

## COMP 304: Operating Systems - Spring 2025

### *Project 1 Report*

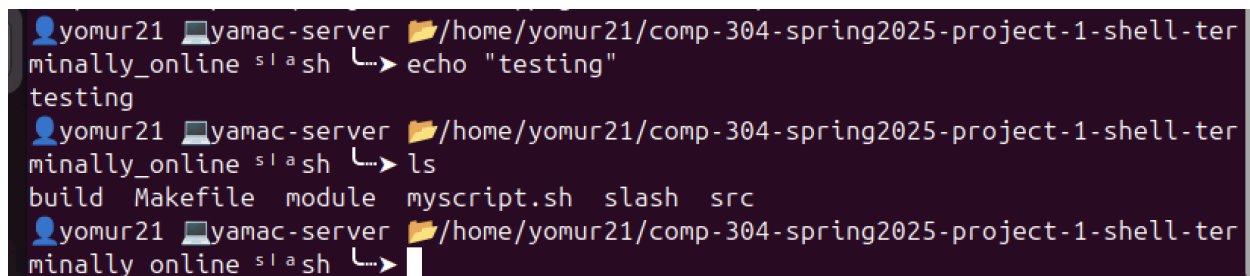
**Team Members:** Sinemis Toktaş (0076644) & Yamaç Ömür (0079458)

**GitHub Classroom Team Name:** terminally\_online

### **PART 1:**

The first part of the project was mainly about implementing path resolution and modifying the `execp()` function call to use the `execv()` function call instead. An important difference between these two `exec` functions is that “`execv()`” requires an absolute path rather than a simple name of the executable file. Therefore, the file to execute must first be found in the user’s environment. First, the code, located inside the “`parse_command`” function, obtains the “`PATH`” environment variable of the user, which includes the possible directories that contain executable files, each delimited by a “`:`” character.

Thus, we implemented path resolution by tokenizing the “`PATH`” environment variable, appending “`/`” and the user’s input command to each possible directory location. Then, we used the “`access(..., X_OK)`” function to check if a function with a given name exists, and if it is executable. If the file is executable, then we assumed that this was the command that the user was looking for. This means that when there are multiple locations for a command in the user’s environment, the code invokes the **first** one that it finds. Finally, we used the found path as the argument to the “`execv`” function, and finished Part 1. An example usage is given below:



```
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online $ echo "testing"
testing
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online $ ls
build Makefile module myscript.sh slash src
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online $
```

**Figure 1:** A screenshot of the `ls` and `echo` commands, both are found using path resolution.

## PART 2:

## 1. Executing Shell Scripts

First, in order to obtain the user’s provided arguments (the name of the shell file), we added the “int argc” and “char\* argv[]” parameters to the main() function. Simply, the main function checks if the user has provided two arguments, and if the second argument ends with a “.sh”, meaning that it is a shell script. If true, the function then runs the “run\_shell\_script” function, which is a helper function that we define ourselves to execute the script line by line.

The “run\_shell\_script” function uses a similar logic to the main function, working as if we were to execute a command. Each line will be treated as a command, so we first create a command structure by calling “parse\_command”. Then we call “process\_command” to execute the command structure, and free the memory that was set for it. Since we don’t call the “prompt” function, the shell script executes and prints its output without asking the user for a prompt. In brief, it can be stated that this part can be easily done using the provided functions. An example usage is given below:

[illegible]

**Figure 2:** A shell script for testing, called “myscript.sh”.

```
yomur21@yamac-server:~/comp-304-spring2025-project-1-shell-terminally_online$ ./slash myscript.sh
Linux
--- 123 ---
"Hello World"
build Makefile module myscript.sh slash src
--- 456 ---
yomur21@yamac-server:~/comp-304-spring2025-project-1-shell-terminally_online$
```

**Figure 3:** The execution of the file in Figure 2.

Please note that the shell exits when the shell script is done.

## 2. Auto-complete

The auto-complete functionality is implemented in the `prompt` and `process_command` functions. First, if the command is an auto-complete command, meaning that the user has pressed the TAB character, the terminating “?” character is removed from the string. Then, similar to the path resolution implementation, we take the “PATH” environment variable and iterate through it to find an executable file whose beginning matches the user’s input. We added an “already\_in\_array” variable to ensure that the same command is not put more than once in the matching files list. When we find a file name that matches, we first check if it is executable and if it has already been added to the matches array, called “matching\_exes”.

