CO327 – Lab 01

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

I. Processes

- **Exercise 1**

I. Sorted by CU usage

```
root@WISHULAJAYATHUNGA:~# top
top - 05:30:27 up 7 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:   6 total,   1 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3664.8 total,   3081.0 free,    319.9 used,    263.9 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   3196.5 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0    2280   1520   1412 S   0.0   0.0   0:00.00 init(Ubuntu)
    6 root      20   0    2280      4      0 S   0.0   0.0   0:00.00 init
    9 root      20   0    2288    112      0 S   0.0   0.0   0:00.00 SessionLeader
   10 root      20   0    2304    120      0 S   0.0   0.0   0:00.01 Relay(11)
   11 root      20   0    9212   5244   3416 S   0.0   0.1   0:00.05 bash
  147 root      20   0   10796   3656   3048 R   0.0   0.1   0:00.05 top
```

Sorted by memory usage

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

```
root@WISHULAJAYATHUNGA:~# top
top - 05:31:30 up 8 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:   6 total,   1 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3664.8 total,   3074.5 free,    326.4 used,    263.9 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   3189.9 avail Mem
top - 05:32:28 up 9 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:   6 total,   1 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3664.8 total,   3073.8 free,    327.1 used,    263.9 buff/cache
MiB Swap:   1024.0 total,   1024.0 free,      0.0 used.   3189.2 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   11 root      20   0    9212   5244   3416 S   0.0   0.1   0:00.05 bash
  147 root      20   0   10796   3656   3048 R   0.0   0.1   0:00.08 top
    1 root      20   0    2280   1520   1412 S   0.0   0.0   0:00.00 init(Ubuntu)
   10 root      20   0    2304    120      0 S   0.0   0.0   0:00.01 Relay(11)
    9 root      20   0    2288    112      0 S   0.0   0.0   0:00.00 SessionLeader
    6 root      20   0    2280      4      0 S   0.0   0.0   0:00.00 init
```

II. “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.

```
root@WISHULAJAYATHUNGA:~# ps -a
  PID TTY          TIME CMD
    1 hvc0     00:00:00 init(Ubuntu)
    6 hvc0     00:00:00 init
  149 pts/0    00:00:00 ps
```

“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.

```
root@WISHULAJAYATHUNGA:~# ps -x
  PID TTY      STAT   TIME COMMAND
    1 hvc0     Sl+    0:00 /init
    6 hvc0     Sl+    0:00 plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
    9 ?        Ss     0:00 /init
   10 ?        R      0:00 /init
   11 pts/0    Ss     0:00 -bash
  150 pts/0    R+     0:00 ps -x
```

“ps -u” command

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

```
root@WISHULAJAYATHUNGA:~# ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2280  1520 hvc0     Sl+  05:23   0:00 /init
root         6  0.0  0.0   2280     4 hvc0     Sl+  05:23   0:00 plan9 --control-socket 5 --log-level 4 --server-fd 6 --pipe-fd 8 --log-truncate
root        11  0.0  0.1   9212  5244 pts/0    Ss   05:23   0:00 -bash
root       151  0.0  0.0  10464  3272 pts/0    R+   06:03   0:00 ps -u
```

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

```
root@WISHULAJAYATHUNGA:~# ps -u root
  PID TTY          TIME CMD
    1 hvc0     00:00:00 init(Ubuntu)
    6 hvc0     00:00:00 init
    9 ?        00:00:00 SessionLeader
   10 ?        00:00:00 Relay(11)
   11 pts/0    00:00:00 bash
  152 pts/0    00:00:00 ps
```

“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.

```
root@WISHULAJAYATHUNGA:~# ps -w
  PID TTY          TIME CMD
   11 pts/0    00:00:00 bash
  153 pts/0    00:00:00 ps
```

Name of the process with PID 1

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

```
root@WISHULAJAYATHUNGA:~# ps -p 1 -o comm=
init(Ubuntu)
```

1.1 Creating a new process

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main (void) {
        int pid;
        pid = fork();
        if (pid < 0) {
                perror("fork");
                exit(1);
        }
        if (pid == 0) {
                puts("This is the child process");
        } else {
                puts("This is the parent process");
        }
        return 0;
}
```

