The behavior can vary between systems or even between runs.

When a delay is added towards the parent, following was observed.

Figure 5: DELAYED PARENT

2. How many children will the following program spawn? Draw a diagram illustrating the parent-child relationships between processes.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void)

{

for (int i = 0; i < 3; i++)

fork();

}

We can track this like a tree:

Iteration 0: i = 0

- fork() is called once

- Now we have 2 processes

Iteration 1: i = 1

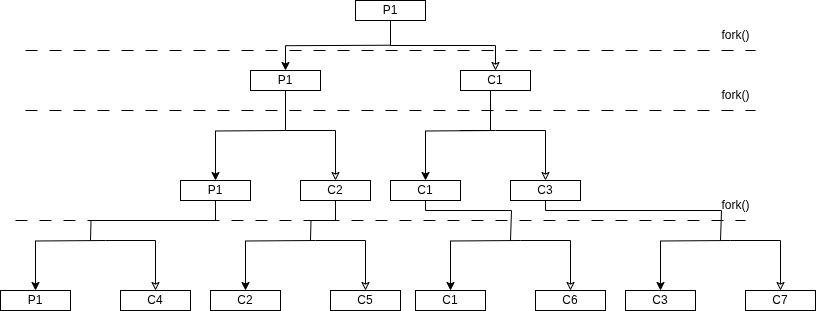
- Both of those 2 processes call fork() again → each makes 1 child

- Now we have 4 processes total

Iteration 2: i = 2

- All 4 processes call fork() again → each makes 1 child

- Now we have 8 processes total

Figure 6: 3 FORK ITERATIONS

### 1.2. Waiting for children

The system call wait() lets the parent process wait until the child process has exited. For example, a shell must wait until a command a user has run completes before prompting the user for the next command.

#### Exercise 3:

Modify the program in section 1.1 so that the parent always prints its message after the child. Refer to man 2 wait for details.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

int main(void)

{

int pid;

pid = fork();

if (pid < 0)

{

perror("fork");

exit(1);

}

if (pid == 0)

{

sleep(1); // delay child

puts("This is the child process");

}

else

{

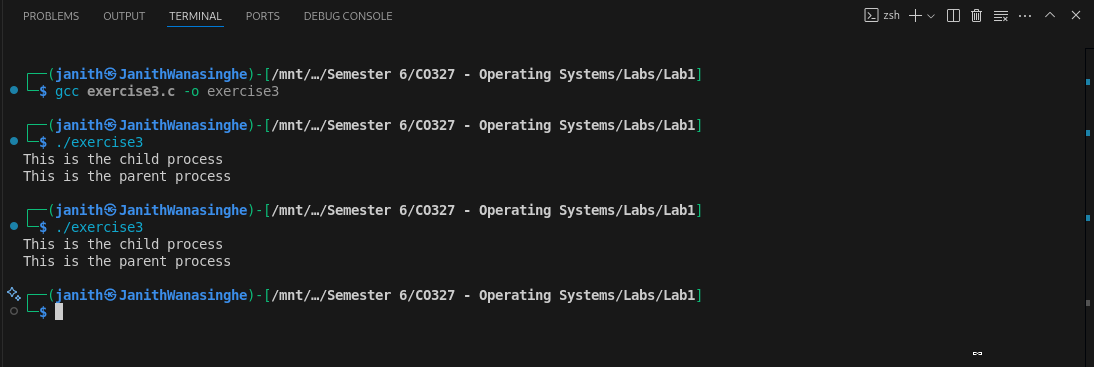
wait(NULL); // wait for child to finish

puts("This is the parent process");

}

return 0;

}

Figure 7: IMPLEMENTING WAIT()

### 1.3. Replacing the process image

In certain cases we would like to execute another program within a process. For example, a shell must create a new process and then run an external program within that process. This is made possible by the exec() system call. Doing an exec replaces the current process image in memory with a new program. Therefore a call to exec does not return.

