COMP 304 - Operating Systems: Project 1

Due: April 27th, 2025 - 23:59

Hakan Ayral Spring 2025

Notes: This project corresponds to 15% of your course grade; it can be done individually or as a team of 2. You may discuss the problems with other teams and post questions to the discussion forum, but the submitted work must be your own.

Any sources, services or material you use from external sources such as any form of AI and the internet resources should be properly cited in your report.

Contact TAs: Doğan Sağbili, Semih Erken, Ali Bozkurt

Github Classroom Link: https://classroom.github.com/a/72rbdH8n

Description

This project is a variation on the programming project **Project 1 - UNIX Shell** at the end of Chapter 3 of our textbook (Operating System Concepts).

The main part of the project requires you to develop an interactive Unix-style operating system shell, called `sla`**`sh`** in C. After executing `sla`**`sh`** , it will read commands from the user via console (i.e. stdin) or a file and execute them. Some of these commands will be *builtin* commands, i.e., specific to `sla`**`sh`** and implemented in the same executable binary, while it should be able to launch other commands which are available as part of your own linux system. The project has four main parts (95 points) in addition to a report (5 points), the report should contain enough details for a reader to understand the following three things: the structure of source code, how the code works, your development process (your approach, steps, findings, unexpected events). We suggest starting with the first part and building the rest on top of it.

Part I (15 pts.)

- Use the skeleton program provided as a starting point for your implementation. The skeleton program reads a line of commands from `stdin`, parses it, and separates it into arguments using whitespace as the delimiter. You will implement the action that needs to be taken based on the command and its arguments entered in `sla`**`sh`** . Feel free to modify the command line prompt and parser as you wish.
- Do not change the fundamental structure of the source code and do not rename the fundamental types and function names. Use the given **cmd_t** structure (you can ex-

tend it if needed) and the provided loop around the prompt and command processing functions. If in doubt ask through the **discussion board**.

- Use the provided **`Makefile`** to compile your code. Type **`make help`** to get a list of build targets. If your application needs any extra configuration steps (i.e. installing a package or library, copying files to correct locations, setting up environment variables, etc.) to build your code, you can make additions to the make file, but do not change the name or location of the files.
  - This is important because an automated system will be compiling your code using that make file, we are not going to manually compile or execute your code.
  - If you mention **any manual steps on your report or readme.md file on git repo they will be ignored** for the purposes of grading.
  - Make sure to check whether the latest version of the code on your repo compiles and works on its own without requiring any extra steps with just **git clone** and **make all**; test this on a clean VM to prevent the "but it is working fine on my computer™" case.
- Command line inputs (except builtin commands) should be interpreted as program invocation. The shell must **fork** and **execute** the requested programs. Refer to ***Part I - Creating a child process*** from the textbook.
- Do not use the **`exec()`** family of calls that are prefixed with `p` such as **`execp()`** that automatically search for executable files. Instead, use the **`execv()`** library call and implement **path resolution** yourself.

Part II (15 + 15 pts.)

1. Executing shell scripts (15 pts.)

General purpose shell programs support shell scripting [1] with a rich set of functionalities. For this project we expect your shell program to provide a very simple script execution environment with only the following requirements:

- Your shell program should take the name of a **.sh** file as a command argument, which contains the command to be executed line by line in a text format.
- It should execute each line in order, one after the other, as if the commands were typed through the keyboard.
  - You should wait for the execution of a line to end before proceeding with the execution of the next line.
- Your shell should provide the output to the screen (i.e. stdout) the same way it does for manually entered commands; but it should not print out the prompt and the line being executed.

[1] see Shell scripts at https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html