```
root@WISHULAJAYATHUNGA:/mnt/d/AAApera/Sem6/CO327/e19-CO327-Labs/Lab 01# ./a.out
This is the parent process
This is the child process
```

There is no getcpid() call. Why?

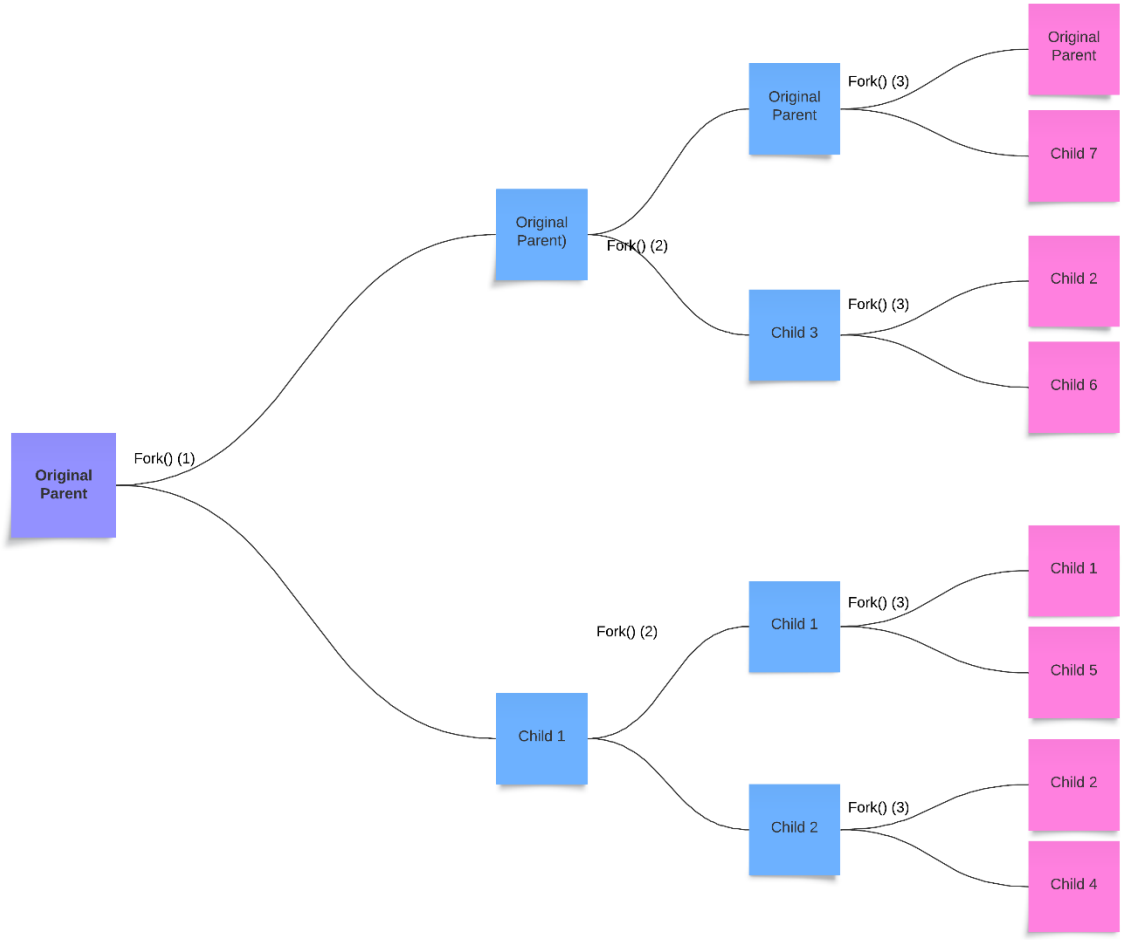
In Unix-like operating systems, including Linux, the fork() system call is used to create a new process. When a process calls fork(), it creates a new process known as the child process. The fork() system call returns a value that allows us to distinguish between the parent process and the child process:

- 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**
  I. 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.
  II. 7 children will be spawn (8 processes including the original parent)



```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid, mpid, ppid, cpid;
    int i;

    for (i = 0; i < 3; i++)
    {
        // Fork a child process
        pid = fork();
        mpid = getpid();
        ppid = getppid();

        if (pid < 0)
        {
            // Error occurred
            fprintf(stderr, "Fork Failed!");
            return 1;
        }
        else if (pid == 0)
        {
            // Child process
            printf("child: my pid = %d, parent pid = %d\n", mpid, ppid);
        }
        else
        {
            // Parent process
            printf("parent: my pid = %d, parent pid = %d ", mpid, ppid);
            printf(" -- > I just spawned a child with pid %d\n", pid);
        }
    } // end for

    wait(NULL); // Wait for all child processes to finish
    return 0; // end main
}
```