#include <unistd.h>

#include <stdio.h>

int main(char argc, char \*\*argv)

{

execl("/bin/ls", "-l", argv[1], NULL);

puts("Program ls has terminated");

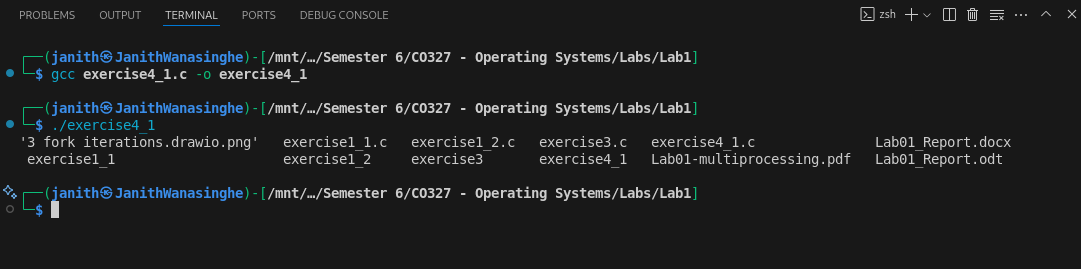
}

This example is using the execl() variation provided by the standard library. See man 3 exec for details.

#### Exercise 4:

1. Compile and run the above code giving it a path as an argument. How many times is the message “Program ls has terminated” printed?

“Program ls has terminated” does not print any time as the program is never returned to the next line.

Figure 8: EXEC() IS NEVER RETURNED

2. Write a very simple shell that repeatedly prompts the user for a command and runs it with any arguments given. Make sure your shell waits until the command has completed before prompting the user for the next command.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#define MAX\_CMD\_LEN 1024 // Maximum length of command input

#define MAX\_ARGS 64 // Maximum number of arguments in a command

int main()

{

char command[MAX\_CMD\_LEN]; // Buffer to hold the command input

char \*args[MAX\_ARGS]; // Array to hold the command arguments

while (1)

{

printf("WANASINGHE\_SHELL> "); // Print the shell prompt

fflush(stdout); // Ensure the prompt is printed immediately

if (fgets(command, MAX\_CMD\_LEN, stdin) == NULL) // Read command from standard input

break; //Break if EOF is encountered (EOF means Ctrl+D in terminal)

// Remove newline

command[strcspn(command, "\n")] = '\0'; // Remove trailing newline character by replacing it with null terminator

// Tokenize command into arguments

int i = 0; // Index for arguments

char \*token = strtok(command, " "); // Split command by spaces

while (token != NULL && i < MAX\_ARGS - 1) // Ensure we don't exceed the maximum number of arguments

{

args[i++] = token; // Store the token in the args array

token = strtok(NULL, " "); // Get the next token

}

args[i] = NULL; // Null-terminate the args array to indicate the end of arguments

if (args[0] == NULL) // If no command was entered, continue to the next iteration

continue;

if (strcmp(args[0], "exit") == 0) // Check if the command is "exit"

break;

pid\_t pid = fork(); // Create a new process

if (pid == 0) // Child process

{

execvp(args[0], args); //This line will make the child process execute the command

// If execvp returns, it means there was an error

perror("exec failed");

exit(1);

}

else

{

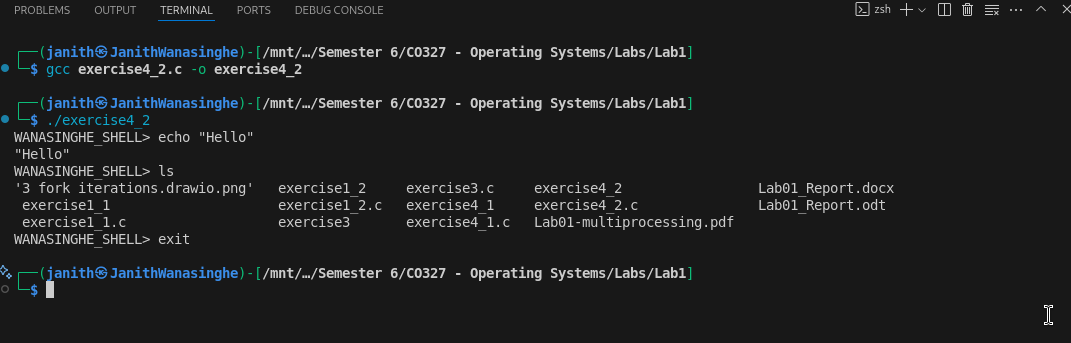
wait(NULL);

