COMP 304

Spring 2020

Project 1: Shellgibi

Ahmet Uysal

Furkan Şahbaz

Contents

1 Introduction

Shellgibi is a UNIX-style operating system shell for Linux that is built using C programming language. It supports running all builtin Linux shell programs, user installed applications that are included in `PATH` and few additional builtin commands that are explained in later sections. It has auto-complete, piping, input and output redirection and background program execution capabilities.

Note: All parts of the project function as requested, however, the `$psvis` command output is printed into a text file, instead of an image.

2 Program Flow

When the **Shellgibi** is first initiated, it scans all directories that are listed under `PATH` environment variable and preserves them for later usage on auto-completion capability. Basic execution flow of **Shellgibi** consists of handling the user input with assisting capabilities, parsing the user input to its internal representation of the `command_t` struct, processing and executing the given command, and finally freeing the memory resources that are allocated for the execution of this command. Finally, **Shellgibi** frees the memory allocated to store all available command names on exit.

2.1 Parent & Child Process Responsibilities

```c
pid_t pid = fork();
if (pid == 0) {
    return process_command_child(command);
} else {
    if (!command->background || command->next) {
        waitpid(pid, NULL, 0); // wait for child process
    }
}
```

Code snippet 1: Sample process forking mechanism for command execution

A child process can be forked by the parent process in order to further complete the execution of a program. After forking the child, the command at hand can be passed to the child process by the parent process, while it waits for the child process' execution. **Shellgibi** supports both foreground and background command execution. In order to allow running of commands without blocking the shell, **Shellgibi** utilizes child processes to run the command while the parent process is either waiting for its child or continues to interact with the user.

Code Snippet 1 demonstrates this process in the simplest manner. In later sections, we will make some additions to this logic to support I/O redirection and piping.

3 Executing Linux Programs

Executing Linux programs require the `execv` command to be executed with the given program name including the full path, and the program arguments. In order to find the path for a given program, all directories listed under `PATH` were traversed, and the resulting paths that the program was found in were combined, in order to be passed into the `execv` command.

```c
int execv_command(struct command_t *command) {
    // command->args modifications are discarded for brevity
    char *path = getenv("PATH");
    char *path_tok = strtok(path, ":");
    while (path_tok != NULL) {
        char full_path[strlen(path_tok) + strlen(command->name) + 1];
        combine_path(full_path, path_tok, command->name);
        execv(full_path, command->args);
        path_tok = strtok(NULL, ":");
    }
    // If we reach here, we couldn't find the command on path
    printf("-%s: %s: command not found\n", sysname, command->name);
    return UNKNOWN;
}
```

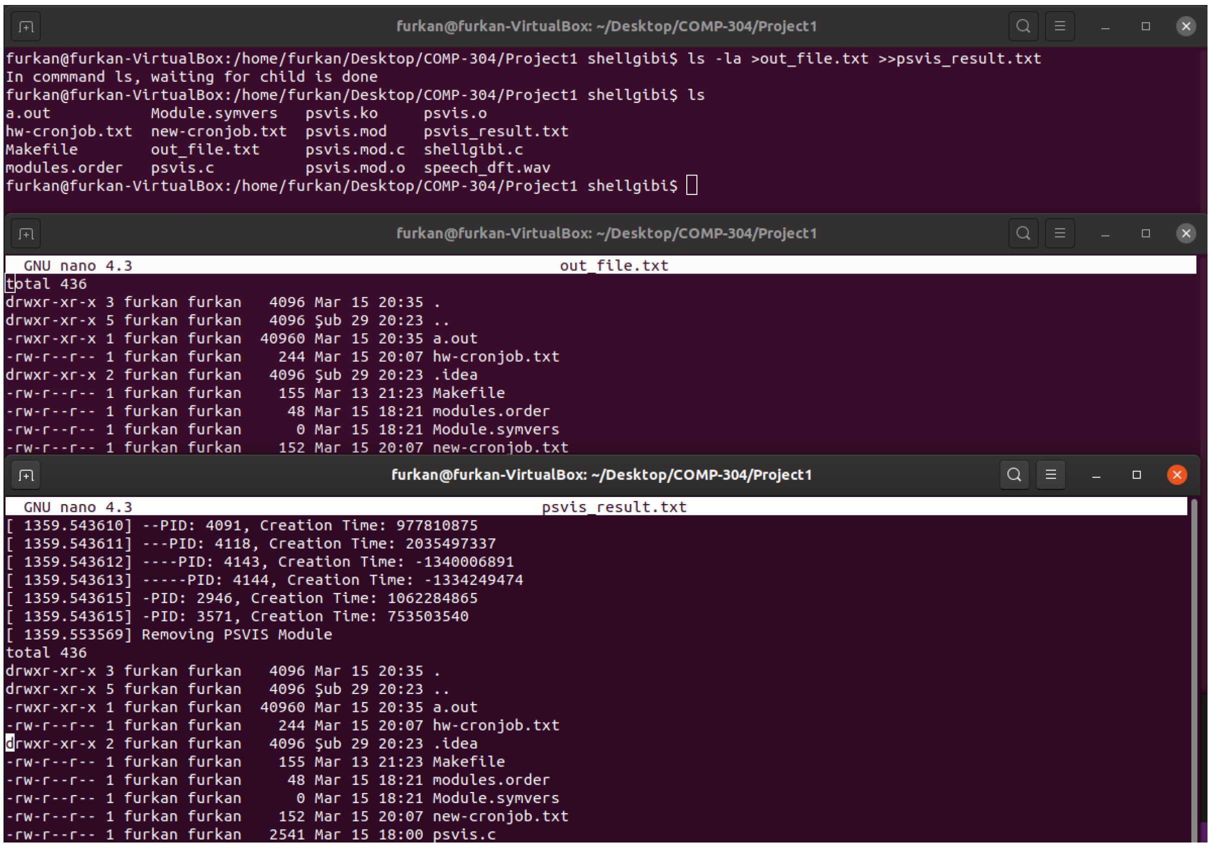
Code snippet 2: Linux program execution implementation using `execv`