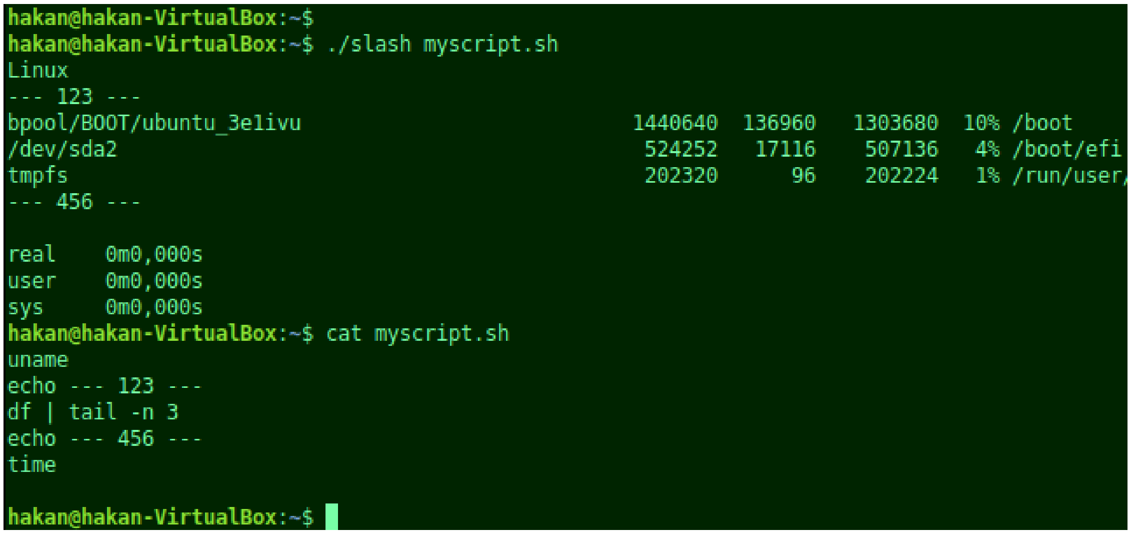

Figure 1: Slash executing a shell script named myscript.sh, later user shows the content of script with cat

- When all the lines in the **.sh** file are executed your shell should exit (and **not** wait for any additional input through the keyboard).

2. Auto-complete (15 pts.)

You are required to handle auto-completion of partially written commands in your shell; this includes all the executable files on the directories listed in the **PATH** environment variable of the system (not just the executables in current directory or built-in commands).

- While typing a command if the Tab key is pressed, sla**sh** should automatically complete the command.
- If there are more than one match found, it should list all the possible matches.
- If the command is fully typed, then the Tab key is pressed, it should list the files in the current directory.

Part III (10 + 10 + 10 pts.)

1. I/O redirection (10 pts.)

In this part of the project, you will implement I/O redirection for sla**sh** . For I/O redirection if the redirection character is >, the output file is created if it does not already exist and overwritten if it does exist. For the redirection symbol >> the output file is created if it does not exist and appended to if it does exist. The < character means that input is read from a file. See "**IV. Redirecting Input and Output**" on page P-14 in the textbook.

A sample terminal line is given for I/O redirection below:

```
slash> program arg1 arg2 > outputfile >> appendfile < inputfile
```

2. Piping (10 pts.)

In this part, you will handle program piping for sla**sh** . Piping enables passing the output of one command as the input of second command. To handle piping, you would need to execute multiple children (**not limited to two**), and create a pipe that connects the output of the first process to the input of the second process, etc.
Start by supporting piping between two processes, but at the end you must be able to handle arbitrarily long chains of pipes. See "**V. Communication via a Pipe**" on page P-15 in the textbook.

Below is a simple example for piping:

```
slash> ls -la | grep search-this-text | wc
```

3. History (10 pts.)

Your program should implement a command history feature. By using the up and down arrow keys user should be able to browse through the previously executed commands (very much like how bash does). It should also implement a **built-in command** called **history** which lists the last executed commands with an integer id (once again very much like the actual history command on bash). Your command history does **not** have to persist through different invocations of the shell (i.e. you don't have to save it to a file).

Part IV (20 pts.)

lsfd <PID> <output file> (Must be written in C):

You are required to implement an **lsfd** command which retrieves information about all the files open by a process with a given PID. Each line in the output text file should correspond to a file descriptor of an open file as kept in the kernel data structures, and should contain the **File descriptor no**, **Name**, **Size** in bytes and **Path** of the file.

Obtaining process information requires kernel-level support, you are required to write a loadable kernel module. Please read the sections "Programming Project - Introduction to Linux Kernel Modules" at the end of **Chapter 2**, "Project 2 - Linux Kernel Module for Task Information" and "Project 3 - Linux Kernel Module for Listing Tasks" at the end of **Chapter 3** of the textbook. Any code which is not part of the kernel module must be implemented as part of the sla**sh** executable.

