COMP304 Operating Systems
Project 1: Shellgibi

Ceren Akkalyoncu 60145
İrem Arpag

15 March 2020

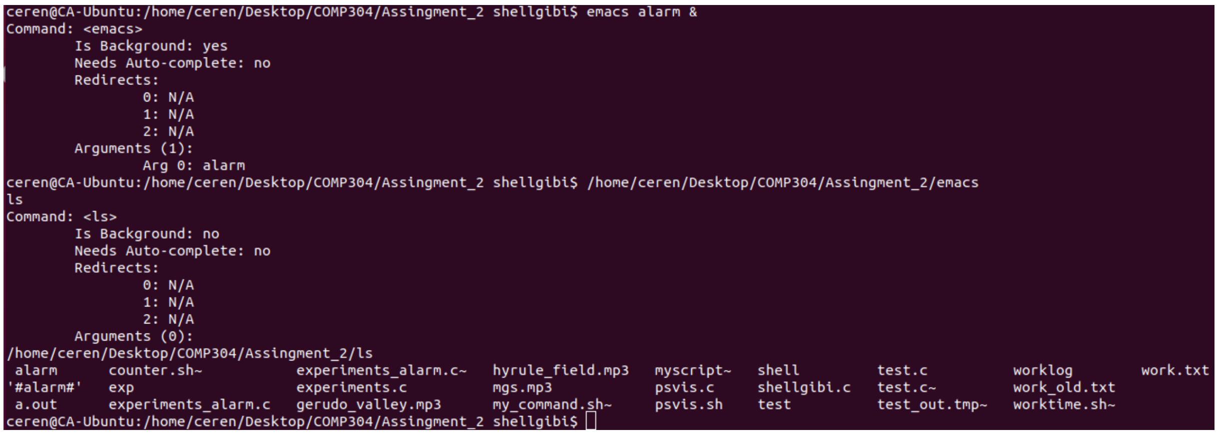
Introduction

In this project, we implemented a Shell-like Unix style operating system, Shellgibi. When we start Shellgibi, we enter a terminal like program where we can execute both the existing commands, such as cd and ls, and the newly defined commands we implemented. The given code handles getting and parsing the input from the user, tokenizing it to the command and its arguments.

Part 1: Shellgibi Implementation

We first worked on changing the `execvp` command, which was used in the original version, to `execv` to specify the command path. The `execvp` command can handle the execution of the existing commands as it can resolve the path. However, it cannot resolve the path to the builtin commands we implemented, therefore does not execute them. On the other hand, we specify the command path when using `execv` function, and are able to execute the user defined commands as well. To be able to resolve the path for all commands, we defined a `resolve_path` function, that takes in the name of the command, and tries to find the corresponding path. For this, we first check the current working directory, where I have implemented the builtin commands, and calls `execv` with the given path. The `execv` function does not return anything when it runs successfully, and -1 otherwise [1]. Using this feature, we can decide if the program call was successful or not. If it was not, we then search for the command's path where the existing commands are, under /bin/. We check if this program call was successful or not again, and try using the `execvp` command if not. The execvp command is only used when the specified command is a downloaded module, such as emacs. We have not addressed this issue in the resolve_path function as the downloaded modules may not exist on some other users; however for testing purposes, made such functions available through `execvp`.

Afterwards, we made sure that the Shellgibi system supports ampersand (&) operation. The system can handle background processes, by assigning a background tag to the process, and waiting for it to finish if it is not a background process, or continuing if it is. This way, we can execute multiple processes at the same time, without waiting for one to finish. The test we had with `emacs` and `ls` command run right after can be seen below. The `print_command` function was also un-commented to observe the background tags given to the process with ampersand.



```
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ emacs alarm &
Command: <emacs>
        Is Background: yes
        Needs Auto-complete: no
        Redirects:
                0: N/A
                1: N/A
                2: N/A
        Arguments (1):
                Arg 0: alarm
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ /home/ceren/Desktop/COMP304/Assingment_2/emacs
ls
Command: <ls>
        Is Background: no
        Needs Auto-complete: no
        Redirects:
                0: N/A
                1: N/A
                2: N/A
        Arguments (0):
/home/ceren/Desktop/COMP304/Assingment_2/ls
 alarm      counter.sh~           experiments_alarm.c~   hyrule_field.mp3   myscript~   shell         test.c          worklog       work.txt
'#alarm#'   exp                   experiments.c          mgs.mp3            psvis.c     shellgibi.c   test.c~         work_old.txt
 a.out      experiments_alarm.c   gerudo_valley.mp3      my_command.sh~     psvis.sh    test          test_out.tmp~   worktime.sh~
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ 
```