}

}

return 0;

}

Figure 9: SIMPLE SHELL PROCESSES

## 2. Multi process servers

We can now apply these techniques to build servers that concurrently handle multiple client requests using multiple server processes. A socket is set up to listen() for client connection requests in the same way as iterative servers. When a new client request arrives accept() returns a new socket connected to the client. The main loop of a multi process server is where the difference lies. Instead of handling the request itself, the server spawns a child process to handle the client while the parent process continues to listen() for new connections. In this way the server is able to handle multiple clients concurrently.

A multi process server handles multiple clients **concurrently** by forking child processes.

**Key Concepts:**

* listen() prepares the server to accept incoming client connections.
* accept() waits for and returns a new socket descriptor for a client.
* fork() creates a **new child process** to handle the client.
* The **child** handles the client with handle\_client(newsockfd) and then exits.
* The **parent** closes the newsockfd and loops to listen for the next client.

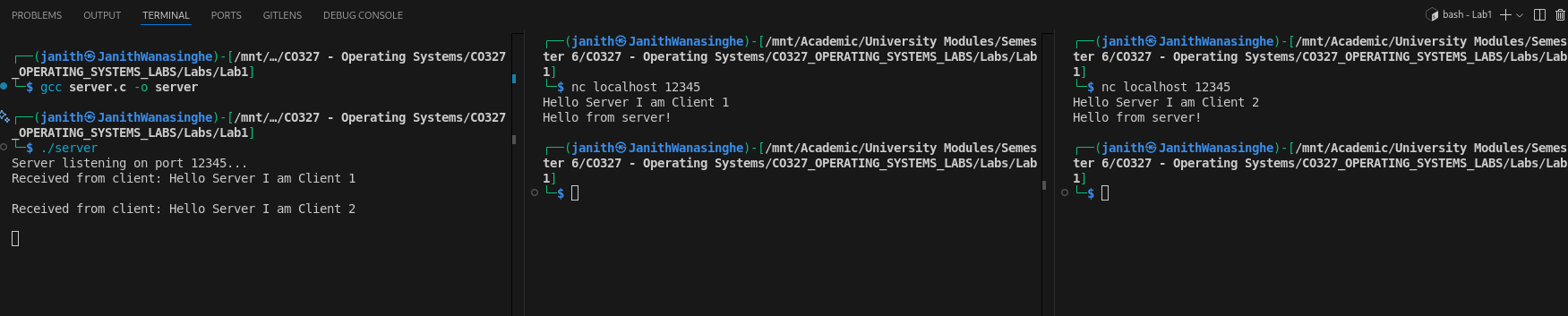
**Why Fork?**

If the server handled each client in the same process, it would **block** on one client and ignore others. Forking allows each client to be served in a **separate process**, so they don’t interfere with each other.

#### Exercise 5:

1. Open three terminals and run the server in one. Use nc() to connect as two clients concurrently on port 12345. Type some text in both clients and examine the client and server outputs.

To simulate, long process of time a sleep time of 10 was artificially implemented in the code so that the other terminal could be worked on when program is being run.

