**Report**
**Project I Report**

**Part I**

This part is dealing with running system commands. Here, there are two variants. In the first one, the command is residing in current directory, and there is a ./ at the beginning of it. If this is the case, the ./ is removed and the command is run using execv. In the second one, the command resides in one of the directories where system commands are located. In order to run such a command, we need to iterate through the candidate folders and try to see if there is an executable with the proper name.

```
if (command->args[0][0] == '.')
{
  command->args[0] += 2;
  execv(command->args[0], command->args);
}
else
{
  char *general_path = getenv("PATH");
  char *token = strtok(general_path, ":");
  int len2 = strlen(command->args[0]);
  while (true)
  {
    int len1 = strlen(token);
    char *command_path = malloc(len1 + len2 + 2);
    strcpy(command_path, token);
    strcat(command_path, "/");
    strcat(command_path, command->args[0]);
    execv(command_path, command->args);
    free(command_path);
    token = strtok(NULL, ":");
    if (token == NULL) {break;}
  }
  free(general_path);
}
```

The implementation of this part is inspired from multiple internet sources. We have written our own code snippet for this purpose, which looks pretty similar to the other implementations we've encountered since the search for the system commands inevitably requires a loop to iterate over the tokenized folders.

**Part II**

a) Redirection: This part is handled by checking the redirection fields of the received command's struct and connecting the redirection file with the stdin/stdout file with dup.

```
if (command->redirects[0] != NULL) {
  dup2(fileno(fopen(command->redirects[0],"r")),STDIN_FILENO);
}
if (command->redirects[1] != NULL) {
  dup2(fileno(fopen(command->redirects[1],"w")),STDOUT_FILENO);
}
if (command->redirects[2] != NULL) {
  dup2(fileno(fopen(command->redirects[2],"a")),STDOUT_FILENO);
}
```

b) Piping: This part is ensured by checking each command recursively to see if it has a piped next command.

```
pid_t pid = fork();
if (pid == 0) // child
{
  while (command->next != NULL) {
    int fd[2];
    int pid_2;
    if (pipe(fd) < 0) {fprintf(stderr,"Pipe failed.");}
    pid_2 = fork();
    if (pid_2 == 0) {
      dup2(fd[1], STDOUT_FILENO);
      close(fd[0]);
      run_command(command);
      exit(0);
    } else {
      wait(NULL);
      close(fd[1]);
      dup2(fd[0],STDIN_FILENO);
      command = command->next;
    }

  }
  run_command(command);
  exit(0);
```

**Part III**

**a) Uniq:** This command takes input as sorted lines. The aim of Uniq is to print all of the unique elements, and their number of occurences (if -c parameter is given.). To do

this, our shellax invokes a function called uniq, when the command→name is “uniq”. Then given input is collected under arr array, which is a dynamic array created by realloc. Then we have two char pointers for iterating over the input. Since it is sorted, we can use them back-to-back, which means when a new word is shown, we can assume that the previous words will not be shown again. Same logic applies to counting, every time a new word arrives, a new index in int* occurrence is placed and initialized as 1 (since it cannot be 0 for a word). If next word is same, then that specific index is increased by one.

```
479 void uniq(struct command_t *command) {
480   char *str = NULL;
481   size_t size = 0;
482   size_t lines = 0;
483   ssize_t len= 0;
484   char** arr = NULL;
485   char** distinct_strings = NULL;
486
487   while ((len= getline(&str,&size, stdin)) != -1) {
488     arr = realloc(arr, sizeof * arr * lines + 1);
489     arr[lines++] = strdup(str);
490   }
491
492   int* occurence= NULL;
493   char* new;
494   char* old = "random_string";
495   int num_results = 0;
496   int counter = -1;
497   int lines2 = 0;
498
499   for (int i = 0; i < lines; i++) {
500     new = arr[i];
501     if (strcmp(new,old) != 0) {
502       old = new;
503       distinct_strings = realloc(distinct_strings, sizeof * distinct_strings * lines2 + 1);
504       distinct_strings[lines2++] = strdup(old);
505       num_results++;
506       counter++;
507       occurence = realloc(occurence, sizeof * occurence * counter +1);
508       occurence[counter] = 1;
509     } else { occurence[counter] += 1;}
510   }
511
512   for (int k = 0; k < num_results; k++) {
513     if (command->arg_count) {
514       if (strcmp(command->args[0],"-c")==0) {printf("%d ",occurence[k]);}
515     }
516     printf("%s",distinct_strings[k]);
517   }
518 }
519
```

```
hsenol@hsenol:/home/hsenol/Desktop/github/COMP304-Project-I shellax$ sort <input.txt | uniq
Cinnamon
Egg
Flour
Milk
hsenol@hsenol:/home/hsenol/Desktop/github/COMP304-Project-I shellax$ sort <input.txt | uniq -c
1 Cinnamon
2 Egg
2 Flour
3 Milk
```