```
root@WISHULAJAYATHUNGA:/mnt/d/AAApera/Sem6/CO327/e19-CO327-Labs/Lab 01# ./ex2
parent: my pid = 30, parent pid = 11  -- > I just spawned a child with pid 31
child: my pid = 31, parent pid = 30
parent: my pid = 31, parent pid = 30  -- > I just spawned a child with pid 33
parent: my pid = 30, parent pid = 11  -- > I just spawned a child with pid 32
child: my pid = 32, parent pid = 30
child: my pid = 33, parent pid = 31
parent: my pid = 30, parent pid = 11  -- > I just spawned a child with pid 34
parent: my pid = 32, parent pid = 30  -- > I just spawned a child with pid 35
parent: my pid = 33, parent pid = 31  -- > I just spawned a child with pid 36
child: my pid = 34, parent pid = 30
child: my pid = 36, parent pid = 33
child: my pid = 35, parent pid = 32
parent: my pid = 31, parent pid = 30  -- > I just spawned a child with pid 37
child: my pid = 37, parent pid = 31
```

1.2 Waiting for Children

Exercise 3

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h> // Include for wait()

int main()
{
    pid_t pid;
    int i;

    for (i = 0; i < 3; i++)
    {
        // Fork a child process
        pid = fork();

        if (pid < 0)
        {
            // Error occurred
            fprintf(stderr, "Fork Failed!");
            return 1;
        }
        else if (pid == 0)
        {
            // Child process
            printf("child: my pid = %d, parent pid = %d\n", getpid(), getppi
d());
            //return 0; // Child exits
        }
        else
        {
            // Parent process
            wait(NULL); // Wait for child to exit
            printf("parent: my pid = %d, I just waited for child pid %d\n",
getpid(), pid);
        }
    }
    return 0; // end main
}
```

```
root@WISHULAJAYATHUNGA:/mnt/d/AAApera/Sem6/CO327/e19-CO327-Labs/Lab 01# ./ex3
child: my pid = 137, parent pid = 136
child: my pid = 138, parent pid = 137
child: my pid = 139, parent pid = 138
parent: my pid = 138, I just waited for child pid 139
parent: my pid = 137, I just waited for child pid 138
child: my pid = 140, parent pid = 137
parent: my pid = 137, I just waited for child pid 140
parent: my pid = 136, I just waited for child pid 137
child: my pid = 141, parent pid = 136
child: my pid = 142, parent pid = 141
parent: my pid = 141, I just waited for child pid 142
parent: my pid = 136, I just waited for child pid 141
child: my pid = 143, parent pid = 136
parent: my pid = 136, I just waited for child pid 143
```

1.3 Replacing the process image

Exercise 4:

1.

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    // Check if a path argument is provided
    if (argc != 2) {
        printf("Usage: %s <path>\n", argv[0]);
        return 1;
    }

    // Execute the 'ls' command with the provided path
    execl("/bin/ls", "ls", "-l", argv[1], NULL);

    // This line will not be executed if 'execl' is successful
    puts("Program ls has terminated");

    return 0;
}
```

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.

2.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_COMMAND_LENGTH 1024

int main() {
    char command[MAX_COMMAND_LENGTH];
    char *args[MAX_COMMAND_LENGTH / 2];
    int status;

    while (1) {
        printf("simple-shell> ");
        fflush(stdout);

        // Read the command from the user
        if (!fgets(command, MAX_COMMAND_LENGTH, stdin)) {
            // Handle error if user input fails
            perror("fgets failed");
            exit(EXIT_FAILURE);
        }

        // Remove newline character from command
        size_t length = strlen(command);
        if (command[length - 1] == '\n') {
            command[length - 1] = '\0';
        }

        // Parse the command into arguments
        char *token = strtok(command, " ");
        int i = 0;
        while (token != NULL) {
            args[i++] = token;
            token = strtok(NULL, " ");
        }
        args[i] = NULL; // Null-terminate the arguments list

        // If the user types 'exit', terminate the shell
```

```
        if (strcmp(args[0], "exit") == 0) {
            break;
        }

        // Create a child process to run the command
        pid_t pid = fork();
        if (pid == 0) {
            // Child process
            if (execvp(args[0], args) == -1) {
                perror("execvp failed");
                exit(EXIT_FAILURE);
            }
        } else if (pid < 0) {
            // Fork failed
            perror("fork failed");
            exit(EXIT_FAILURE);
        } else {
            // Parent process waits for the child to complete
            do {
                waitpid(pid, &status, WUNTRACED);
            } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        }
    }

    return 0;
}
```

```
root@WISHULAJAYATHUNGA:/mnt/d/AAApera/Sem6/CO327/e19-CO327-Labs/Lab 01# ./ex42
simple-shell> ls
 CO327.docx       a.out      ex2     ex3     ex4     ex42    '~$CO327.docx'
'Fork tree.png'   create.c   ex2.c   ex3.c   ex4.c   ex42.c
simple-shell> ./a.out
This is the parent process
This is the child process
simple-shell> ./ex4
Usage: ./ex4 <path>
simple-shell> exit
root@WISHULAJAYATHUNGA:/mnt/d/AAApera/Sem6/CO327/e19-CO327-Labs/Lab 01#
```