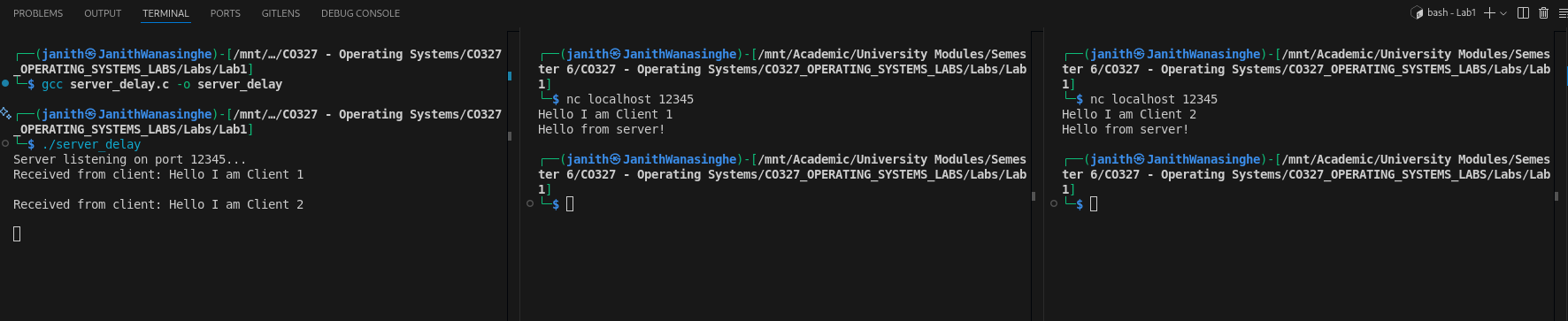
Figure 10: SERVER HANDLES BOTH CLIENTS AT THE SAME TIME

2. Suppose we modify the server parent process to call wait() on the last line above (highlited) to wait until the child serving a client terminates. What would happen?

The **parent process blocks** after handling one client.

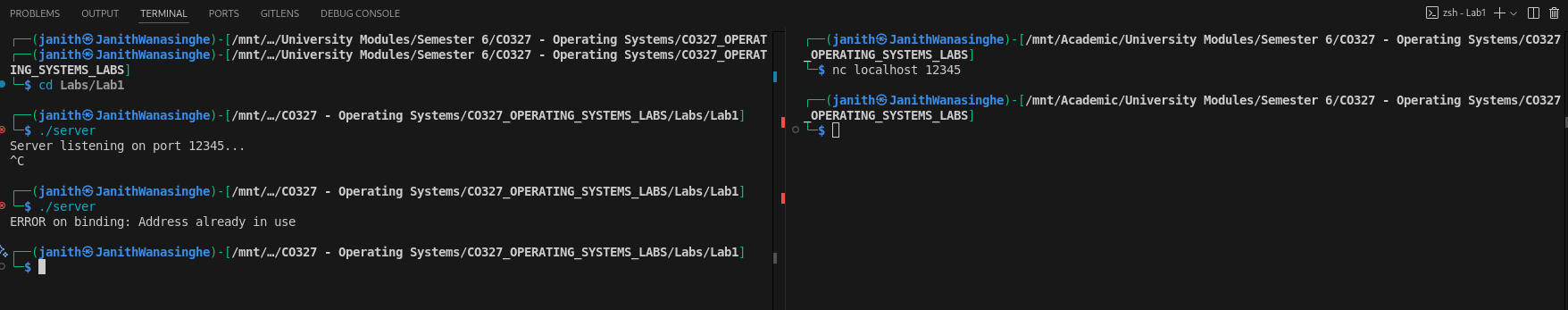
* It **won’t accept any new clients** until the child finishes.
* Therefore, **no concurrent connections** — it behaves **like an iterative server**, not a concurrent one.

Takes a longer time to execute as the responses are sequential.

Figure 11: SERVER HANDLES BOTH CLIENTS SEQUENTIALLY

3. What happens if you terminate the the server while a client is connected, and then try to restart it? (Resolving this issue requires a signal handler.)

The following error appears in the terminal, when the server is terminated and restarted.