Figure 1: Background process with `emacs &`

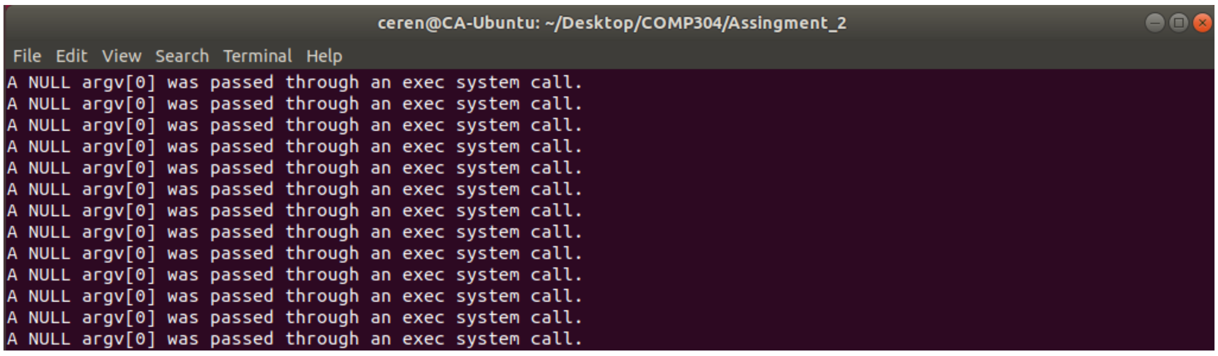
The program calls are handled by the child processes forked from the parent process when the user enters a command. This way, even after the child process is done, we can return back to the parent process and keep using the Shellgibi system.

Part 2

Piping

In this part, we tried to implement the piping functionality to our Shellgibi system. For this, we followed a recursive approach to handle simple pipe expressions as well as piped chains. The pipe symbol is already processed in the given code, and added as a `command->next` argument to the command data structure. We can interpret this as following, if a given command has a next, it is a piped command and therefore should be processed in a different fashion. In our `handle_pipe(command)` command, we create a pipe, and a child process. The child process calls `process_command` with the `command->next`, whereas the parent process calls it with `command`. When we tried to test this, it first did not invoke the commands after the pipe. Then after working on the issue for a while, it started to get in an infinite loop and the system crashed.

In the end, we could not manage to solve the issue with the pipes. The system crashed when we tried to execute a piped command. When we try running `*not existing command* | ls` or `ls | *not existing command*`, the first one returned the



```
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
A NULL argv[0] was passed through an exec system call.
```

Figure 2: Background process with `emacs &`

result of ls, and the second one gave an unknown command error. We may be missing a trivial point about the implementation that we can't recognize at this point.

Part 3

Auto-complete

We have spent a fair amount of time searching about this functionality, however, could not implement it in our code. The suggested methods to use were `rl_complete` method from the GNU readline library, and the following code snippet.

```
char *bash_argv[] = { "bash", "-i", 0 };
execv("/bin/bash", bash_argv);
```

This approach above has already been implemented as a part of the given code, and does not handle auto-completion but simply executes the command - acting like newline. I could not implement more complicated approaches I could think of.

Job management

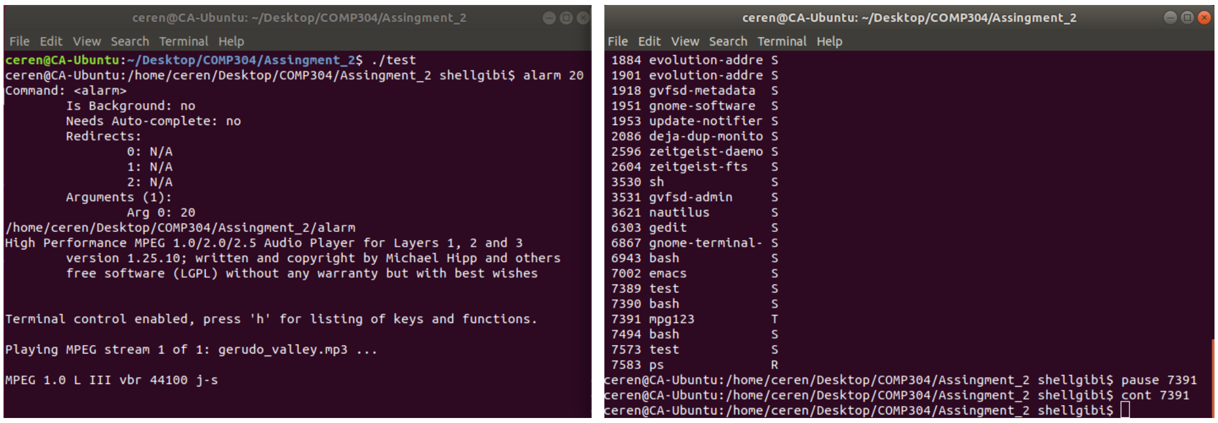
We also implemented simple job management tools, as described below.