**b) Chatroom:** This command is implemented by creating a child for each user. The child keeps reading the named pipe that is allocated to the user and the parent keeps receiving input from the user in order to write it to other named pipes in the room’s directory. Note that parent does not fork a new child for each new user in the room

because this is very much unnecessary (Doing so requires adding 3-4 more lines in the parent's for loop and is easy to do, but I choose not to do it. This one works better and requires less system resources).

```
} else {
  int fd_parent;
  char file_name[256];
  struct dirent *de;
  DIR *dr;
  while (1) {
    strcpy(buff,"[");
    strcat(buff,command->args[0]);
    strcat(buff,"] ");
    strcat(buff,command->args[1]);
    strcat(buff," > ");
    printf("%s ", buff);
    fflush(stdout);
    scanf("%s",msg);
    if (strcmp(msg,"exit")==0) {break;}
    strcat(buff, msg);
    strcat(buff,"\0");
    dr = opendir(room_name_buf);
    while ((de = readdir(dr)) != NULL) {
      if ((isalpha(de->d_name[0]) != 0) && (strcmp(de->d_name,command->args[1])!=0)) {
        strcpy(file_name, room_name_buf);
        strcat(file_name, "/");
        strcat(file_name, de->d_name);
        fd_parent = open(file_name, O_NONBLOCK | O_WRONLY);
        write(fd_parent, buff, strlen(buff));
        close(fd_parent);
      }
      file_name[0] = '\0';
    }
    closedir(dr);
    buff[0] = '\0';
  }
  remove(user_pipe_buf);
  kill(pid, SIGKILL);
  exit(0);
```

parent's loop

```
int pid = fork();
if (pid == 0) {
  int fd_child;
  fd_child = open(user_pipe_buf, O_RDONLY);
  while (1) {
    reader = realloc(reader,1024*sizeof(char));
    memset(reader, 0, 1024*sizeof(char));
    read(fd_child, reader, 1024);
    if (strlen(reader)>0) {
      printf("\33[2K");
      printf("\r");
      printf("%s\n",reader);
    }
    fflush(stdout);
  }
  close(fd_child);
  exit(0);
```

child's loop

```
numan@numan: ~/Desktop/deneme
numan@numan:~/Desktop/deneme$ ./a.out
numan@numan:/home/numan/Desktop/deneme shellax$ chatroom comp416 huseyin
[comp416] huseyin >  hey
[comp416] numan > hi
[comp416] huseyin > sup?
[comp416] numan > good,you?
[comp416] huseyin > good.i-realized-we-are-not-using-the-space-character.why-is-
that?
[comp416] numan > interestingly,when-u-type-space,the-words-are-sent-in-seperate
-messages
[comp416] huseyin > oh-i-see.no-worries,this-wont-cost-us-any-points
[comp416] kefah > i-guess-we-will-see-:)
[comp416] huseyin >
```

c) **Wiseman:** The implementation of this command is pretty straight forward as what it does it simply writing a job to the crontab file. This is ensured by creating a new command struct, carrying the command "echo", along with the "<mins> * * * * fortune | espeak" argument. This argument is going to be given to the crontab command. Therefore, a second command struct is created with the name "crontab",