2. Multiprocess Servers

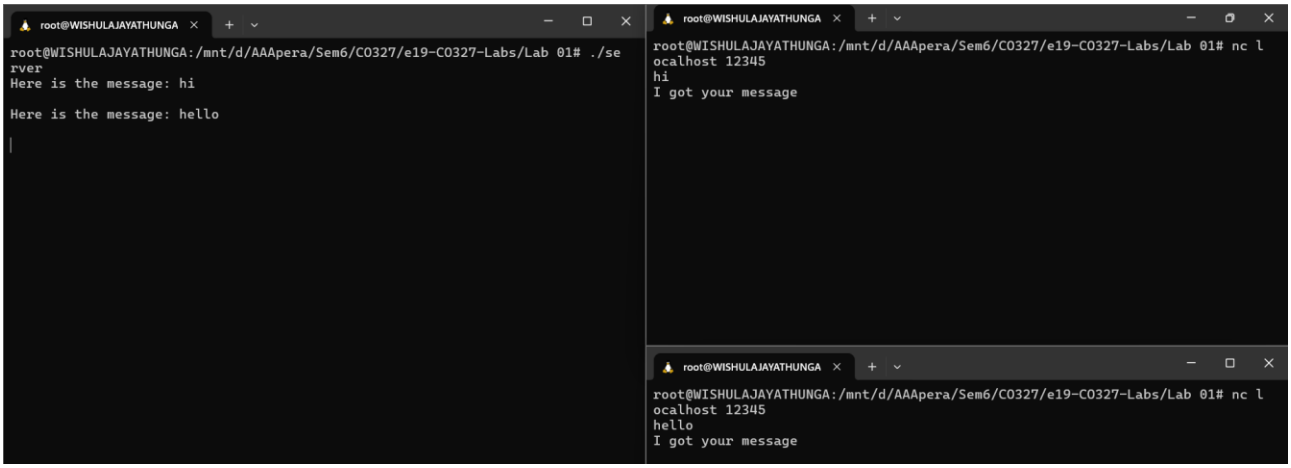
Exercise 5

1.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

void handle_client(int newsockfd) {
    char buffer[256];
    int n;

    // Clear the buffer
    bzero(buffer, 256);

    // Read message from the client
    n = read(newsockfd, buffer, 255);
    if (n < 0) perror("ERROR reading from socket");

    printf("Here is the message: %s\n", buffer);

    // Send a response back to the client
    n = write(newsockfd, "I got your message", 18);
    if (n < 0) perror("ERROR writing to socket");

    // Close the client's socket
    close(newsockfd);
}

int main(int argc, char *argv[]) {
    int sockfd, newsockfd, portno = 12345;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    int pid;

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("ERROR opening socket");
```

```
        exit(1);
    }

    // Initialize socket structure
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    // Bind the host address
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        perror("ERROR on binding");
        exit(1);
    }

    // Start listening for the clients
    listen(sockfd, 5);
    clilen = sizeof(cli_addr);

    // Accept actual connection from the client
    while (1) {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) {
            if (errno == EINTR) continue; // Ignore interrupted system calls
            perror("ERROR on accept");
            exit(1);
        }