Later, we check if the file’s name is completely equal to the user’s input, in that case, we iterate over the user’s current directory and list the files in the current directory. If not, the matching file is added to the `matching_exes` array.

After all the matching executable files have been found, we check the number of matches. If it is 1, we write the string on the “autocomplete\_buf” variable. At the end of the `prompt` function, we check whether this buffer variable is Null or not. If not, this means that the shell must autocomplete their command. Thus, we reset the user’s cursor, erase everything from their current input, and print the auto-completed command name.

If there is more than one matching executable file, the function simply prints each matching file. An example usage of this part is given below:

```
sl^ash Shell implemented by Yamaç Ömür (0079458) and Sinemis Toktaş (0076644)
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online sl^ash ↵ lo
losetup
logrotate
locale-gen
loads.bt
logsave
lowntfs-3g
locale-check
logger
localedef
loweb
logname
login
loadkeys
look
lofromtemplate
lodraw
loadunimap
lomath
loffice
localc
loimpress
loalectl
lowriter
locale
loginctl
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online sl^ash ↵
```

**Figure 4:** The user types “lo” and presses the TAB character. Line by line, the shell lists the executable files that match.

```
loginctl
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online sl^ash ↵ localed
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online sl^ash ↵ localedef
```

**Figure 5:** The user types “localed” and the shell completes it with the only matching common name, “localedef”. **Please note that one keyboard key must be pressed for the auto-completed version to show up in our current implementation.**

```
yomur21 yamac-server /home/yomur21/comp-304-spring2025-project-1-shell-terminally_online sl^ash ↵ vim
.clang-format
module
.git
slash
src
..
myscript.sh
.
Makefile
.gitignore
build
```

**Figure 6:** The user types a command’s name entirely, like “vim”, and the shell lists all the files in the current directory.

## Part 3

### 1. I/O Redirection

The third part of the project required implementing an input and output redirection logic for the shell using the `<`, `>`, and `>>` operators.

The implemented I/O Redirection operator options and their functionalities are as follows:

- `< inputfile` : Reads an input for a command from a file instead of standard input (stdin) which is the keyboard.
- `> outputfile` : Writes output of a command to a file, overwriting the file if it exists, if not it creates a new one.
- `>> outputfile` : Writes output of a command to a file, appending to the file if it exists, if not it creates a new one.

We handled redirection logic inside the `process_command` function after forking a child process.

```

// Command is not a builtin then
pid_t pid = fork();

if (pid == 0) {
    // CHILD

    // This shows how to do exec with auto-path resolve
    // add a NULL argument to the end of args, and the name to the beginning
    // as required by exec

    if (cmd->redirects[0]){ // for the < case
        int fd = open(cmd->redirects[0], O_RDONLY); // Open the file for reading

        if (fd == -1){ // open() failed
            printf("ERROR! : Problem about input redirection (<)");
            exit(1);
        }

        dup2(fd, STDIN_FILENO); // redirect stdin to the file
        close(fd); // close file descriptor
    }

    if (cmd->redirects[1]){ // for the > case

        int fd = open(cmd->redirects[1], O_WRONLY | O_CREAT | O_TRUNC, // Open the file for writing
            FILE_MODE); // Create file permissions: user read and write, group read, other read

        if (fd == -1){ // open() failed
            printf("ERROR! : Problem about output redirection (>)");
            exit(1);
        }

        dup2(fd, STDOUT_FILENO); // redirect stdout to the file
        close(fd); // close file descriptor
    }
}

```

**Figure 7:** Code implementation for the “<” and the “>” cases.

```

if (cmd->redirects[2]){ // for the >> case
    int fd = open(cmd->redirects[2], O_WRONLY | O_CREAT | O_APPEND, // Open the file for writing
        FILE_MODE); // Create file permissions: user read and write, group read, other read

    if (fd == -1){ // open() failed
        printf("ERROR! : Problem about append redirection (>>)");
        exit(1);
    }

    dup2(fd, STDOUT_FILENO); // redirect stdout to the file
    close(fd); // close file descriptor
}

```

**Figure 8:** Code implementation for the “>>” case.

In the child process, based on redirects values, (they were not null if redirection was required) we first open the corresponding file using `open()` with necessary arguments. Then we use `dup2()` to duplicate the file descriptor (FD) onto `STDIN_FILENO` (if `<`) or `STDOUT_FILENO` (if `>` or `>>`). At the end, we close the old FD and continue to the command execution using `execv()`.

```
sinemis linux /home/sinemis/D
-> echo hi > input.txt
sinemis linux /home/sinemis/D
-> cat < input.txt
hi
sinemis linux /home/sinemis/D
-> echo bye >> input.txt
sinemis linux /home/sinemis/D
-> cat < input.txt
hi
bye
```

**Figure 9:** An example usage of the I/O redirection part.

## 2. Piping

In addition to redirection, we implemented support for piped commands, where the output of one command is passed as the input to another command.

