PROJECT REPORT

The purpose of this project is to create a custom shell.
Notice: The project code is tested on an Ubuntu 20.04 virtual machine. It it not recommended to run the program directly on your machine especially if you want to use the custom kernel module.
Notice: At the beginning of the skeleton code, I added some #define blocks. You will need to adjust the MOUDLE_PATH and PLOTTER_FILE_PATH to be able to use the custom kernel module.

Part I - Execution of the Commands

Other than the parent-child relationships between processes, there exist another concept concerning the processes, which is the “process group”. There exist couple of reasons why there exist such a grouping. First one is that, when a signal is sent through the keyboard such as SIGINT, this signal is captured by the current group of process that have the control of the terminal. This group is called the “terminal controlling group”. Being “terminal controlling group” means that the current input that is typed to the terminal can be captured only by this group of processes (this is not the case for STDOUT). That is the reason why the processing belonging to the terminal controlling group are the ones receiving the signal. The shell is creating a process group for the entered command, and assigns it as the terminal controlling group. This allows our custom shell to not be affected by the signals such as SIGINT, which are captured by the currently running command.
To handle the group creation, I needed some kind of synchronization between the child and the parent. I solved this issue with the usage of pipes. There exist 3 pipes. First one is for child to tell the parent that the group creation was successful. Parent waits for that message and if the child process should be run at the foreground, it assigns the group of the child as terminal controlling group (as the child created the new group, the group id will be pid of the child). Then the parent tells the child that it has the control of the terminal. Child waits for this response to proceed with executing the command, because if the child executes the command before the parent assigns the child as terminal controlling group, the child may not get the input from the terminal. Then the parent wait for a last signal to proceed. This signal is in fact for the case where the child is a background process (which doesn’t require parent to wait for the child before moving on) but it cannot find the command to execute in the path. In this case the child prints a message to the console indicating that the command was not found. If the parent doesn’t wait for the response, the message may be printed after the new prompt is displayed to the user. For that, I used a special type of pipe. When the execv() command is executed the text area of the address space of the child process is replaced but the file descriptors remain opened. This is a problem because the parent is waiting for a response from the child in a blocking read() state. If the child cannot find the process on path it closes the write end of the pipe and parent gets a null for the read() call. But if the execv is executed, no one closes it and parent waits indefinitely.To solve this, I initialized the last pipe such that if an execv executes, the file descriptor is closed which will tell the parent

to not to wait by a null message. Additionally, if the executed command is a foreground command, the parent waits for the process to be finished and takes control of the terminal again. The problem here is as the parent being “shellect” is not the session leader nor in the terminal controlling group anymore. Thus, when trying to take control back, it receives a signal SIGTTOU. I handle this problem by ignoring that signal before trying to get the control back, and restore it again after taking the control.
Other than handling the process groups, mainly the code works as the following:
Child receiving the command to execute, looks at the command name. If the command starts with a “/“ it means the user provided an absolute path and the child only try to execute that command without checking other paths. If the command starts with “./“ the child understands that this is a relative path and again does not look the command in the path. If only the command name is given, the code first checks the current directory for that command, then checks the directories which are in the environment variable PATH. On the parent side, if the process is a background process, the parent doesn’t wait for the child, but if this is a foreground process, parent waits for the child to finish execution.

Part II - I/O Redirection

For this part, I investigated how the bash in my Ubuntu 20.04 machine processes the commands. I realized that, if there are multiple “to” redirections (> or >>) it only writes to the one indicated by the last one. But for the files indicated by the (>), it removes the content anyways. I applied the same logic in my code. I also place another field to the command_t data structure which hold which redirection index is the last one. This field is set at the parser. To do that I am opening the provided files anyways with truncation (if “>”) or appendix (if “>>”) and closing if the last provided redirection is not that one.
Other than this logic, I changed the file descriptors of the child before entering into execv() (or another built-in command) and after the execution, I am restoring the “stdout” of the child so that it can print any informative messages after the execution. To replace the STDIN_FILENO and STDOUT_FILENO with the content of another file, I use dup2() and to store the original STDOUT, I used dup().