        // Fork a new process
        pid = fork();
        if (pid < 0) {
            perror("ERROR on fork");
            exit(1);
        }

        if (pid == 0) { // In the child process
            close(sockfd); // Close the original socket
            handle_client(newsockfd); // Handle the client's request
            exit(0);
        } else { // In the parent process
```

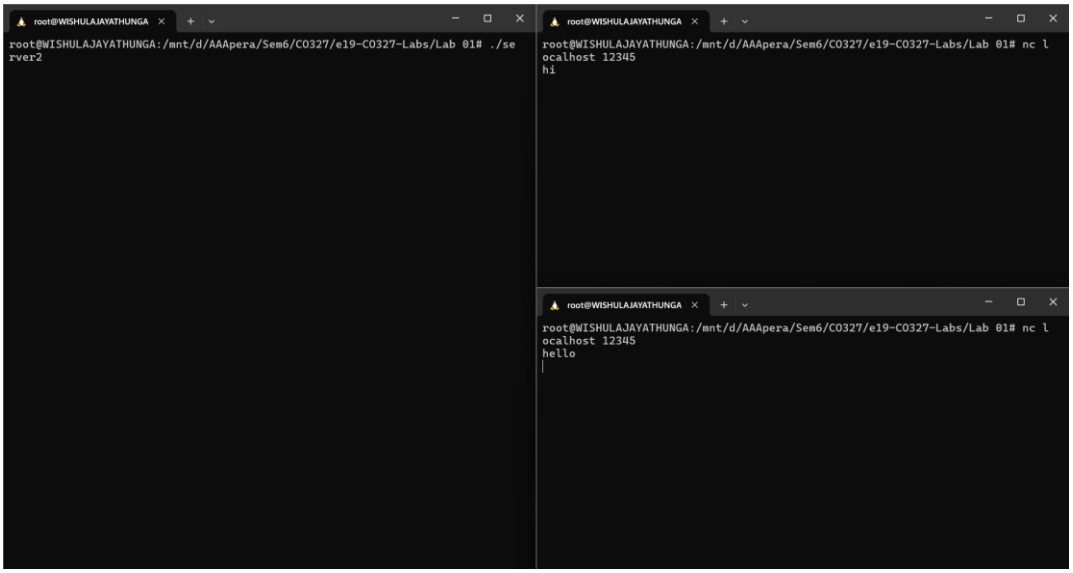
```
            close(newsockfd); // Close the new socket
        }
    }

    return 0; // We never get here
}
```

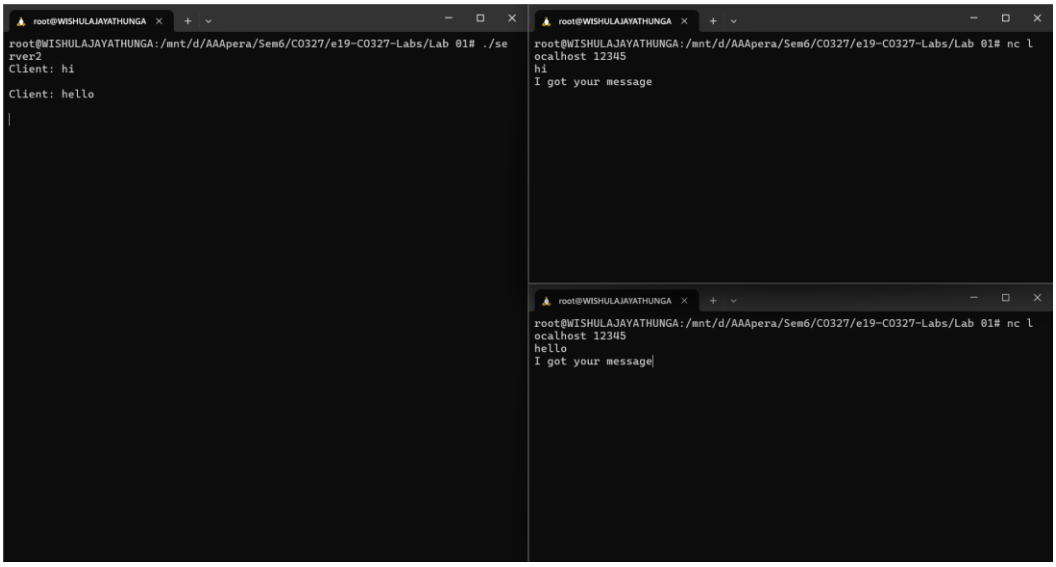


2.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>

void handle_client(int newsockfd) {
    char buffer[256];
    ssize_t n;

    // Clear the buffer
    bzero(buffer, 256);

    // Read message from the client
    n = read(newsockfd, buffer, 255);
    if (n < 0) perror("ERROR reading from socket");

    printf("Client: %s\n", buffer);

    // Send a response back to the client
    n = write(newsockfd, "I got your message", 18);
    if (n < 0) perror("ERROR writing to socket");

    // Close the client's socket
    close(newsockfd);
}

int main(int argc, char *argv[]) {
    int sockfd, newsockfd, portno = 12345;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    pid_t pid;

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
```