Piping was implemented using `pipe()` to create a unidirectional communication channel between processes. For each command in the pipeline, we handled input and output redirection manually:

- For the left command, standard output (`stdout`) is redirected to the pipe's write end.
- For the right command, standard input (`stdin`) is redirected from the pipe's read end.

Multiple pipes are handled iteratively by chaining commands together using the next pointer of the `cmd_t` structure.

```
sinemis linux /home/sinemis/De
-> cat input.txt | grep hi | wc -l
1
sinemis linux /home/sinemis/De
-> 
```

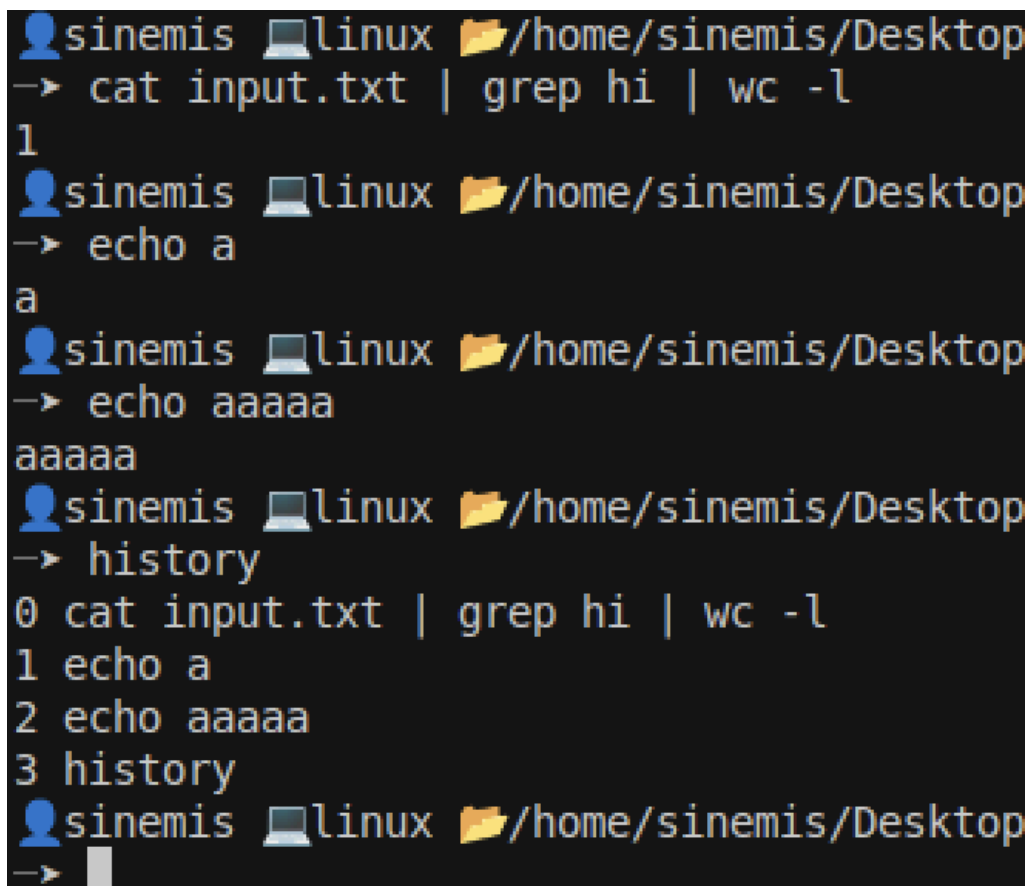
**Figure 10:** An example usage of piping.

### 3. History

Lastly, we implemented a command history feature. The shell now stores up to 140 previous commands. Users can navigate through the command history by pressing the UP (↑) and DOWN (↓) arrow keys.

The history feature was implemented by saving each entered command into a static array `history[MAX_HISTORY_SIZE][512]` inside the prompt function. When the UP arrow key is pressed, the program copies the previous command into the input buffer and prints it. The DOWN arrow key allows navigating forward.