Part III - Custom Implementation of Some Shell Commands

a-Hexdump

The task in this part can be divided into two subtasks. The first one is handling the file to read the input and the second is to handle the printing logic of the output. For first part, I first checked whether a file is provided as argument. The file descriptor that I am reading from is constructed as the following: if the user provided a file to read, that file is opened. If not provided the program duplicates the STDIN_FILENO and uses it to read the data. The reason why I duplicated the STDIN is to get rid of some if else statements for closing the file descriptor that is being used. If an input file is provided, I need to close it, it not I shouldn’t close the STDIN. Thus, I duplicated it if the file is not given to close the file descriptor anyways.

To handle the program printing the hexadecimal numbers, I first calculated the size of each group, and read that much bytes from the input file (or STDIN). Then for each group, I first printed the initial part giving the number of bytes printed until that (0x000008 for ex.), and then for each read character from the input file, I printed the corresponding hexadecimal number. Then after printing all the numbers, I print the characters in the buffer which I used for storing the current group.

```
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ xdd -g 1 example.txt
00000000:  41 74 20 68 6f 6d 65 2c 20 77 65 20 68 61 76 65  At home, we have
00000010:  20 68 61 64 20 74 77 65 6c 76 65 20 75 6e 68 61   had twelve unha
00000020:  70 70 79 20 79 65 61 72 73 20 6f 66 20 74 75 72  ppy years of tur
00000030:  6d 6f 69 6c 20 61 6e 64 20 64 69 73 73 65 6e 73  moil and dissens
00000040:  69 6f 6e 2c 20 6f 66 20 67 72 6f 75 70 20 63 6f  ion, of group co
00000050:  6e 66 6c 69 63 74 20 61 6e 64 20 63 6c 61 73 73  nflict and class
00000060:  20 73 74 72 69 66 65 2e 20 4f 66 20 64 69 76 69   strife. Of divi
00000070:  73 69 6f 6e 73 2c 20 61 6e 64 20 68 61 74 72 65  sions, and hatre
00000080:  64 73 20 61 6e 64 20 61 6e 74 61 67 6f 6e 69 73  ds and antagonis
00000090:  6d 73 2e 20 48 61 6c 66 20 61 20 67 65 6e 65 72  ms. Half a gener
000000a0:  61 74 69 6f 6e 20 68 61 73 20 67 72 6f 77 6e 20  ation has grown
000000b0:  75 70 20 6b 6e 6f 77 69 6e 67 20 6e 6f 20 6f 74  up knowing no ot
000000c0:  68 65 72 20 61 74 6d 6f 73 70 68 65 72 65 2e 20  her atmosphere.
000000d0:  49 20 62 65 6c 69 65 76 65 20 6f 75 72 20 63 68  I believe our ch
000000e0:  69 6c 64 72 65 6e 2c 20 6f 75 72 20 77 68 6f 6c  ildren, our whol
000000f0:  65 20 63 6f 75 6e 74 72 79 2c 20 63 61 6e 20 61  e country, can a
00000100:  67 61 69 6e 20 6c 69 76 65 20 69 6e 20 61 20 77  gain live in a w
00000110:  6f 72 6c 64 20 77 68 65 72 65 20 70 65 61 63 65  orld where peace
00000120:  2c 20 66 72 69 65 6e 64 73 68 69 70 20 61 6e 64  , friendship and
00000130:  20 6d 75 74 75 61 6c 20 72 65 73 70 65 63 74 2c   mutual respect,
00000140:  20 61 62 69 64 65 2e 0a                          abide.
```

```
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ xdd -g 2
hello my name is Fatih, I am 23 years old.
00000000:  68 65 6c 6c 6f 20 6d 79  hello my
00000008:  20 6e 61 6d 65 20 69 73   name is
00000010:  20 46 61 74 69 68 2c 20   Fatih,
00000018:  49 20 61 6d 20 32 33 20  I am 23
00000020:  79 65 61 72 73 20 6f 6c  years ol
00000028:  64 2e 0a  d.
```