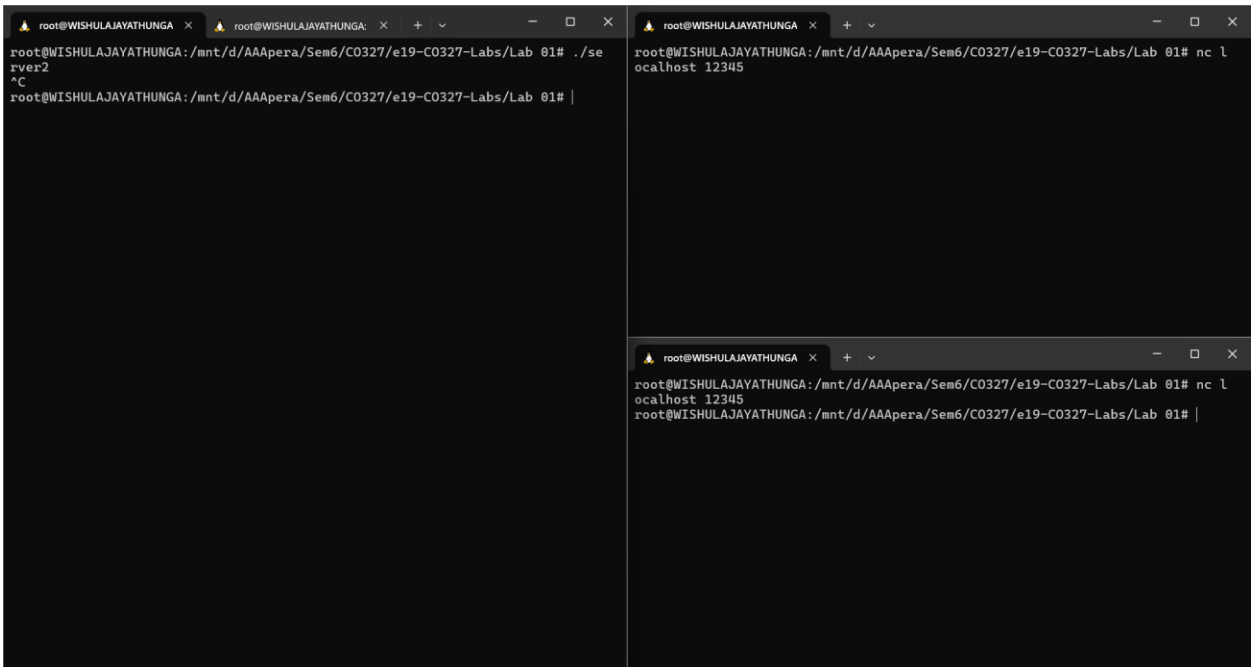
```c
    if (sockfd < 0) {
        perror("ERROR opening socket");
        exit(EXIT_FAILURE);
    }

    // Initialize socket structure
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    // Bind the host address
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
 {
        perror("ERROR on binding");
        exit(EXIT_FAILURE);
    }

    // Start listening for the clients
    listen(sockfd, 5);
    clilen = sizeof(cli_addr);

    // Accept actual connection from the client
    while (1) {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) {
            perror("ERROR on accept");
            continue;
        }

        // Fork a new process
        pid = fork();
        if (pid < 0) {
            perror("ERROR on fork");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) { // In the child process
            close(sockfd); // Close the original socket
            handle_client(newsockfd); // Handle the client's request
              exit(EXIT_SUCCESS);
         } else { // In the parent process
              wait(NULL); // Wait for the child process to terminate
              close(newsockfd); // Close the new socket
         }
    }

    // Close the server socket
    close(sockfd);
    return 0; // We never get here
}
```