4 I/O Redirection

After the command is determined to be redirected to one or more files, `stdout`, `stderr` and `stdin` file descriptors are overriden by the files with given names. However, **Shellgibi** supports two different output redirection mechanisms: truncate (>) and append (>>) modes, and these mechanisms can be used simultaneously. In this case, we needed to develop a new aproach since the `stdout` and `stderr` cannot be redirected to more than one file at once. To overcome this problem, we simply created a temporary file and redirected the `stdout` and `stderr` to this file. After the child process is finished, the parent simply copies the content of this temporary file to target locations in the requested writing modes (append or truncate). The temporary file is deleted once this process is completed.

```c
int process_command_child(struct command_t *command, const int *child_to_parent_pipe) {
    // <: input is read from a file
    if (command->redirects[0] != NULL)
        freopen(command->redirects[0], "r", stdin);
    // >: output is written to a file, in write mode
    if (!stdout_redirected_to_multiple_files && command->redirects[1] != NULL) {
        freopen(command->redirects[1], "w", stdout);
        freopen(command->redirects[1], "w", stderr);
    }
    // >>: output is written to a file, in append mode
    else if (!stdout_redirected_to_multiple_files && command->redirects[2] != NULL) {
        freopen(command->redirects[2], "a", stdout);
        freopen(command->redirects[2], "a", stderr);
    } else if (stdout_redirected_to_multiple_files) {
        char temp_filename[16 + 1];
        tmpnam(temp_filename);
        printf("File name: %s\n", temp_filename);
        pid_t pid2 = fork();
        if (pid2 == 0) {
            // grandchild
            FILE *tmp_file = fopen(temp_filename, "w");
            dup2(fileno(tmp_file), STDOUT_FILENO);
            return execute_command(command);
        } else {
            // wait for grandchild to finish and copy temp file to redirected files
            int status;
            waitpid(pid2, &status, 0);
            FILE *temp_fp = fopen(temp_filename, "r");
            FILE *trunc_fp = NULL, *append_fp = NULL;
            if (command->redirects[1] != NULL)
                trunc_fp = fopen(command->redirects[1], "w");
            if (command->redirects[2] != NULL)
                append_fp = fopen(command->redirects[2], "a");
            char buffer[BUFSIZ];
            size_t chars_read = 0;
            while ((chars_read = read(fileno(temp_fp), &buffer, sizeof(buffer))) > 0) {
                if (command->redirects[1] != NULL)
                    write(fileno(trunc_fp), &buffer, chars_read);
                if (command->redirects[2] != NULL)
                    write(fileno(append_fp), &buffer, chars_read);
            }
            fclose(temp_fp);
            if (command->redirects[1] != NULL)
                fclose(trunc_fp);
            if (command->redirects[2] != NULL)
                fclose(append_fp);
            remove(temp_filename);
            return SUCCESS;
        }
    }
}
```

Code snippet 3: I/O Redirection to file implementation



```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls -la >out_file.txt >>psvis_result.txt
In commmand ls, waiting for child is done
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls
a.out           Module.symvers   psvis.ko      psvis.o
hw-cronjob.txt  new-cronjob.txt  psvis.mod     psvis_result.txt
Makefile        out_file.txt     psvis.mod.c   shellgibi.c
modules.order   psvis.c          psvis.mod.o   speech_dft.wav
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$
```

```
  GNU nano 4.3                                                out_file.txt
total 436
drwxr-xr-x 3 furkan furkan   4096 Mar 15 20:35 .
drwxr-xr-x 5 furkan furkan   4096 Şub 29 20:23 ..
-rwxr-xr-x 1 furkan furkan  40960 Mar 15 20:35 a.out
-rw-r--r-- 1 furkan furkan    244 Mar 15 20:07 hw-cronjob.txt
drwxr-xr-x 2 furkan furkan   4096 Şub 29 20:23 .idea
-rw-r--r-- 1 furkan furkan    155 Mar 13 21:23 Makefile
-rw-r--r-- 1 furkan furkan     48 Mar 15 18:21 modules.order
-rw-r--r-- 1 furkan furkan      0 Mar 15 18:21 Module.symvers
-rw-r--r-- 1 furkan furkan    152 Mar 15 20:07 new-cronjob.txt
```

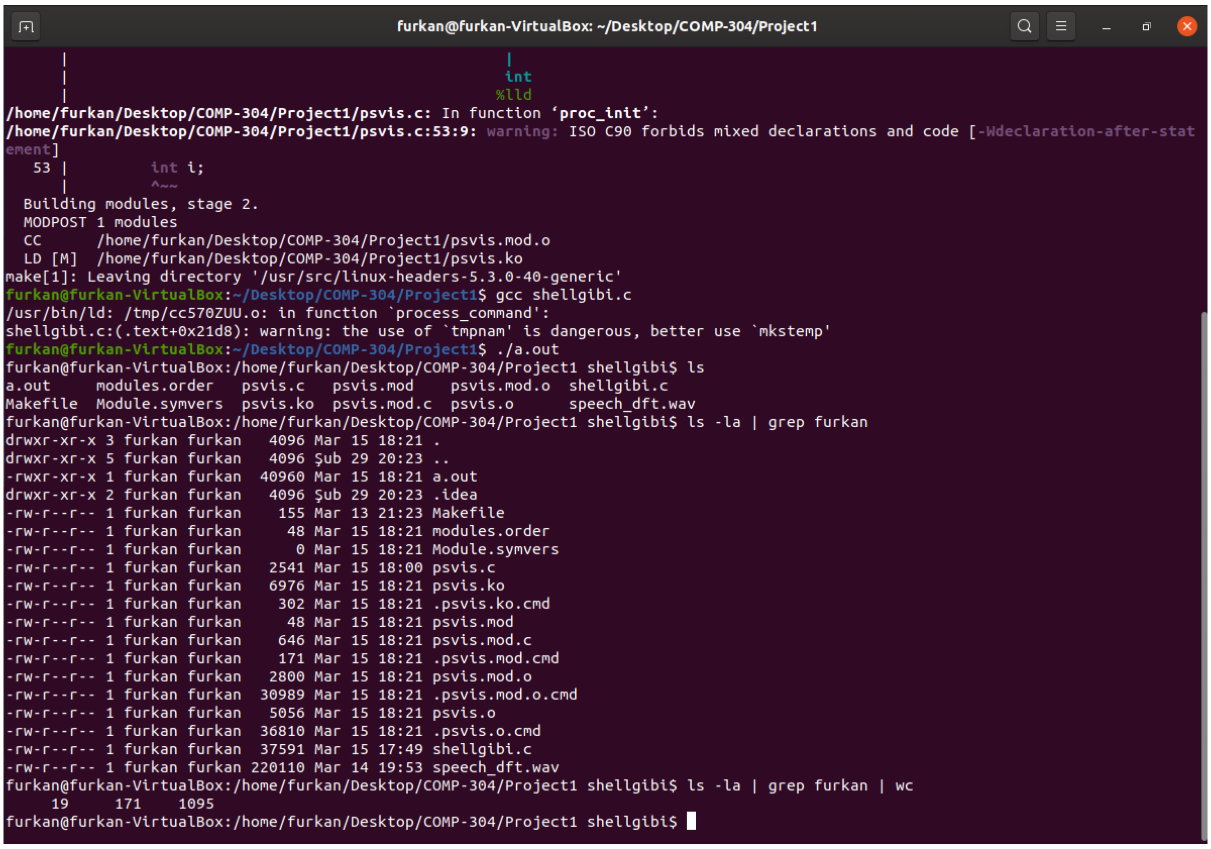
```
  GNU nano 4.3                                              psvis_result.txt
[ 1359.543610] --PID: 4091, Creation Time: 977810875
[ 1359.543611] ---PID: 4118, Creation Time: 2035497337
[ 1359.543612] ----PID: 4143, Creation Time: -1340006891
[ 1359.543613] -----PID: 4144, Creation Time: -1334249474
[ 1359.543615] -PID: 2946, Creation Time: 1062284865
[ 1359.543615] -PID: 3571, Creation Time: 753503540
[ 1359.553569] Removing PSVIS Module
total 436
drwxr-xr-x 3 furkan furkan   4096 Mar 15 20:35 .
drwxr-xr-x 5 furkan furkan   4096 Şub 29 20:23 ..
-rwxr-xr-x 1 furkan furkan  40960 Mar 15 20:35 a.out
-rw-r--r-- 1 furkan furkan    244 Mar 15 20:07 hw-cronjob.txt
drwxr-xr-x 2 furkan furkan   4096 Şub 29 20:23 .idea
-rw-r--r-- 1 furkan furkan    155 Mar 13 21:23 Makefile
-rw-r--r-- 1 furkan furkan     48 Mar 15 18:21 modules.order
-rw-r--r-- 1 furkan furkan      0 Mar 15 18:21 Module.symvers
-rw-r--r-- 1 furkan furkan    152 Mar 15 20:07 new-cronjob.txt
-rw-r--r-- 1 furkan furkan   2541 Mar 15 18:00 psvis.c
```