- `myjobs` : List the processes owned by the current user
- `pause <PID>` : Pause the execution of a job given its pid.
- `mybg <PID>` : Bring the paused job to background and running state, given its pid.
- `myfg <PID>` : Bring the paused job to foreground and running state, given its pid.

For the `myjobs` command, we used the `ps -u user` command to get the list of the processes. We also applied formatting to the ps output to only display the required information -their pids, names and states- about the processes. The formatting can be done by giving additional arguments to `ps` in the form, `ps -o pid,state,comm`, as explained in the manual [2].

To pause a job given its pid, we used the `kill -SIGSTOP <PID>` command, and displayed the related information, such as its pid, name and the current state.

We first tested a simple `continue` command, using `kill -SIGCONT <PID>` on the `alarm` command, as seen below. When we call the `pause` command, we can pause the process, and continue later with `cont` from where we left off.


ceren@CA-Ubuntu: ~/Desktop/COMP304/Assingment_2
File Edit View Search Terminal Help
ceren@CA-Ubuntu: ~/Desktop/COMP304/Assingment_2
File Edit View Search Terminal Help

```
ceren@CA-Ubuntu:~/Desktop/COMP304/Assingment_2$ ./test
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ alarm 20
Command: <alarm>
        Is Background: no
        Needs Auto-complete: no
        Redirects:
                0: N/A
                1: N/A
                2: N/A
        Arguments (1):
                Arg 0: 20
/home/ceren/Desktop/COMP304/Assingment_2/alarm
High Performance MPEG 1.0/2.0/2.5 Audio Player for Layers 1, 2 and 3
        version 1.25.10; written and copyright by Michael Hipp and others
        free software (LGPL) without any warranty but with best wishes


Terminal control enabled, press 'h' for listing of keys and functions.

Playing MPEG stream 1 of 1: gerudo_valley.mp3 ...

MPEG 1.0 L III vbr 44100 j-s
```

```
 1884 evolution-addre S
 1901 evolution-addre S
 1918 gvfsd-metadata  S
 1951 gnome-software  S
 1953 update-notifier S
 2086 deja-dup-monito S
 2596 zeitgeist-daemo S
 2604 zeitgeist-fts   S
 3530 sh              S
 3531 gvfsd-admin     S
 3621 nautilus        S
 6303 gedit           S
 6867 gnome-terminal- S
 6943 bash            S
 7002 emacs           S
 7389 test            S
 7390 bash            S
 7391 mpg123          T
 7494 bash            S
 7573 test            S
 7583 ps              R
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ pause 7391
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ cont 7391
ceren@CA-Ubuntu:/home/ceren/Desktop/COMP304/Assingment_2 shellgibi$ 
```

Figure 3: Pausing/Continuing the `alarm` command

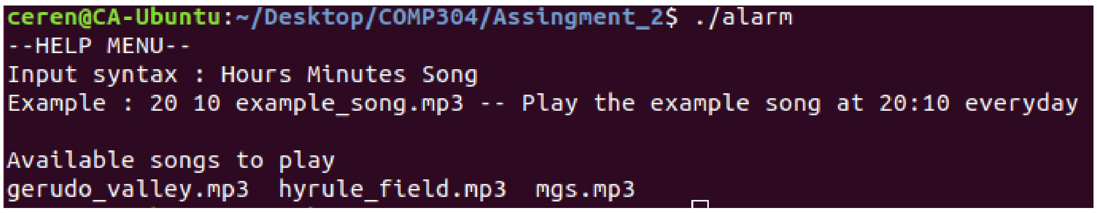
We could not implement the feature for continuing the job given its pid on background/ foreground, therefore we only use `cont` command for both.

Psvis Command

Not implemented. The command `psvis` simply executes the `pstree -u $USER` command.

Alarm

The alarm command is called in the form `shellgibi $ alarm hour minute song` (eg. `alarm 20 10 mgs.mp3` to play the song mgs.mp3 every day at 20:10). This alarm feature is supposed to be handled by adding a job to crontab via `crontab -e` [3], in the form `HOUR MINUTE * * * /pathto/playsong song.mp3`. I downloaded a module `mpg123` to play music from the terminal, and added several song options to be used as an alarm. The `alarm` command runs the help function when called without any arguments, which displays the necessary arguments and the possible songs to choose from. It expects at least the hour argument to start an alarm, and can work with the default minute and song values even when they are not specified.