b-Command Aliases

The implementation logic for this part is as the following: when the program starts executing, restore the aliases from the file where the aliases were saved. This step is done by parsing the file and saving alias names in an array and saving the corresponding commands in another array of struct command_t. When an alias is given, store it a file where the program can read this file no matter where it was ran. Then make the command available for the current session. When a command is provided, first compare the name with the created aliases. If the command name exists in the alias names array, find the corresponding command in the alias commands list and replace the content of the current provided command with this one and continue processing the command as usual.

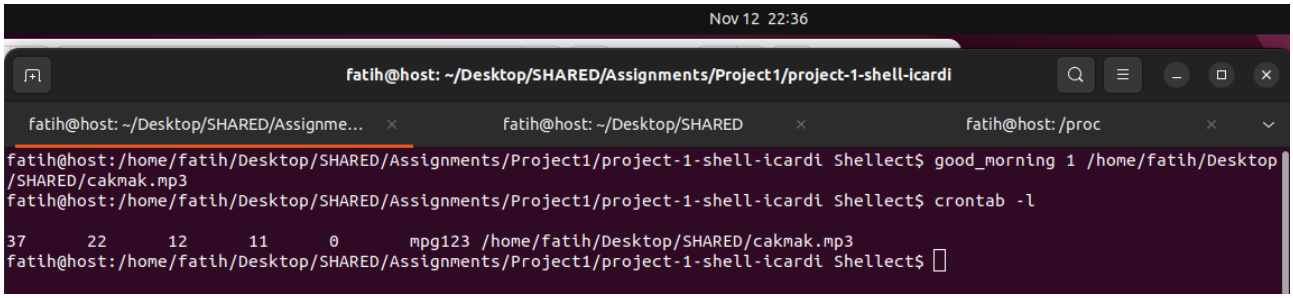
To achieve these steps, I first created my alias file in the home directory (not manually, inside the C code) as that one is accessible from everywhere. Then, when a new alias is added, I retrieve the prompt that is provided by the user using the fields of the parsed command. Then, I store it in the alias file. While doing this, I also keep some of the fields such as redirection, being background or not etc. in some variables. Then I send the command name and the arguments (not other fields such as redirection) to the parser as a new command string. Upon receiving the raw command which contains command name and arguments, I place other information such as redirection files and being background or not etc. into the received command. Then, I add the name of the command to the alias command names list and I put the created command into the alias

commands list. This process makes the newly created alias available for the current session as well as saves it into the alias file. When I open a new session, before the program enters in the loop, these aliases available in the file where the aliases were saved are again parsed and put into the corresponding lists.