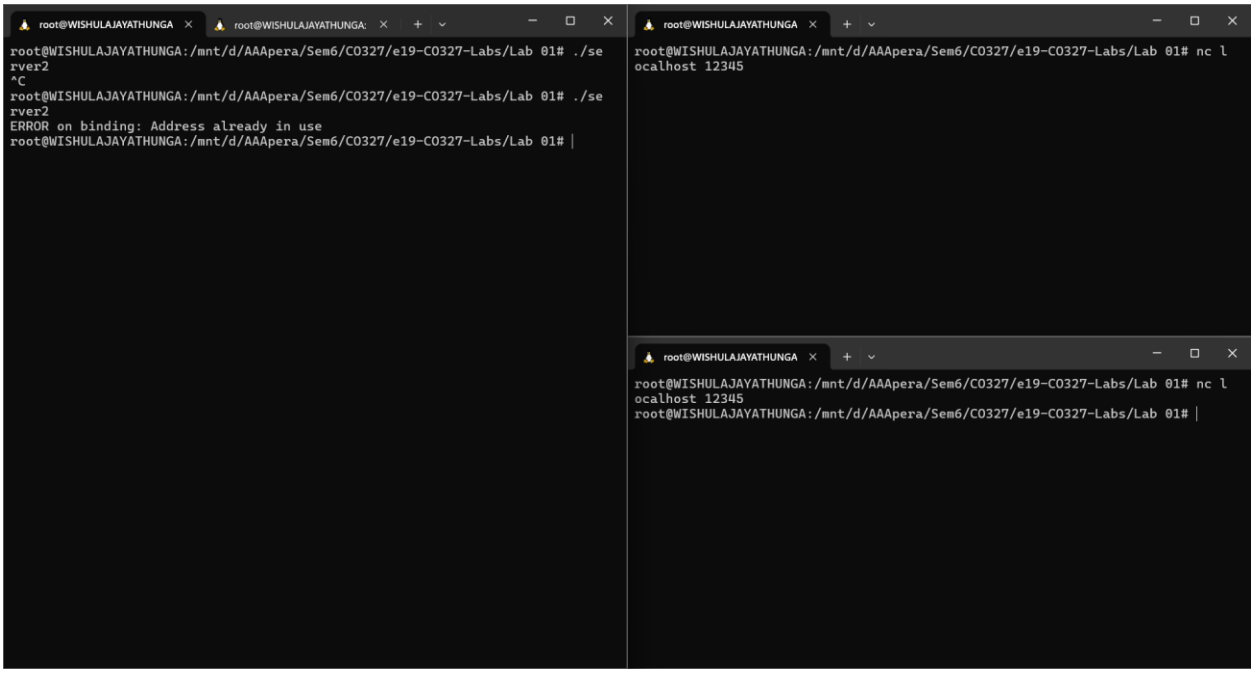




The server parent process calls wait() to wait until the child serving a client terminates, the server would handle one client at a time sequentially. It would not accept new connections until the current child process finishes, leading to a non-concurrent, single-client-at-a-time service model.

3.





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.

4.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

void handle_client(int newsockfd) {
    char buffer[256];
    int n;

    // Clear the buffer
    bzero(buffer, 256);

    // Read message from the client
    n = read(newsockfd, buffer, 255);
    if (n < 0) perror("ERROR reading from socket");

    printf("Here is the message: %s\n", buffer);

    // Send a response back to the client
    n = write(newsockfd, "I got your message", 18);
    if (n < 0) perror("ERROR writing to socket");

    // Close the client's socket
    close(newsockfd);
}

int main(int argc, char *argv[]) {
    int sockfd, newsockfd, portno = 12345;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    int pid;

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("ERROR opening socket");
```

```
        exit(1);
    }

    // Initialize socket structure
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);

    // Bind the host address
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
{
        perror("ERROR on binding");
        exit(1);
    }

    // Start listening for the clients
    listen(sockfd, 5);
    clilen = sizeof(cli_addr);

    // Accept actual connection from the client
    while (1) {
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
        if (newsockfd < 0) {
            if (errno == EINTR) continue; // Ignore interrupted system calls
            perror("ERROR on accept");
            exit(1);
        }

        // Fork a new process
        pid = fork();
        if (pid < 0) {
            perror("ERROR on fork");
            exit(1);
        }

        if (pid == 0) { // In the child process
            close(sockfd); // Close the original socket
            handle_client(newsockfd); // Handle the client's request
            exit(0);
```

```
            exit(0);
        } else { // In the parent process
            close(newsockfd); // Close the new socket
        }
    }

    return 0; // We never get here
}
```

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.