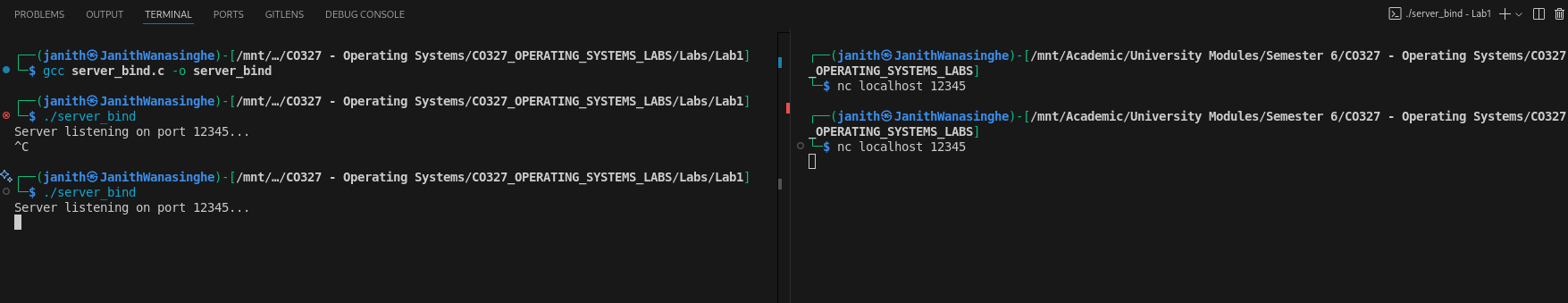
Figure 12: WHEN THE SERVER IS TERMINATED AND RESTARTED DURING CLIENT CONNECTION

**How to resolve it:**

Use the socket option SO\_REUSEADDR before bind():

int opt = 1;

setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt)); //This will allow the socket to be reused immediately after the program exits

Figure 13: IMMEDIATE RESTART POSSIBLE WITH RESUSEADDR

int opt = 1;

- Declares an integer variable opt and sets its value to 1.

- This variable is used to enable (1 for true/on) the socket option.

setsockopt(sockfd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

setsockopt is a system call used to set options for sockets.

Arguments:

sockfd: The file descriptor of the socket you want to configure.

SOL\_SOCKET: This specifies that the option is at the socket level (not protocol-specific).

SO\_REUSEADDR: This is the option that is being set. It allows the socket to bind to an address that is in a TIME\_WAIT state.

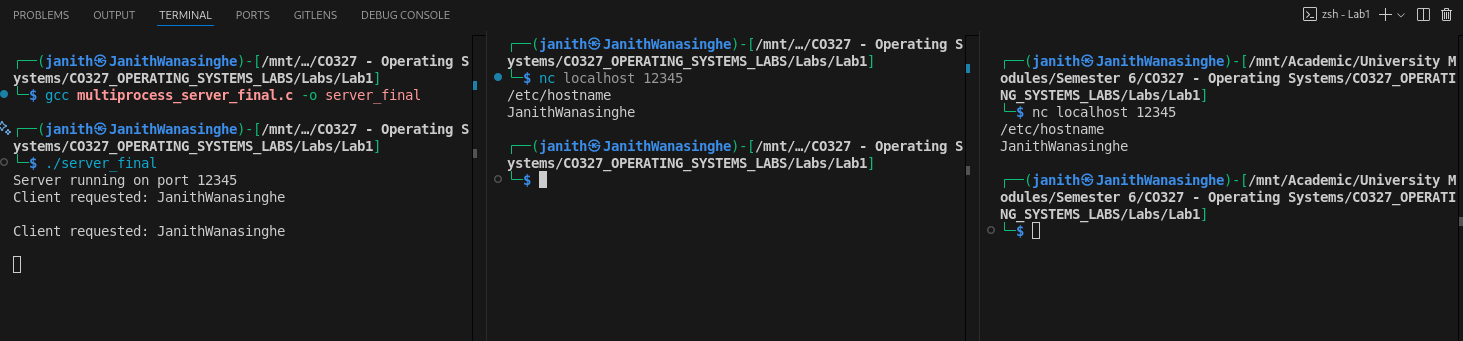
&opt: A pointer to the value that want to be set for the option (1 in this case, to enable).

sizeof(opt): The size of the option value.

Summary:

This code tells the OS to allow immediate reuse of the port after the program exits, by setting the SO\_REUSEADDR option on the socket.

4. Modify this server to do the following: The client sends the path to a file whose contents the server will send back to the client (if the file exists.) Verify that your new server can handle multiple concurrent connections by using nc(). Can two concurrent clients request the same file?

Figure 14: WHEN TWO CLIENTS REQUEST THE SAME THING

YES — and the current code **already supports it**.

Each client request is handled by its **own process**. So:

* Client 1 sends /etc/hostname
* Client 2 also sends /etc/hostname
* Two separate FILE \*fp = fopen(...) calls happen in **independent address spaces**

No interference. Both clients get the file contents independently.