and the output of the first command is piped to the next one. After these chained commands are executed, you will see that the crontab file has a new line indicating the scheduled wiseman job.

```
void wiseman(struct command_t * command) {
        char* temp;
        struct command_t *new_command = malloc(sizeof(struct command_t));
        memset(new_command, 0, sizeof(struct command_t));
        temp = "echo";
        new_command->name = (char *)malloc(strlen(temp) + 1);
        strcpy(new_command->name, temp);
        new_command->arg_count = 1;
        temp = (char*) malloc(sizeof(char)*29);
        strcat(temp, "*/");
        strcat(temp, command->args[0]);
        strcat(temp, " * * * * fortune | espeak");
        new_command->args = (char **) malloc(sizeof(char*));
        new_command->args[0] = temp;

        struct command_t *crontab_command = malloc(sizeof(struct command_t));
        memset(crontab_command, 0, sizeof(struct command_t));
        temp = "crontab";
        crontab_command->name = (char *)malloc(strlen(temp) + 1);
        strcpy(crontab_command->name, temp);
        crontab_command->arg_count = 1;
        temp = "-";
        crontab_command->args = (char **) malloc(sizeof(char*));
        crontab_command->args[0] = temp;

        new_command->next = crontab_command;
        process_command(new_command);
}
```

**d) Openai:** This command is invoked by typing “openai some-sentence-describing-an-image”. After being invoked, it calls the python scripts that connects to the Dall E 2 API and downloads the image that Dall E AI generates with the given sentence. In order to have it working, the necessary python libraries should be installed. When giving a sentence, use the “-“ character in place of spaces.

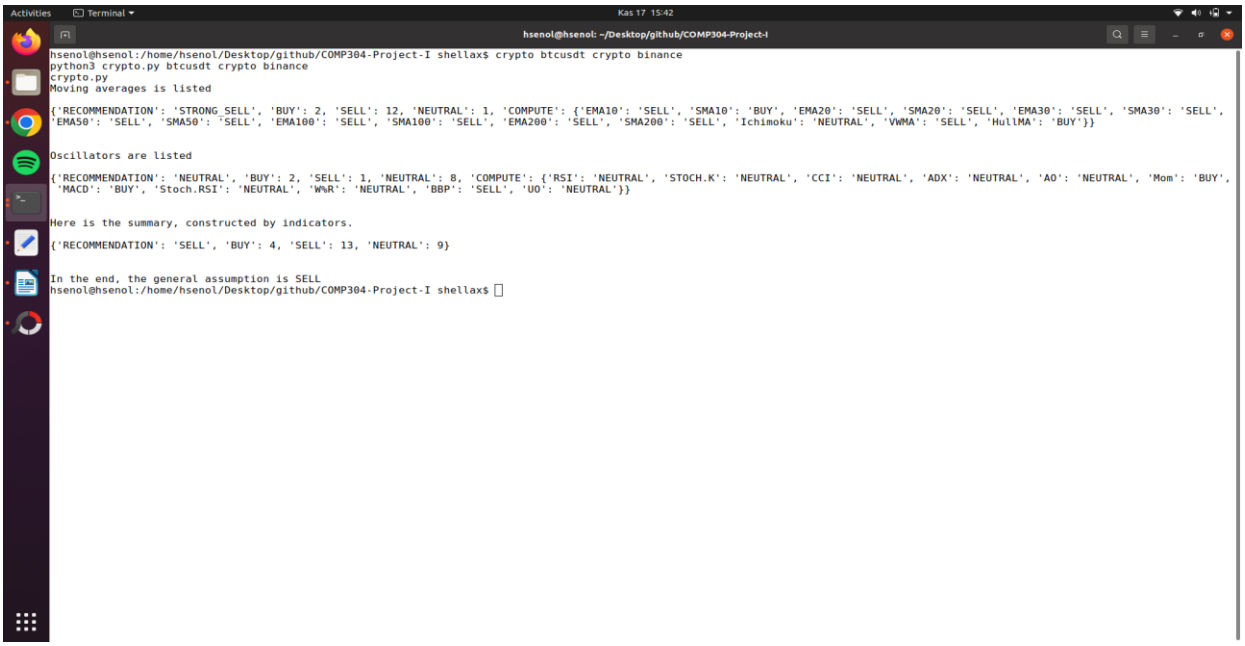
```c
void openai(struct command_t *command) { // custom command (Numan Batur)
  // give the input with - in place of spaces
  // e.g. openai a-man-giving-his-son-money-for-school
  // necessary libraries must be installed for python
  size_t init = 0;
  char *inp = NULL;
  if (command->arg_count == 0) {getline(&inp, &init, stdin);}
  else {inp = command->args[0];}
  printf("%s\n",inp);
  execlp("python", "python", "openai_code.py",inp,(char*) NULL);
}
```

```python
#! pip3 install openai
import os
import openai
import sys
import urllib.request
from PIL import Image as Imagepil

secret = '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'

def main():
        if(len(sys.argv)!=2):
                return -1;

        inp = sys.argv[1].replace("-"," ")
        openai.api_key = secret

        image = openai.Image.create(
                prompt = inp,
                n = 1,
                size = "1024x1024"
                )
        image_url = image['data'][0]['url']

        urllib.request.urlretrieve(image_url, "openai.png")
        img = Imagepil.open("openai.png")

        img.show()


if __name__ == "__main__":
        main()
```