furkan@furkan-VirtualBox: ~/Desktop/COMP-304/Project1
furkan@furkan-VirtualBox: ~/Desktop/COMP-304/Project1
furkan@furkan-VirtualBox: ~/Desktop/COMP-304/Project1

Figure 1: I/O redirection example in **Shellgibi**

5 Program Piping

Program piping is similarly implemented by utilizing redirection of the `stdin`, `stdout`, and `stderr` file descriptors. **Shellgibi** supports arbitrary number of chaining operations. To implement program piping in a way that scales to any number of piping operations, we decided to assign the responsibility of transferring data between processes to the parent process. The parent process accomplishes this by using pipes. In the case of redirecting the output of a process to more than one destination (>, >> or |), parent process reads the output from the temporary file similar to I/O redirection logic.



furkan@furkan-VirtualBox: ~/Desktop/COMP-304/Project1

```
      |                                              |
      |                                              int
      |                                             %lld
/home/furkan/Desktop/COMP-304/Project1/psvis.c: In function ‘proc_init’:
/home/furkan/Desktop/COMP-304/Project1/psvis.c:53:9: warning: ISO C90 forbids mixed declarations and code [-Wdeclaration-after-stat
ement]
   53 |         int i;
      |         ^~~
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/furkan/Desktop/COMP-304/Project1/psvis.mod.o
  LD [M]  /home/furkan/Desktop/COMP-304/Project1/psvis.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-40-generic'
furkan@furkan-VirtualBox:~/Desktop/COMP-304/Project1$ gcc shellgibi.c
/usr/bin/ld: /tmp/cc570ZUU.o: in function `process_command':
shellgibi.c:(.text+0x21d8): warning: the use of `tmpnam' is dangerous, better use `mkstemp'
furkan@furkan-VirtualBox:~/Desktop/COMP-304/Project1$ ./a.out
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls
a.out     modules.order   psvis.c   psvis.mod    psvis.mod.o  shellgibi.c
Makefile  Module.symvers  psvis.ko  psvis.mod.c  psvis.o      speech_dft.wav
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls -la | grep furkan
drwxr-xr-x 3 furkan furkan   4096 Mar 15 18:21 .
drwxr-xr-x 5 furkan furkan   4096 Şub 29 20:23 ..
-rwxr-xr-x 1 furkan furkan  40960 Mar 15 18:21 a.out
drwxr-xr-x 2 furkan furkan   4096 Şub 29 20:23 .idea
-rw-r--r-- 1 furkan furkan    155 Mar 13 21:23 Makefile
-rw-r--r-- 1 furkan furkan     48 Mar 15 18:21 modules.order
-rw-r--r-- 1 furkan furkan      0 Mar 15 18:21 Module.symvers
-rw-r--r-- 1 furkan furkan   2541 Mar 15 18:00 psvis.c
-rw-r--r-- 1 furkan furkan   6976 Mar 15 18:21 psvis.ko
-rw-r--r-- 1 furkan furkan    302 Mar 15 18:21 .psvis.ko.cmd
-rw-r--r-- 1 furkan furkan     48 Mar 15 18:21 psvis.mod
-rw-r--r-- 1 furkan furkan    646 Mar 15 18:21 psvis.mod.c
-rw-r--r-- 1 furkan furkan    171 Mar 15 18:21 .psvis.mod.cmd
-rw-r--r-- 1 furkan furkan   2800 Mar 15 18:21 psvis.mod.o
-rw-r--r-- 1 furkan furkan  30989 Mar 15 18:21 .psvis.mod.o.cmd
-rw-r--r-- 1 furkan furkan   5056 Mar 15 18:21 psvis.o
-rw-r--r-- 1 furkan furkan  36810 Mar 15 18:21 .psvis.o.cmd
-rw-r--r-- 1 furkan furkan  37591 Mar 15 17:49 shellgibi.c
-rw-r--r-- 1 furkan furkan 220110 Mar 14 19:53 speech_dft.wav
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls -la | grep furkan | wc
     19     171    1095
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$
```

Figure 2: Piping example in **Shellgibi**

```c
int process_command_child(struct command_t *command, const int *child_to_parent_pipe) {
    ...
    else if (!stdout_redirected_to_multiple_files && command->next) {
        dup2(child_to_parent_pipe[1], STDOUT_FILENO);
        dup2(child_to_parent_pipe[1], STDERR_FILENO);
        close(child_to_parent_pipe[1]);
        return execute_command(command);
    }
    ...
}
int process_command(struct command_t *command, int parent_to_child_pipe[2]) {
    pid_t pid = fork();
    if (pid == 0) { // child
        if (parent_to_child_pipe != NULL)
            dup2(parent_to_child_pipe[0], STDIN_FILENO);
        return process_command_child(command, child_to_parent_pipe);
    } else { // parent site
        if (parent_to_child_pipe != NULL) {
            close(parent_to_child_pipe[0]);
            close(parent_to_child_pipe[1]);
        }
        if (have_child_to_parent_pipe)
            close(child_to_parent_pipe[1]);
        if (!command->background || command->next)
            waitpid(pid, NULL, 0); // wait for child process to finish
        if (command->next) {
            // transfer pipe data
            int argument_transfer_pipe[2];
            pipe(argument_transfer_pipe);
            char buffer[BUFSIZ];
            ssize_t read_chars;
            while ((read_chars = read(child_to_parent_pipe[0], &buffer, sizeof(buffer))) > 0) {
                write(argument_transfer_pipe[1], &buffer, read_chars);
            }
            close(child_to_parent_pipe[0]);
            close(argument_transfer_pipe[1]);
            return process_command(command->next, argument_transfer_pipe);
        }
        return SUCCESS;
    }
}
```

Code snippet 4: Program piping implementation

6 Auto-complete

To implement the auto-complete functionality, we directly changed the implementation of the `prompt` function.