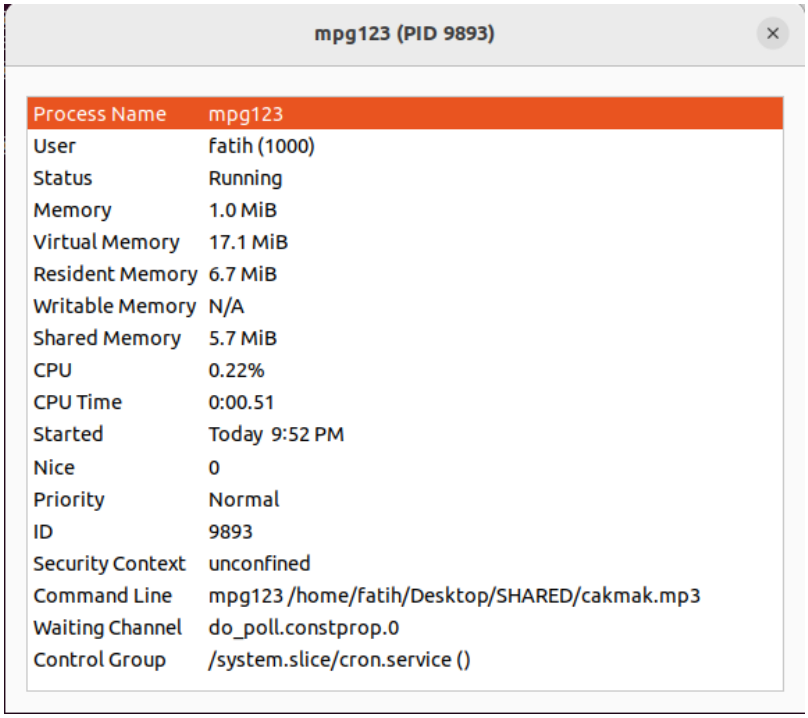
```
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ alias lll ls -l
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ lll
total 88
drwxrwxr-x  5 fatih fatih   160 Nov 12 21:40 build
-rw-rw-r--  1 fatih fatih   486 Nov  4 17:22 CMakeLists.txt
-rw-rw-r--  1 fatih fatih   328 Nov  8 22:52 example.txt
-rw-rw-r--  1 fatih fatih   130 Nov 11 16:05 hey.txt
-rw-rw-r--  1 fatih fatih  1389 Nov  9 21:19 Makefile
drwxrwxr-x 20 fatih fatih   640 Nov 12 21:37 module
-rw-rw-r--  1 fatih fatih 20692 Nov 12 15:25 out.png
-rw-rw-r--  1 fatih fatih  2408 Nov 12 15:22 plot.gp
-rw-rw-r--  1 fatih fatih   529 Nov  4 17:22 README.md
-rwxrwxr-x  1 fatih fatih 36872 Nov 12 21:40 shellect
drwxrwxr-x  3 fatih fatih    96 Nov 12  2023 src
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ exit
fatih@host:~/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi$ ./shellect
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ lll
total 88
drwxrwxr-x  5 fatih fatih   160 Nov 12 21:40 build
-rw-rw-r--  1 fatih fatih   486 Nov  4 17:22 CMakeLists.txt
-rw-rw-r--  1 fatih fatih   328 Nov  8 22:52 example.txt
-rw-rw-r--  1 fatih fatih   130 Nov 11 16:05 hey.txt
-rw-rw-r--  1 fatih fatih  1389 Nov  9 21:19 Makefile
drwxrwxr-x 20 fatih fatih   640 Nov 12 21:37 module
-rw-rw-r--  1 fatih fatih 20692 Nov 12 15:25 out.png
-rw-rw-r--  1 fatih fatih  2408 Nov 12 15:22 plot.gp
-rw-rw-r--  1 fatih fatih   529 Nov  4 17:22 README.md
-rwxrwxr-x  1 fatih fatih 36872 Nov 12 21:40 shellect
drwxrwxr-x  3 fatih fatih    96 Nov 12  2023 src
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$
```

c-Good Morning

Crontab runs a daemon in the background and checks a file where it keeps the jobs to be executed at a given time. The format was as minute, hour, day of month, month, day of week (Sunday being 0) which are followed by the command to execute. Then I learnt that I can add an entry to the file that is checked by the crontab by [(crontab -l 2>/dev/null; echo “<new_entry>") | crontab -u <usr> -]. The first part is to get the list of existing crontab entries, the following part is for your own entry. The part before the pipe will result in a string where the new entry is appended



under the pre-existing commands, which will given to the following crontab command to update its entry file. Following the information that I just provided, I first created my entry string benefitting from the time_t and tm objects. Then I replaced the parts in the command that I provided using snprintf command and used system() command to execute it.