We also implemented a built-in history command, which lists all previous commands together with their assigned integer IDs.

A terminal window with a dark background and light-colored text. The prompt is 'sinemis linux /home/sinemis/Desktop'. The user enters 'cat input.txt | grep hi | wc -l', which outputs '1'. Then the user enters 'echo a', which outputs 'a'. Then the user enters 'echo aaaaa', which outputs 'aaaaa'. Finally, the user enters 'history', which lists the previous commands with their IDs: '0 cat input.txt | grep hi | wc -l', '1 echo a', '2 echo aaaaa', and '3 history'. The cursor is at the end of the 'history' command line.

```
sinemis linux /home/sinemis/Desktop
-> cat input.txt | grep hi | wc -l
1
sinemis linux /home/sinemis/Desktop
-> echo a
a
sinemis linux /home/sinemis/Desktop
-> echo aaaaa
aaaaa
sinemis linux /home/sinemis/Desktop
-> history
0 cat input.txt | grep hi | wc -l
1 echo a
2 echo aaaaa
3 history
sinemis linux /home/sinemis/Desktop
->
```

**Figure 11:** An example usage of history.

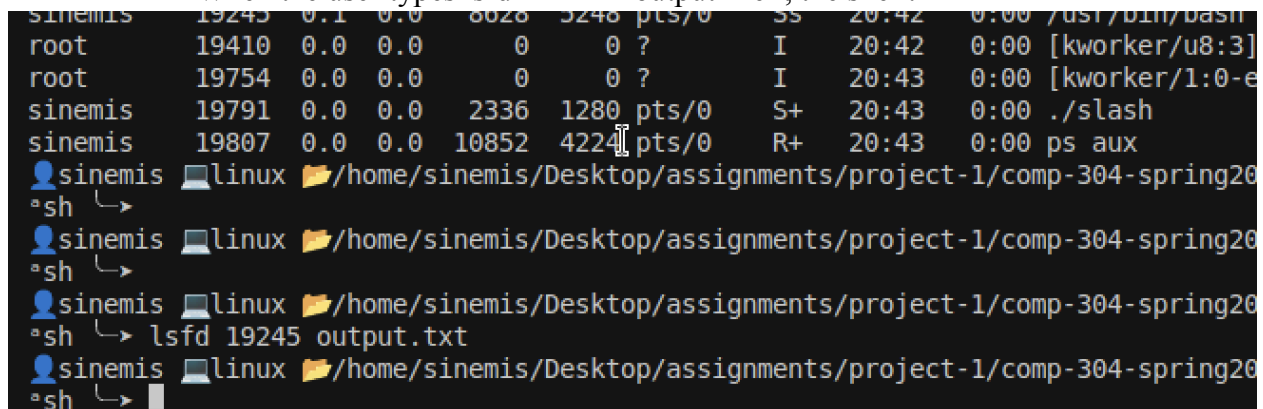


## Part 4

The fourth part of the project required us to develop a custom kernel module (mymodule) and integrate it with our slash shell. With this integration, we had to support `lsfd <PID> <output file>` command inside our slash shell. This part focused on accessing the file descriptors (FDs) of a process, and then writing the command `lsfd <PID> <output file>` that listed all the FDs of the process with the given PID.

Our implementation can be explained in these steps:

- Writing the kernel module (mymodule.c)
  - We updated the given mymodule.c kernel module so that it would create a new file `/proc/lsfd`.
  - The module mymodule allows users to:
    - Write a PID into `/proc/lsfd`
    - Read the open FDs of that process like explained in the introduction of part 4.
  - Functions inside the module:
    - `proc_create()`: to create the `/proc/lsfd` file
    - `lsfd_write()`: handles writing the PID
    - `lsfd_read()`: returns a list of open FDs
  - We used `task_struct`, `files_struct`, and `fdtable` to access the internal kernel structures.
  - We used `kmalloc` to dynamically allocate memory for output.
  - At the end, all resources of module are cleaned when the module is removed.
- Integrating with slash shell
  - When the user types `lsfd <PID> <output file>`, the shell:



```
sinemis 19245 0.1 0.0 8628 5248 pts/0 Ss 20:42 0:00 /usr/bin/bash
root 19410 0.0 0.0 0 0 ? I 20:42 0:00 [kworker/u8:3]
root 19754 0.0 0.0 0 0 ? I 20:43 0:00 [kworker/1:0-e
sinemis 19791 0.0 0.0 2336 1280 pts/0 S+ 20:43 0:00 ./slash
sinemis 19807 0.0 0.0 10852 4224 pts/0 R+ 20:43 0:00 ps aux
sinemis linux /home/sinemis/Desktop/assignments/project-1/comp-304-spring20
sh
sinemis linux /home/sinemis/Desktop/assignments/project-1/comp-304-spring20
sh
sinemis linux /home/sinemis/Desktop/assignments/project-1/comp-304-spring20
sh lsfd 19245 output.txt
sinemis linux /home/sinemis/Desktop/assignments/project-1/comp-304-spring20
sh
```