6.1 Changes to Given `prompt` function and `command_t` struct

Initial skeleton program treated the tab character as one the characters which terminates the `prompt` function and stored this auto-complete request with a field named `auto_complete`. We chose to implement the auto-complete functionality directly inside the `prompt` function to make buffer management process easier to manage. Therefore, the `auto_complete` field became unnecessary and we removed it from the `command_t` struct.

```c
struct autocomplete_match {
    int match_count;
    char **matches;
};
```

Code snippet 5: `autocomplete_match` struct

```c
if (c == 9) { // handle tab
    if (index == 0)
        continue;
    char *buf_dup = strdup(buf);
    buf_dup[index] = '\0';
    struct autocomplete_match *match;
    int is_filename = should_complete_filename(buf_dup, filename_buf);
    if (is_filename) {
        match = filename_autocomplete(filename_buf);
        if (match->match_count == 1) {
            int input_filename_len = strlen(filename_buf);
            int match_len = strlen(match->matches[0]);
            if (match_len != input_filename_len) {
                for (int i = input_filename_len; i < match_len; i++) {
                    putchar(match->matches[0][i]); // echo the character
                    buf[index++] = match->matches[0][i];
                }
            }
            c = ' ';
        } else if (match->match_count > 1) {
            printf("\n");
            for (int i = 0; i < match->match_count; i++)
                printf("%s\t", match->matches[i]);
            printf("\n");
            show_prompt();
            printf("%s", buf);
        }
    } else {
        // auto complete command
        buf_dup = strdup(buf);
        buf_dup[index] = '\0';
        char *command_name = get_command_name(buf_dup);
        match = shellgibi_autocomplete(command_name);
        if (match->match_count == 1) {
            int input_command_len = strlen(command_name);
            int match_len = strlen(match->matches[0]);
            if (match_len != input_command_len) {
                for (int i = input_command_len; i < match_len; i++) {
                    putchar(match->matches[0][i]); // echo the character
                    buf[index++] = match->matches[0][i];
                }
            }
            c = ' ';
        } else if (match->match_count > 1) {
            printf("\n");
            for (int i = 0; i < match->match_count; i++)
                printf("%s\t", match->matches[i]);
            printf("\n");
            show_prompt();
            printf("%s", buf);
        }
    }
    free_autocomplete_match(match);
    if (c == 9)
        continue;
}
```

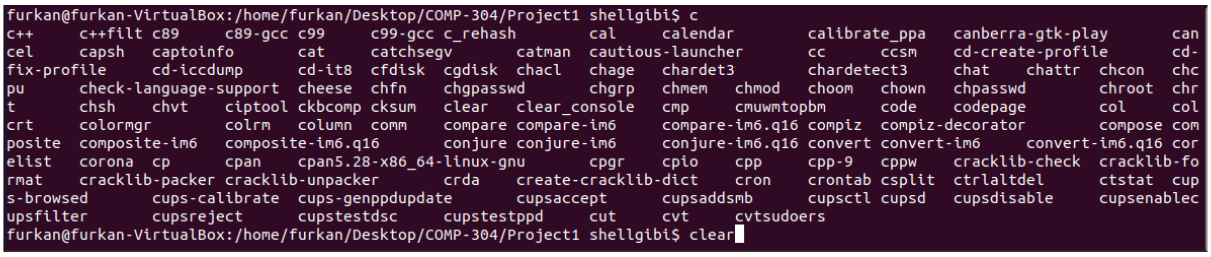
Code snippet 6: Handling of tabs inside `prompt` function

6.2 Gathering All Available Commands

In order to implement auto-complete, we need to gather all possible command names together. This is accomplished by traversing all directories that are included in `PATH` and putting all executable files together. However, since multiple directories could have executable with the same name, we need to remove duplicate filenames from the collection. This is implemented by sorting the filenames with `qsort` function and checking adjacent filenames for duplication.

6.3 Command Auto-completion

Once we have all available commands and the part user have entered so far, we can simply use `strncmp` function to check whether a filename matches this input. All matching filenames are put into a `autocomplete_match` struct and returned back to `prompt` function. `prompt` interprets this result as shown in code snippet 6.



```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ c
c++     c++filt c89     c89-gcc c99     c99-gcc c_rehash        cal     calendar        calibrate_ppa   canberra-gtk-play       can
cel     capsh   captoinfo       cat     catchsegv       catman  cautious-launcher       cc      ccsm    cd-create-profile       cd-
fix-profile     cd-iccdump      cd-it8  cfdisk  cgdisk  chacl   chage   chardet3        chardetect3     chat    chattr  chcon   chc
pu      check-language-support  cheese  chfn    chgpasswd       chgrp   chmem   chmod   choom   chown   chpasswd        chroot  chr
t       chsh    chvt    ciptool ckbcomp cksum   clear   clear_console   cmp     cmuwmtopbm      code    codepage        col     col
crt     colormgr        colrm   column  comm    compare compare-im6     compare-im6.q16 compiz  compiz-decorator        compose com
posite  composite-im6   composite-im6.q16       conjure conjure-im6     conjure-im6.q16 convert convert-im6     convert-im6.q16 cor
elist   corona  cp      cpan    cpan5.28-x86_64-linux-gnu       cpgr    cpio    cpp     cpp-9   cppw    cracklib-check  cracklib-fo
rmat    cracklib-packer cracklib-unpacker       crda    create-cracklib-dict    cron    crontab csplit  ctrlaltdel      ctstat  cup
s-browsed       cups-calibrate  cups-genppdupdate       cupsaccept      cupsaddsmb      cupsctl cupsd   cupsdisable     cupsenablec
upsfilter       cupsreject      cupstestdsc     cupstestppd     cut     cvt     cvtsudoers
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ clear
```

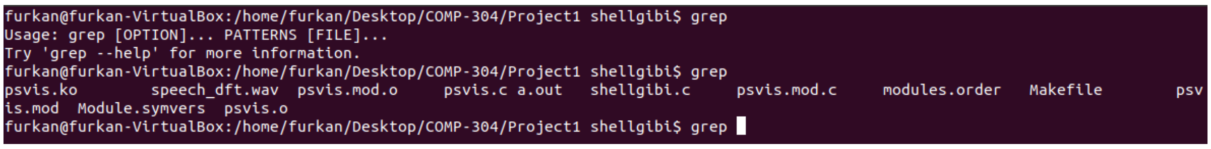
Figure 3: Auto-completion in **Shellgibi**

6.4 Deciding Between Filename and Command Auto-completion

Since **Shellgibi** supports auto-completion of both commands and filenames, we need to decide whether it should consider commands or filenames for the given buffer. To accomplish this, **Shellgibi** first checks whether the given buffer has at least one completed word and returns `true` since that means user has already entered the command name. It also determines which prefix should be searched for filenames by getting the last token. Otherwise, it returns `false` and **Shellgibi** auto-completes the command instead.