Note: I used mpg123.

d-Custom Command

For this part, I created a piano like program (command name is piano). Inspiring from the previous task, I learnt that I could achieve that using SoX utilities “play” command. The syntax was as the following: “play -n synth 0.5 sin <frequency of the note>”. I knew that I needed to capture the input from the terminal as soon as the buttons are pressed. Before starting the project, I had checked the “prompt” function and I had investigated the part related to the changing terminal settings (termios). I used the exact same modifications to get the character inputs as soon as they are available (and I restored them when the user quits). I assigned (a, s, d, f, h, j, k ,l) characters (do, re, mi, fa, sol, la, si, do). Other than playing the notes, the program is able to record the played notes into a file and play them later. For this part, I also implemented a mechanism which is measuring the time between the consecutive character pushes and this information is also saved in the file where the recording is saved. The content of a recording file is typically: [letter(1 byte)waiting_duration(4 bytes)]*new_line_chracter (indicates end of recording). However, the waiting times recorded are a bit noisy in a way that, as the program sleeps by providing number of seconds to sleep, the sleeping time takes discrete values rounded to floor. Other than that, I believe the program works pretty well. I also tried to run the new sound at the background by creating a new child process for each pressed key so that the program can accept and play another sound while waiting for other child. However, the sounds were mixing, resulting in a bad sound. Here is how you can use the command:

Type “piano -r record.txt”, then press the aforementioned keys to hear the sound and record them in record.txt. Press enter to quit.

Type “piano -p record.txt”, to play your recording.

Unfortunately I cannot show a demo for this command as the demo requires sound.

Part IV - Kernel Module Implementation

For this part, I benefitted from a variety of sources. I used the method provided in this link from “informit.com" to iterate over the children of a process. I checked this link from “huioo.com" to investigate and search for some fields of the task_struct. And most importantly I dived into this link from “tldp.org" to build my module. I used the last link especially to figure out how I can exchange information between a process and a module. I learnt that, as the kernel modules are used as an interpreter between the computer and the devices, a module can introduce a device to the kernel. Receiving the registration request from the device, kernel adds this device type among its supported device types (check “/proc/devices” file) and it returns the corresponding “major number”. When a new device is introduced, there is an abstract file which is used to handle the communication between the device and the computer. The major number is used at this point. The new abstract file is associated with a major number, which indicates which module is responsible for handling the open/read/write/close operations of this file in a special manner. There exist (as far as I know) two types of devices. One is character device and the other one is block device. I used the former for my module.

My module which has a single parameter (being an integer representing a pid), when loaded with insmod command, registers a new device together with the file operations to handle open/read/write/close and it uses printk() kernel function to print the major number to the kernel log. The purpose of this is to inform the process loading the module about the major number to use when creating an abstract device file. When the module is unloaded, it simply unregisters the device. The file operations are as the following:

Open: the module increments its Device_Open counter which is used for controlling the number of processes that are operating on the device file at the same time. As in the documentation that I read, only a single process was allowed, I kept it as it is.

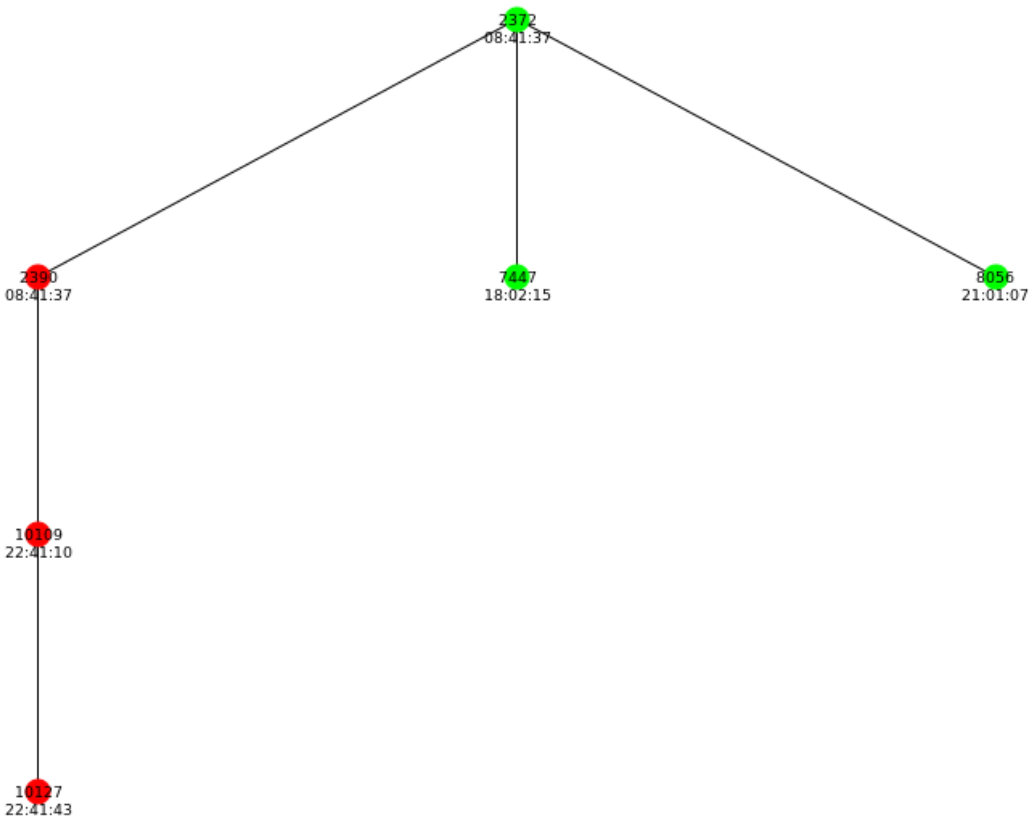
Close: decrement the Device_Open counter and make the device file available for further usage.