- When `slash` is executed it should check whether the kernel module is already loaded (this can happen if another instance of `slash` is already running) and load the kernel module with the **insmod** command using **sudo** only if it is not already loaded
- if module is already loaded it should just notify the user that the module is already loaded.
- It is your code's responsibility to load the kernel module at run time (except asking for sudo password if necessary); do not expect the user to load the kernel module for you. (If your code can not load the kernel module on its own, this part will be graded zero.)
- `slash` should remove the module from kernel if it is the last instance (i.e. no other `slash` processes running) and is currently exiting.

Read "II. Loading and Removing Kernel Modules" on **page P-3** in textbook.

Implementation Hints:

The information below can be found on the kernel programming textbook and internet, but here is a step by step count of which kernel data structures you need to navigate through to obtain the necessary information. Although it may look scary at first, lsfd corresponds to a small amount of code relative to the rest of the project.

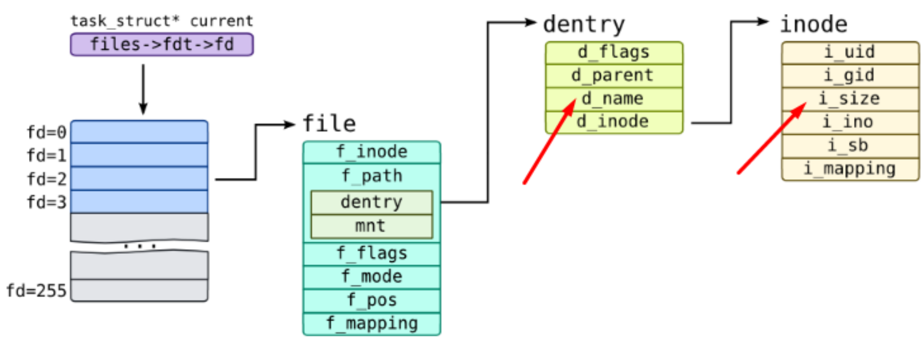

Figure 2: Kernel data structures to navigate

1. The struct named **task_struct** defined in `linux/sched.h line 785` is the key data structure in the kernel which represent the PCB for the processes. You need to use it in order to obtain necessary information such as process name and process start time.
2. The **for_each_process()** macro allows easy iteration over all processes in the system.
3. Open file information for a process is stored in the pointer to **file_struct** named **files** in the **task_struct** (i.e. PCB) for the process. `linux/sched.h line 1155`

4. The **files_fdtable()** macro defined in linux/fdtable.h line 66 lets you access the file descriptor table for the open files. It takes a pointer to **file_struct** and returns a pointer to **fdtable**.

5. The **fdtable** structure contains a pointer to an array named **fd** (see image below). You can access each item as **fd**[**i**] with **i** starting from **0** and getting incremented one by one as long as **fd**[**i**] is not equal to **NULL**. The value of **i** corresponds to **file descriptor no** for that file (i.e. i=0 → FD0, i=1 → FD1, ...). The total number of file descriptor can only be known if you keep counting until **NULL**.

```
struct fdtable {
        unsigned int max_fds;
        struct file __rcu **fd;      /* current fd array */
        unsigned long *close_on_exec;
        unsigned long *open_fds;
        unsigned long *full_fds_bits;
        struct rcu_head rcu;
};
```

Figure 3: fdtable structure

6. Each member of the **fd** array is a pointer to a **file** struct which is defined in linux/fs.h line 1094. You can use the **f_path** field of the **file** struct to access the **dentry** struct which in turn let's you access the name, the path and the file size. (see figure 2)

```
1094 struct file {
1095         file_ref_t                      f_ref;
1096         spinlock_t                      f_lock;
1097         fmode_t                         f_mode;
1098         const struct file_operations    *f_op;
1099         struct address_space            *f_mapping;
1100         void                            *private_data;
1101         struct inode                    *f_inode;
1102         unsigned int                    f_flags;
1103         unsigned int                    f_iocb_flags;
1104         const struct cred               *f_cred;
1105         /* --- cacheline 1 boundary (64 bytes) --- */
1106         struct path                     f_path;
1107         union {
```