6.5 Filename Auto-completion

For filename auto-completion, **Shellgibi** opens the current directory by calling `DIR *directory = opendir(".")` and compares the given filename prefix with each file in this directory. All matches are returned inside a `autocomplete_match` struct instance.



```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ grep
Usage: grep [OPTION]... PATTERNS [FILE]...
Try 'grep --help' for more information.
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ grep
psvis.ko        speech_dft.wav  psvis.mod.o     psvis.c a.out   shellgibi.c     psvis.mod.c     modules.order   Makefile        psv
is.mod  Module.symvers  psvis.o
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ grep
```

Figure 4: Auto file completion in **Shellgibi**

7 Builtin Commands

7.1 `$myjobs`

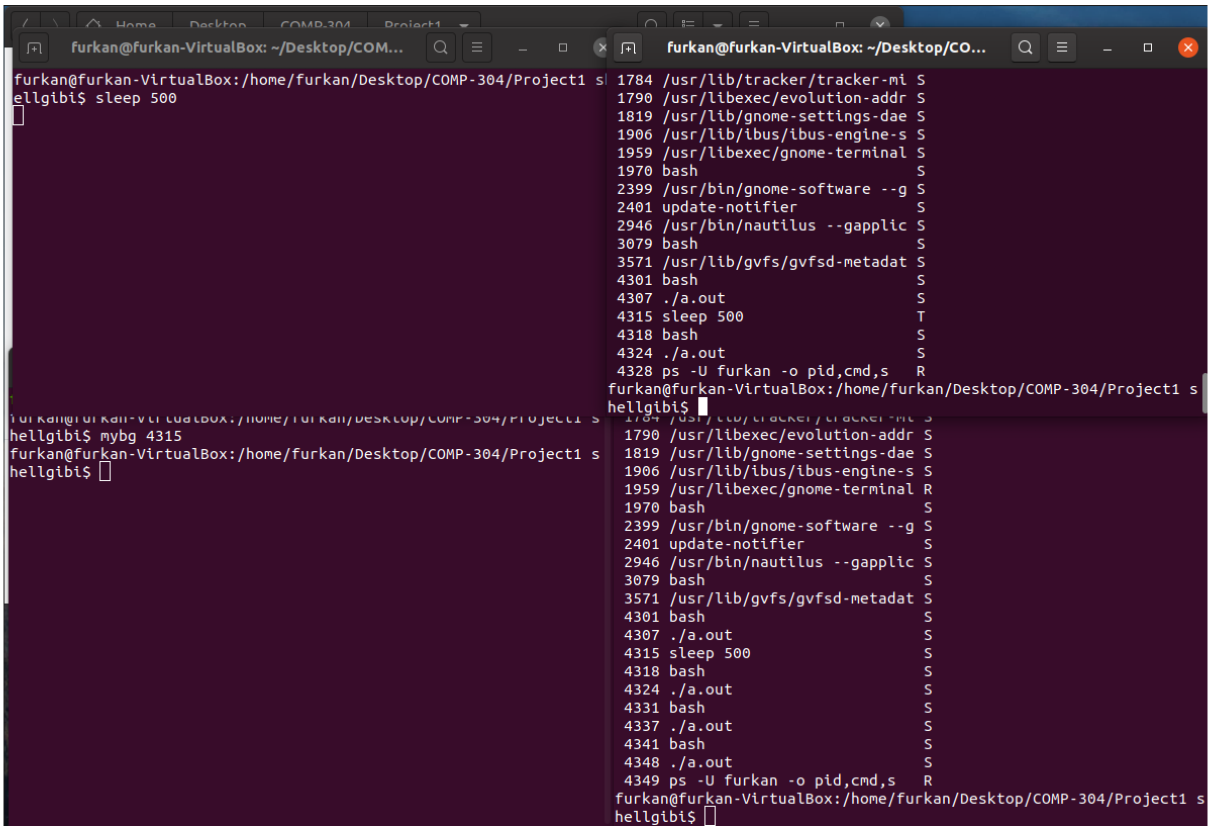
`$myjobs` utilizes the ps command in order to list the processes owned by the current user. In order to output the PID's, names, and statuses of such processes owned by the user, a command that contains the arguments of '-U', the current user name, '-o', and 'pid,cmd,s' were passed along with the ps command.

7.2 `$pause`

`$pause` command sends the SIGSTOP signal to the process with the given PID in order to stop its execution.

7.3 `$mybg`

`$mybg` command simply sends the SIGCONT signal to the process with the given PID, in order for it to resume operation. Parent process does not wait for the process with the given PID since it is requested to execute in background.


furkan@furkan-VirtualBox: ~/Desktop/COM...
furkan@furkan-VirtualBox: ~/Desktop/CO...

```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 s
ellgibi$ sleep 500
```

```
hellgibi$ mybg 4315
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 s
hellgibi$
```

```
1784 /usr/lib/tracker/tracker-mi S
1790 /usr/libexec/evolution-addr S
1819 /usr/lib/gnome-settings-dae S
1906 /usr/lib/ibus/ibus-engine-s S
1959 /usr/libexec/gnome-terminal S
1970 bash                        S
2399 /usr/bin/gnome-software --g S
2401 update-notifier             S
2946 /usr/bin/nautilus --gapplic S
3079 bash                        S
3571 /usr/lib/gvfs/gvfsd-metadat S
4301 bash                        S
4307 ./a.out                     S
4315 sleep 500                   T
4318 bash                        S
4324 ./a.out                     S
4328 ps -U furkan -o pid,cmd,s   R
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 s
hellgibi$
```