Write: this function is not supported by my module, so I print an error message to the kernel log and returns -EINVAL which means invalid argument that I also find in the link that I mentioned.

Read: this is the file operator which is responsible for the actual task. For this part, I also benefitted from link showing the example iteration over the children of a process. First, I thought to implement a BFS like algorithm; however that would results in creation of a linked list like structure which was a bit weird compared to the lists that we use when coding in C. So, I decided to use a recursive approach. I first constructed a character array of size 12500 and I constructed a function named “recursive_node”. This function takes 3 arguments. First is the task_struct to process, second one is the information whether it is the root node or not and the last one is whether this process is the eldest process of its parent. The last 2 are used for writing process information to the created buffer (of size 12500) to be used when plotting the tree. At each call, the relevant information of the current process is written to output buffer. Then the eldest child of the current node is found, and for each child, the function is called recursively by giving the next task_struct together with the aforementioned properties. I arranged the output in such a format: “PID(int),PPID(int),isEldest(binary),CreationTimeWRTEpoch(unsigned long in seconds)”. Start node

being the “param_pid” parameter of the module, when all children are done being processed, the module writes this into the buffer provided by the reader process with the aid of kernel.
Now, let me explain the process side. Shellect loads the module to the kernel when the “pvis” command is firstly used. Then its only the original shellect (not its children) unloads the module when the user exits. When the module is loaded, shellect parses the log of the kernel by “dmesg” find the major number assigned to the module “mymodule” and then it creates an abstract device file using this major number. Then the pvis, first changes the parameter of the module by changing the content of “/sys/module/mymodule/parameters/param_pid” and then, it opens the device file created by the module loader function. It tries to read it and receives the necessary information from the module in this way. Then it closes the device file and parses the output to construct the data file which will be processed by the “gnuplot”. For the .gp script that visualizes the process tree, I used the script from this link from “pianshen.com” and changed a couple of lines to add a new column, managing the color, taking the input/output files from outside etc. Upon parsing and reorganizing the output from the module and saving it in a temporary data file, I execute “gnuplot -c plot.gp input_file out_image_name” where the last argument is provided by the shellect user.

```
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ pvis 2372 out.png
[sudo] password for fatih:
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$ pvis 2372 out2.png
fatih@host:/home/fatih/Desktop/SHARED/Assignments/Project1/project-1-shell-icardi Shellect$
```



Resources

- https://www.informit.com/articles/article.aspx?p=368650
- https://tldp.org/LDP/lkmpg/2.4/html/c577.htm
- https://docs.huihoo.com/doxygen/linux/kernel/3.7/structtask__struct.html
- https://pianshen.com/ask/980711850948/