e) **Crypto:** This command is to search oscillators, moving averages and their comments of specific finance instruments of Forex, Nasdaq, Crypto. To run this command, user must give “crypto stock_or_crypto_name screener exchange”. It gives hourly data to the user and at the very last line, the summary of indicators is given. It is invoked by system call in the shellax code. Disclaimer: Anything related to this code or its output is not financial advice.

```python
import sys
from tradingview_ta import TA_Handler, Interval, Exchange


print(sys.argv[0]);
symbol1 = sys.argv[1]
screener1 = sys.argv[2]
exchange1 = sys.argv[3]

coin = TA_Handler(
        symbol = symbol1,
        screener = screener1,
        exchange = exchange1,
        interval = Interval.INTERVAL_1_HOUR

)


data2 = coin.get_analysis().moving_averages
data3 = coin.get_analysis().oscillators
data4 = coin.get_analysis().summary

print("Moving averages is listed\n")
print(data2)
print("\n")

print("Oscillators are listed\n")
print(data3)
print("\n")

print("Here is the summary, constructed by indicators.\n")
print(data4)
print("\n")

print("In the end, the general assumption is {fname}".format(fname  = data4.get('RECOMMENDATION')))

```

```c

void get_crypto_info(struct command_t *command) { // custom command (Hüseyin Şenol)

        char* stock = command->args[0];


        char* screener = command->args[1];

        char* exchange = command->args[2];


        char run_it[150] = "python3 crypto.py ";

        strcat(run_it,stock);
        strcat(run_it," ");
        strcat(run_it,screener);
        strcat(run_it," ");
        strcat(run_it,exchange);
        system(run_it);

}
```




## Part IV

In this part, we modified the mymodule.c to apply Depth First Search. This module prints (with printk) all of the process relations to the kernel logs. After that, our psvis function in shellax with parameters pid and output, collects those relations to a txt file which is namely process_info.txt. To do that, we create many system calls and call them in one line, by concenating them with strcat using ; between them. All of the 'Make', Kernel Load and Remove operations are performed under shellax. User does not need to do anything outside of the Shellax.

Then, draw_tree function is to implement graphdot in our project. Given process information, draw_tree firstly opens the process_info.txt and search for least time among the processes. After selecting the elderly child (which has smallest time value), it is stored under a variable. Then, process_info.txt lines are read by function. Strtok operation is performed to create nodes and time information. While writing relations to the graph.dot, pid with selected time is searched and when it is found, it stored also. After reading the file, that specific pid is colored with red.