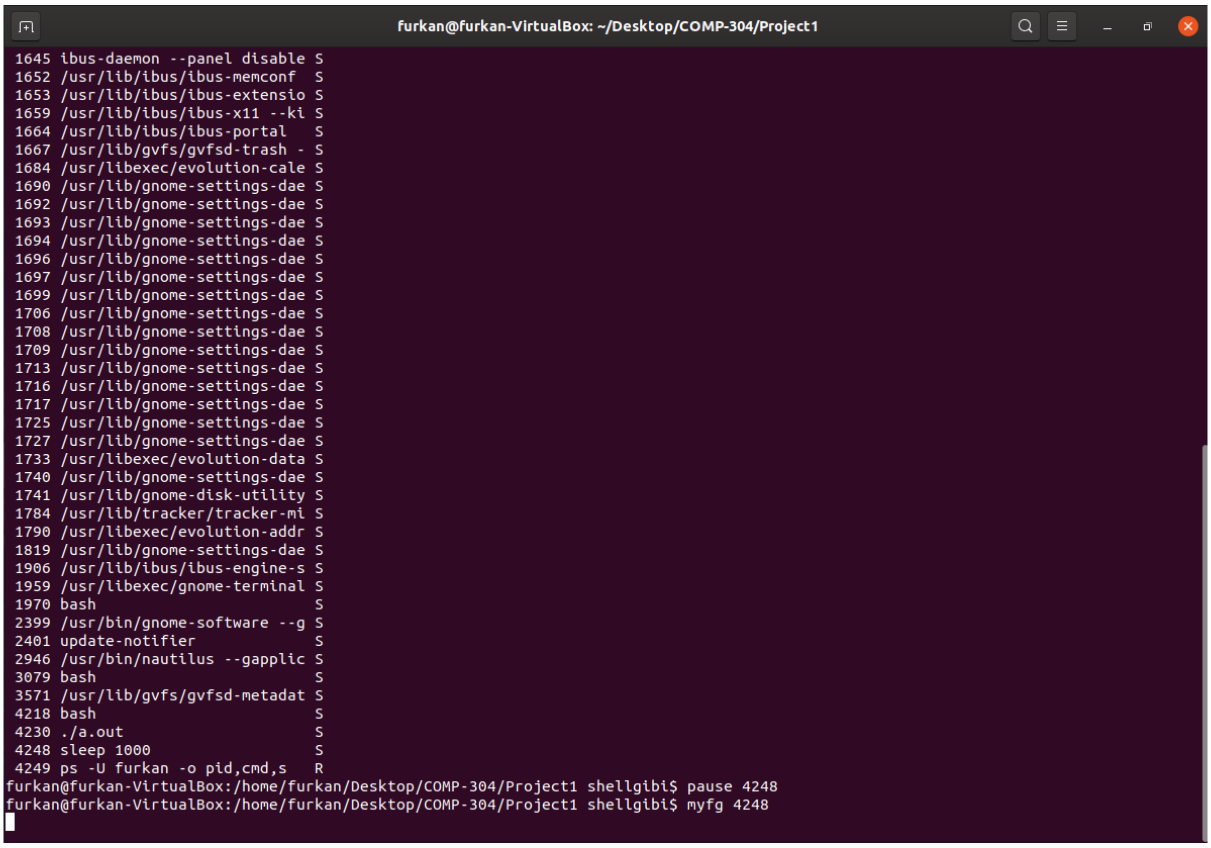
```
1790 /usr/libexec/evolution-addr S
1819 /usr/lib/gnome-settings-dae S
1906 /usr/lib/ibus/ibus-engine-s S
1959 /usr/libexec/gnome-terminal R
1970 bash                        S
2399 /usr/bin/gnome-software --g S
2401 update-notifier             S
2946 /usr/bin/nautilus --gapplic S
3079 bash                        S
3571 /usr/lib/gvfs/gvfsd-metadat S
4301 bash                        S
4307 ./a.out                     S
4315 sleep 500                   S
4318 bash                        S
4324 ./a.out                     S
4331 bash                        S
4337 ./a.out                     S
4341 bash                        S
4348 ./a.out                     S
4349 ps -U furkan -o pid,cmd,s   R
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 s
hellgibi$
```

Figure 5: Sample `$mybg` execution in **Shellgibi**

A sample execution of the `$mybg` command can be seen in Figure 5. A sleep job was first created which, in another terminal, is stopped with the `$pause` command. The resulting output from the `$myjobs` command demonstrates that the sleep job with PID 4315 has been stopped. After executing the `$mybg` command with the PID of 4315, another `$myjobs` output is displayed, showing that the `$sleep` process is back in execution.

7.4 `$myfg`

Similar to `$mybg`, `$myfg` command sends the SIGCONT signal to the process with the given to PID to resume its operation if it had stopped. To bring the process to foreground, parent process' execution needs to be blocked until the process with the given PID completes its execution. However, as the given PID may not belong to a process that was forked by **Shellgibi**, we cannot use the `waitpid` function. Therefore, the process with the given PID is constantly checked with a buffering while loop that sends the signal '0' to that process to check whether it still exists or not, as that process may still be executing. After the process' execution is complete, or finding out that the process does not exist, the buffering loop is exited, and the **Shellgibi** continues its usual execution flow.



```
 1645 ibus-daemon --panel disable S
 1652 /usr/lib/ibus/ibus-memconf  S
 1653 /usr/lib/ibus/ibus-extensio S
 1659 /usr/lib/ibus/ibus-x11 --ki S
 1664 /usr/lib/ibus/ibus-portal   S
 1667 /usr/lib/gvfs/gvfsd-trash - S
 1684 /usr/libexec/evolution-cale S
 1690 /usr/lib/gnome-settings-dae S
 1692 /usr/lib/gnome-settings-dae S
 1693 /usr/lib/gnome-settings-dae S
 1694 /usr/lib/gnome-settings-dae S
 1696 /usr/lib/gnome-settings-dae S
 1697 /usr/lib/gnome-settings-dae S
 1699 /usr/lib/gnome-settings-dae S
 1706 /usr/lib/gnome-settings-dae S
 1708 /usr/lib/gnome-settings-dae S
 1709 /usr/lib/gnome-settings-dae S
 1713 /usr/lib/gnome-settings-dae S
 1716 /usr/lib/gnome-settings-dae S
 1717 /usr/lib/gnome-settings-dae S
 1725 /usr/lib/gnome-settings-dae S
 1727 /usr/lib/gnome-settings-dae S
 1733 /usr/libexec/evolution-data S
 1740 /usr/lib/gnome-settings-dae S
 1741 /usr/lib/gnome-disk-utility S
 1784 /usr/lib/tracker/tracker-mi S
 1790 /usr/libexec/evolution-addr S
 1819 /usr/lib/gnome-settings-dae S
 1906 /usr/lib/ibus/ibus-engine-s S
 1959 /usr/libexec/gnome-terminal S
 1970 bash                        S
 2399 /usr/bin/gnome-software --g S
 2401 update-notifier             S
 2946 /usr/bin/nautilus --gapplic S
 3079 bash                        S
 3571 /usr/lib/gvfs/gvfsd-metadat S
 4218 bash                        S
 4230 ./a.out                     S
 4248 sleep 1000                  S
 4249 ps -U furkan -o pid,cmd,s   R
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ pause 4248
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ myfg 4248
```