Figure 4: file structure

As in linux everything is represented as files, you will notice network sockets, hardware devices and many other things beside the regular files show up as "open file" when you probe different processes (see figure below, entries with size 0); on the other hand this doesn't affect your code as they behave exactly the same as regular files. Sample output expected from the lsfd command:

```
hakan@hakan-VirtualBox:~/Desktop/kernel-module$ cat lsfd-output.txt
Process: cinnamon-sessio[1576]
FD0:     name: null      size: 0 bytes   path: /dev/null
FD1:     name: null      size: 0 bytes   path: /dev/null
FD2:     name: .xsession-errors          size: 8933 bytes       path: /home/hakan/.xsession-errors
FD3:     name: [eventfd]         size: 0 bytes   path: anon_inode:[eventfd]
FD4:     name: [eventfd]         size: 0 bytes   path: anon_inode:[eventfd]
FD5:     name: [eventfd]         size: 0 bytes   path: anon_inode:[eventfd]
FD6:     name: UNIX-STREAM       size: 0 bytes   path: socket:[23227]
FD7:     name: UNIX-STREAM       size: 0 bytes   path: socket:[24184]
FD8:     name: [eventfd]         size: 0 bytes   path: anon_inode:[eventfd]
FD9:     name: renderD128        size: 0 bytes   path: /dev/dri/renderD128
FD10:    name: renderD128        size: 0 bytes   path: /dev/dri/renderD128
FD11:    name: renderD128        size: 0 bytes   path: /dev/dri/renderD128
FD12:    name: UNIX-STREAM       size: 0 bytes   path: socket:[24195]
FD13:    name: UNIX-STREAM       size: 0 bytes   path: socket:[24212]
FD14:    name: [eventfd]         size: 0 bytes   path: anon_inode:[eventfd]
FD15:    name: UNIX-STREAM       size: 0 bytes   path: socket:[24213]
FD16:    name: UNIX-STREAM       size: 0 bytes   path: socket:[23289]
FD17:    name: UNIX-STREAM       size: 0 bytes   path: socket:[23290]
FD18:    name: UNIX-STREAM       size: 0 bytes   path: socket:[23468]
FD19:    name: UNIX-STREAM       size: 0 bytes   path: socket:[23545]
FD20:    name: UNIX      size: 0 bytes   path: socket:[25721]
hakan@hakan-VirtualBox:~/Desktop/kernel-module$
```

Figure 5: Sample output from lsfd command as written to text file

Other Hints:

- To make your user level `sla`**`sh`** process and kernel module commmunicate with each other, use the **/proc** file system. Read from the textbook sections titled "III. The /proc File System" on page P-5, "II. Reading from the /proc File System" on **page P-17** and "I. Writing to the /proc File System" on page **P-16**.
- In the kernel you can not load any 3rd party libraries other than what is already part of linux kernel, therefore you do not have access to stdlib or libc.
- Test your kernel module outside of `sla`**`sh`** first to check if it works.
- You need to be in a working directory with no spaces in the path to build the kernel module with the provided Makefile !

References

We strongly recommend you to start your implementation as early as possible as **it may require a decent amount of research**. The following links might be useful:

- 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
- Linux Cross-Reference:
  https://elixir.bootlin.com/linux/latest/source

Deliverables and Requirements

You are required to submit the following in a zip file (name it username1-username2.zip) to LearnHub.

- You must push your work to GitHub classroom in addition to LearnHub submission. We will be checking the commits as part of project evaluation, your commits must reflect the progression of your work.
- Although not *required*, we highly encourage you to use the provided `.clang-format` file to autoformat your code. Run the following command to apply formatting.

```
find . -name '*.[ch]' -exec clang-format -i {} \;
```

- `.c` source file that implements the sla**sh** shell. Your code must include enough comments to let another computer engineer understand how it works.
- `.c` source file of the Kernel module.
- Any supplementary files for your implementation (Makefiles, header files, etc.)
- (5 points) a report describing your implementation. Include screenshots in your report and submit it as a `pdf` file.
- You should keep your GitHub repo updated from the start to the end of the project. Do not commit at the very end when you are finished, instead make consistent commits as you make progress. You will be graded accordingly.
- Implement your code for the Linux OS and also test your code on a different clean Linux installation (i.e. a VM) to make sure that you don't have parts of code which only works on your local setup but not on other systems.