After all operations are done, a system call is made to invoke graph-drawing. Again, concatenating strings is used here.

```
char buffer2[80] = "dot -Tpng graph.dot -o ";
strcat(buffer2,output_file);
strcat(buffer2,".png");
system(buffer2);
return 0;
```

Then, desired graph is located under working directory.

```
353
354 int draw_tree(char* pid,char* output_file) {
355         FILE *in = fopen("process_info.txt", "r");
356         FILE *out = fopen("graph.dot","w");
357         fprintf(out,"graph G{\n");
358         char line[100];
359         int flag = 0;
360         int safety = 0;
361         unsigned long x = 9999999999;
362         char *ptr;
363         int line_counter = 0;
364         char buffer[sizeof(unsigned long) * 8 +1];
365         char selected[24] = "random_string";
366
367         while (fgets(line,sizeof(line),in)) {
368                         char* tok = strtok(line,"]");
369                         char* tok4 = strtok(NULL,"]");
370                         char* tok1 = strtok(tok4," ");
371                         char* tok2 = strtok(NULL," ");
372                         char* tok3 = strtok(NULL," ");
373                         line_counter += 1;
374
375                         if (strcmp(tok2,pid) == 0) {
376                                 unsigned long iterator = strtoul(tok3,&ptr,10);
377                                 if (iterator <= x) {
378                                         x = iterator;
379                                 }
380                         }
381         }
382
383         const int n = snprintf(NULL, 0, "%lu", x);
384         assert(n > 0);
385         char buf[n+1];
386         int c = snprintf(buf, n+1, "%lu", x);
387         assert(buf[n] == '\0');
388         assert(c == n);
389
390         fclose(in);
391         FILE *bin = fopen("process_info.txt","r");
392         while (fgets(line,sizeof(line),bin)) {
```

```c
444
445 int psvis(char* pid,char* output) {
446         char command[100] = "make;";
447         strcat(command, "sudo dmesg -c;");
448         strcat(command, "sudo insmod mymodule.ko pid=");
449         strcat(command, pid);
450         strcat(command,";");
451         strcat(command,"sudo dmesg -c > process_info.txt;");
452         strcat(command, "sudo rmmod mymodule");
453         system(command);
454
455         int counter = 0;
456         char* output_name = output;
457
458         draw_tree(pid,output_name);
459
460         return 0;
461 }
```

```c
391     FILE *bin = fopen("process_info.txt","r");
392     while (fgets(line,sizeof(line),bin)) {
393
394
395             if (line_counter == 0) {
396                     printf("File is empty.");
397                     }
398
399             else if (line_counter == 1) {
400                     char* tok = strtok(line," ");
401                     char* tok1 = strtok(NULL," "); // pid
402                     char* tok2 = strtok(NULL," "); // ppid
403                     char* tok3 = strtok(NULL," "); // time.
404
405                     fprintf(out, "%s  -- %s\n", tok1,tok2);
406                     break;
407             }
408             else {
409                     if (flag == 0) {
410                             flag += 1;
411                     }
412                     else {
413                             char* tok = strtok(line,"]");
414                             char* tok4 = strtok(NULL,"]");
415                             char* tok1 = strtok(tok4," ");
416                             char* tok2 = strtok(NULL," ");
417                              char* tok3 = strtok(NULL," ");
418                              char* p = strchr(tok3, '\n');
419                              if (p != NULL) {
420                                     *p = '\0';
421                              }
422
423                              if (strcmp(buf,tok3) == 0) {
424                                   strcpy(selected,tok1);
425
426                              }
427                             if (strcmp(tok1,pid) != 0) {
428                             fprintf(out, "\"%s\" -- \"%s\"\n", tok1 ,tok2);}
429                             fprintf(out, "\"%s\" [label= \"%s time= %s\"]\n",tok1,tok1, tok3);
430                     }
431             }
432     }
433     fprintf(out,"%s [color = \"red\"]\n", selected);
434     fprintf(out, "}\n");
435     fclose(in);
436     fclose(out);
437     char buffer2[80] = "dot -Tpng graph.dot -o ";
438     strcat(buffer2,output_file);
439     strcat(buffer2,".png");
440     system(buffer2);
441     return 0;
442 }
```