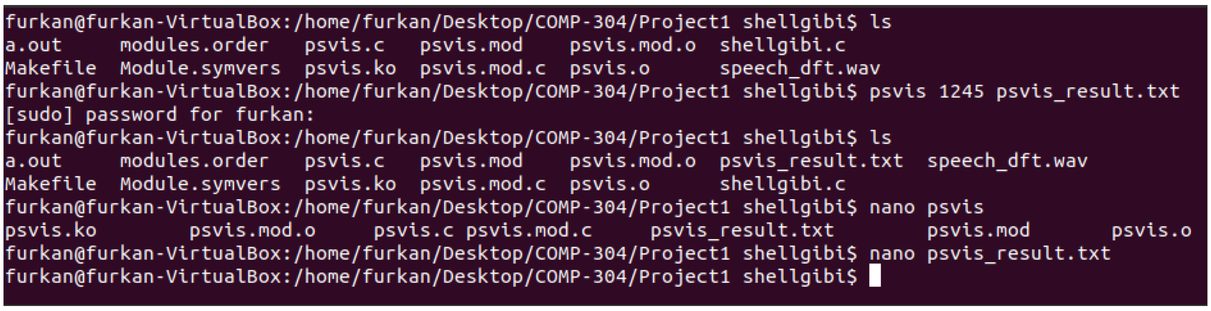
Figure 6: Sample `$myfg` execution in **Shellgibi**

Figure 6 shows an example execution of the `$myfg` command. A sleep job was initially created in the background, by using the & indicator (can be observed executing in the background, with the PID of 4248 in Fig. 2). After the process is called to be executed in the foreground, the **Shellgibi** output also keeps sleeping, as the figure demonstrates.

7.5 `$psvis`

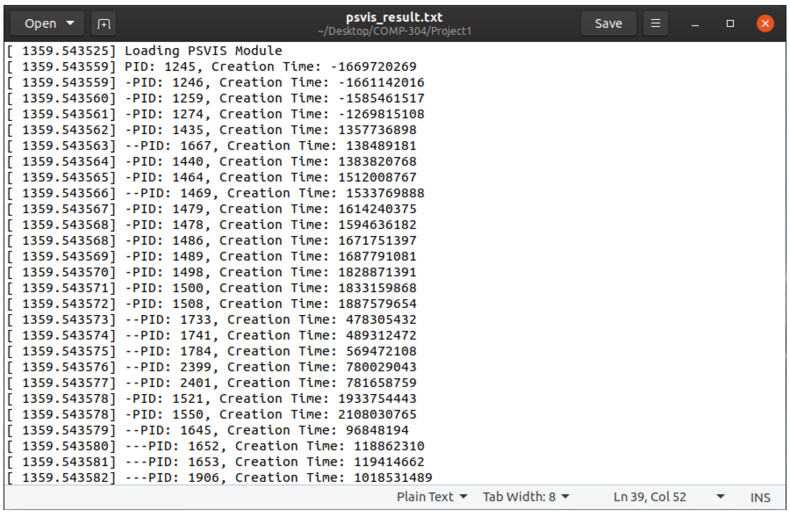
In order to perform the `$psvis` operation, a Kernel module named psvis was first developed. Within the module, all processes that are linked to the process with the given PID are traversed, collected, and printed to the Kernel ring buffer by taking advantage of task_struct. During the traversal, each child process is printed with its PID and creation time according to the depth it's located in (in order to preserve the parent-child relationship among processes). Within **Shellgibi**, the `psvis` command simply triggers the psvis.ko module by providing the given PID to the module while executing the `insmod` command to load the module; and then removes the module with the `rmmod` command.

After the module has been successfully removed, `dmesg` command is executed by redirecting its output to be written to a file with a user-specified name.



```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls
a.out     modules.order   psvis.c   psvis.mod    psvis.mod.o  shellgibi.c
Makefile  Module.symvers  psvis.ko  psvis.mod.c  psvis.o      speech_dft.wav
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ psvis 1245 psvis_result.txt
[sudo] password for furkan:
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ ls
a.out     modules.order   psvis.c   psvis.mod    psvis.mod.o  psvis_result.txt  speech_dft.wav
Makefile  Module.symvers  psvis.ko  psvis.mod.c  psvis.o      shellgibi.c
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ nano psvis
psvis.ko        psvis.mod.o     psvis.c psvis.mod.c     psvis_result.txt        psvis.mod       psvis.o
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ nano psvis_result.txt
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ 
```

Figure 7: Sample `$psvis` execution in **Shellgibi**



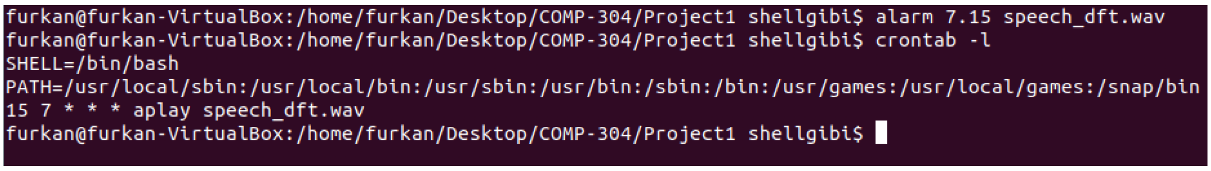
```
[ 1359.543525] Loading PSVIS Module
[ 1359.543559] PID: 1245, Creation Time: -1669720269
[ 1359.543559] -PID: 1246, Creation Time: -1661142016
[ 1359.543560] -PID: 1259, Creation Time: -1585461517
[ 1359.543561] -PID: 1274, Creation Time: -1269815108
[ 1359.543562] -PID: 1435, Creation Time: 1357736898
[ 1359.543563] --PID: 1667, Creation Time: 138489181
[ 1359.543564] -PID: 1440, Creation Time: 1383820768
[ 1359.543565] -PID: 1464, Creation Time: 1512008767
[ 1359.543566] --PID: 1469, Creation Time: 1533769888
[ 1359.543567] -PID: 1479, Creation Time: 1614240375
[ 1359.543568] -PID: 1478, Creation Time: 1594636182
[ 1359.543568] -PID: 1486, Creation Time: 1671751397
[ 1359.543569] -PID: 1489, Creation Time: 1687791081
[ 1359.543570] -PID: 1498, Creation Time: 1828871391
[ 1359.543571] -PID: 1500, Creation Time: 1833159868
[ 1359.543572] -PID: 1508, Creation Time: 1887579654
[ 1359.543573] --PID: 1733, Creation Time: 478305432
[ 1359.543574] --PID: 1741, Creation Time: 489312472
[ 1359.543575] --PID: 1784, Creation Time: 569472108
[ 1359.543576] --PID: 2399, Creation Time: 780029043
[ 1359.543577] --PID: 2401, Creation Time: 781658759
[ 1359.543578] -PID: 1521, Creation Time: 1933754443
[ 1359.543578] -PID: 1550, Creation Time: 2108030765
[ 1359.543579] --PID: 1645, Creation Time: 96848194
[ 1359.543580] ---PID: 1652, Creation Time: 118862310
[ 1359.543581] ---PID: 1653, Creation Time: 119414662
[ 1359.543582] ---PID: 1906, Creation Time: 1018531489
```

Figure 8: Resulting file from the `$psvis` execution in **Shellgibi**

An example execution of the `$psvis` command can be observed in Figure 7. The process that is wanted to be inspected with its relative processes is given with its PID 1245 in this case, where the output file is provided as psvis_result.txt. As the psvis command is being executed, it triggers the psvis.ko module with the input parameter PID=1245, then removes it with the rmmod command. After the module removal, the output of the dmesg command is redirected into the psvis_result.txt file, as requested by the user. The outputting parent-child relations can be observed in Figure 8, which demonstrates the relative distances with '-' as the relative depth increases.