**Figure 12:** An example usage of the “lsfd” command with the process ID of “19245”.

- Opens /proc/<PID>/fd
- Reads all symbolic links inside
- Resolves what each FD points to
- Writes the information into the given output file

```
project-1 > comp-304-spring2025-project-1-shell-terminally_online > ≡ output.txt
1 fd 0 -> /dev/pts/0
2 fd 1 -> /dev/pts/0
3 fd 2 -> /dev/pts/0
4 fd 37 -> /home/sinemis/.config/Code/logs/20250427T164404/ptyhost.log
5 fd 103 -> /usr/share/code/v8_context_snapshot.bin
6 fd 255 -> /dev/pts/0
7
```

**Figure 13:** The resulting file, “output.txt”, of the command.

- Auto-loading and Auto-removing the kernel module
  - At shell startup, if it is the first slash instance,
    - Meaning if /proc/lsfd doesn't exist yet
    - We run sudo insmod for our module inside main()

```
int main(int argc, char*argv[]) {
    // Check if kernel module is loaded, load it using sudo insmod if not
    if (access("/proc/lsfd", F_OK) != 0) { // check if /proc/lsfd exist
        printf("Kernel module not loaded. Loading now...\n");

        int result = system("sudo insmod ./module/mymodule.ko"); // load the kernel module
        if (result != 0){ // if loading fails
            printf("ERROR ! : Kernel module loading has failed.");
            exit(1); // can't continue without module
        }
    } else{ // if already loaded, notify user
        printf("Kernel module already loaded.");
    }
}
```

**Figure 14:** The module loading logic implemented within the main function.

- If a slash is not the first running instance, than we don't load it again, and instead user is prompted with a message indicating that the kernel module has already been loaded.
- When a slash instance exists, it checks if it is the last instance using a helper function called `check_and_remove_module()`. This function is called in `process_command` when user types `exit`.
  - If it is the last instance, it removes the module using `sudo rmmod mymodule`

```

// Helper function to check if this is the last slash instance, and remove the kernel module
void check_and_remove_module() {
    FILE *fp;
    char buffer[128];
    int slash_count = 0;

    // Get number of slash instances currently running
    fp = popen("pgrep slash | wc -l", "r"); // Open a pipe to run the shell command
    if (fp == NULL) {
        perror("popen failed"); // Pipe creation failed
        exit(1);
    }

    if (fgets(buffer, sizeof(buffer), fp) != NULL) { // Read output from command
        slash_count = atoi(buffer); // Convert output string into an integer
    }
    pclose(fp); // Close the pipe

    if (slash_count <= 1) { // If this is the last slash instance
        printf("Last slash instance exiting. Removing kernel module...\n");
        system("sudo rmmod mymodule"); // Remove the module
    }
}

```

**Figure 15:** The implementation of the helper function “check\_and\_remove\_module”.

## Resources:

- **The book:** Operating System Concepts by Peter Baer Galvin, Abraham Silberschatz, Greg Gagne
- **Open System Call in C:**

<https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/>

<https://man7.org/linux/man-pages/man2/open.2.html>

- **File I/O in C:**

<https://www.geeksforgeeks.org/c-program-to-read-contents-of-whole-file/#2-read-a-file-in-c-using-fgets>

- **How To Find Executable Files in Unix-Based Systems:**

<https://superuser.com/questions/238987/how-does-unix-search-for-executable-files>

- **Dup2 System Call:**

<https://www.geeksforgeeks.org/dup-dup2-linux-system-call/>

- **Writing a simple kernel module:**

<https://devarea.com/linux-kernel-development-and-writing-a-simple-kernel-module/>

- **Task Linked List (scroll down to Process Family Tree):**

<https://www.informit.com/articles/article.aspx?p=368650>