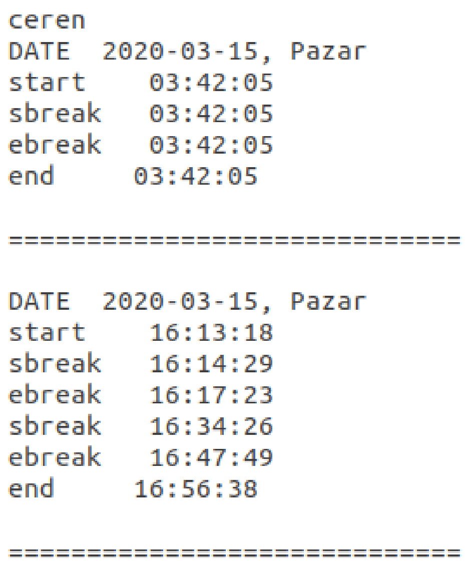
```
ceren@CA-Ubuntu:~/Desktop/COMP304/Assingment_2$ ./alarm
--HELP MENU--
Input syntax : Hours Minutes Song
Example : 20 10 example_song.mp3 -- Play the example song at 20:10 everyday

Available songs to play
gerudo_valley.mp3  hyrule_field.mp3  mgs.mp3
```

Figure 4: Help Menu for `alarm` command

However, I could not figure out how to schedule a job with crontab, without going into the editor. One option I thought of was to access the file `crontab -e` command directed me into and append the job as defined above, which I did not have access to. Therefore, the `alarm` command simply plays the given song, The scheduling feature in theory is described in the code as well.

Custom Command **`worklog`** - Ceren

The custom command `worklog` can be thought of as a tool to collect data on the working habits on oneself. It creates a worklog file on the time the user has spent working throughout the day. The users can start a day, which appends the current day and time values to the output text file. When they take a break, they can simply call this command and send start a break action, and end it when they come back. At the end of the day, they can send a close action, which ends the log for the current day.

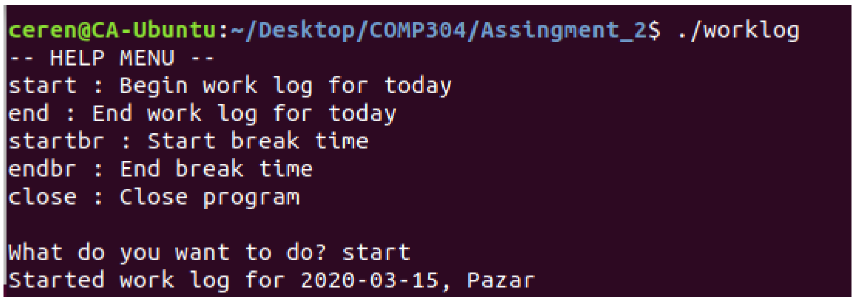


```
ceren
DATE  2020-03-15, Pazar
start    03:42:05
sbreak   03:42:05
ebreak   03:42:05
end     03:42:05

=============================

DATE  2020-03-15, Pazar
start    16:13:18
sbreak   16:14:29
ebreak   16:17:23
sbreak   16:34:26
ebreak   16:47:49
end     16:56:38

=============================
```

Figure 5: Example work.txt output

This command was initially designed as a way to track an employee's working habits (which can later be sent to a database), however it could also be used as a personal tool to analyze habits of oneself. This data can be later processed to acquire information about the total work time of a person, average time spent without taking a break etc.

The command takes the action to execute as an input and does something accordingly. The default action directs the user to the help menu, which explains the possible actions the user can select. Initially, if the target file is not created yet, it creates it and adds the current user's ID to the first line.



```
ceren@CA-Ubuntu:~/Desktop/COMP304/Assingment_2$ ./worklog
-- HELP MENU --
start : Begin work log for today
end : End work log for today
startbr : Start break time
endbr : End break time
close : Close program

What do you want to do? start
Started work log for 2020-03-15, Pazar
```

Figure 6: Options for `worklog` command

References

[1] "Exec(3) - Linux Manual." Exec(3) - Linux Manual Page, man7.org/linux/man-pages/man3/exec.3.html.

[2] "Ps(1) - Linux Manual." Ps(1) - Linux Manual Page, man7.org/linux/man-pages/man1/ps.1.html.

[3] "Linux Crontab Command Help and Examples." Computer Hope, 16 Nov. 2019, www.computerhope.com/unix/ucrontab.htm.