7.6 `$alarm`

The `$alarm` command takes the time the user wants to set an alarm for, and a music source file that will be played at that time. Initially the time value provided by the user is split into its hour and minute values. Afterwards, a crontab file is generated by utilizing these values along with the provided music source file, in order to execute the aplay command at the given time every day, every week, and every month. The following line, "m h * * * aplay music_file" enables the crontab scheduler to do so, where h.m and music_file are the arguments provided by the user. After the crontab file was generated, the crontab command was simply executed with the generated file passed as an argument to it.



```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ alarm 7.15 speech_dft.wav
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ crontab -l
SHELL=/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
15 7 * * * aplay speech_dft.wav
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$
```

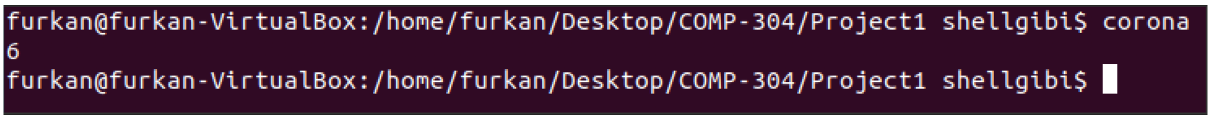
Figure 9: Sample `$alarm` execution in **Shellgibi**

As Figure 9 demonstrates, the `$alarm` command simply places the time value and the music file name in a crontab job file, that will play the given music file every day at the given time.

8 Custom Commands

8.1 Ahmet - `$corona`: Number of COVID-19 Cases in Turkey

We are truly terrified of the progression of coronavirus outbreak. We wanted to add a way to follow the number of patients in Turkey without leaving our comfort zone: shell. `$corona` command prints out the current number of verified COVID-19 cases in Turkey. `$corona` command does not require any arguments, if any arguments are passed to this command it simply ignores them and functions normally.
`$corona` is implemented by using two builtin Linux commands: `$wget` and `$grep`. When the `$corona` is called, the website https://www.worldometers.info/coronavirus/ is downloaded to a tempory file using `$wget`. Then, `$grep` command is used to extract the result from the downloaded html file using the regular expression `"<td[^>]> Turkey </td>( s*)<td[^>]*>\K[0-9]*(?=</td>)"`.

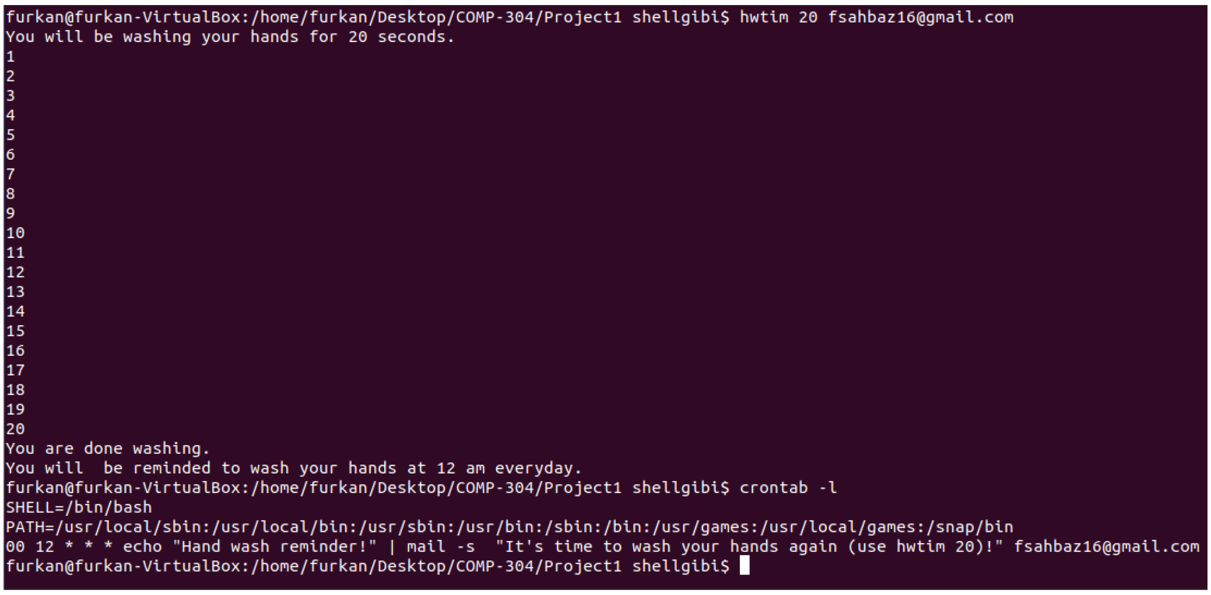


```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ corona
6
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$
```

Figure 10: Sample `$alarm` execution in **Shellgibi**

8.2 Furkan - `$hwtim`: A hand wash timer that can set a daily reminder

`$hwtim` command has two modes of operation: only hand wash timer and a daily hand wash reminder service via email after the timer. If it has only one argument provided, that is the number of seconds to count for the hand wash, it iterates until it reaches the given value by sleeping for 1 second in each iteration. On the other hand, if it has two arguments provided, the first one is again used for the timer, and the second one is the email address that a `$crontab` job will be scheduled to send an email to remind the user to wash their hands at 12 am everyday. This time, the `$crontab` file has the line "00 12 * * * echo "It's time to wash your hands again (use hwtim 20)!" | mail -s "Hand wash reminder!" [email_address]", enabling the `$crontab` scheduler to execute the mail command with the given parameters everyday at 12 am to send a reminder email with the given content.



```
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ hwtim 20 fsahbaz16@gmail.com
You will be washing your hands for 20 seconds.
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
You are done washing.
You will  be reminded to wash your hands at 12 am everyday.
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ crontab -l
SHELL=/bin/bash
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
00 12 * * * echo "Hand wash reminder!" | mail -s  "It's time to wash your hands again (use hwtim 20)!" fsahbaz16@gmail.com
furkan@furkan-VirtualBox:/home/furkan/Desktop/COMP-304/Project1 shellgibi$ 
```

Figure 11: Sample `$hwtim` execution in **Shellgibi**

Figure 11 demonstrates an example execution of the hwtim command. When the email address is entered along with the value the timer counts up to, a crontab job is scheduled to send a reminder email to the given address, daily at 12 am.