KOÇ
UNIVERSITY

COMP304 - Operating Systems
Project 1: Shellax
Fall22

Project Report

Gökçe Sevimli 63992
Melis Oktayoğlu 64388
Github Repository : https://github.com/moktayoglu/comp304-p1

Part I

execv() is implemented by first creating a character array for bin directory. Then the bin path, which is “/usr/bin”, is copied in this character array and the command name is appended in order to search the path for the command invoked. Since all the commands are stored in the bin, this was necessary to give to execv as it doesn’t automatically fill in the path.

```
char bin_dir[100];
strcpy(bin_dir, "/usr/bin/"); // copy for bin direction
strcat(bin_dir, command->name);
execv(bin_dir, command->args);
```

Part II

a) I/O Redirection

I/O redirection is implemented by creating a method redirection_part2(struct command_t *command). In this method three possible cases are handled.

For redirection symbol <, we opened the 0th argument of command->redirects in read only mode and copied the file to stdin using dup2 method as instructed. Here the input file contents are copied to the standard file descriptor, so that in the next command parse the command can directly access the copied input content.

```
void redirection_part2(struct command_t *command){
  // Don’t put spaces after redirection symbols !!!!
  if (command->redirects[0] != NULL) {         // for <
    int in = open(command->redirects[0], O_RDONLY, 0644);
    if (in == -1){
      printf("Error opening redirect source file!\n");
    }
    dup2(in, STDIN_FILENO); // copy file to stdin
    close(in);
  }
```

For the redirection symbol >, this time the output of the command are copied to the standard out file descriptor, so that in the next command parse the command can directly access the copied out content.

```
  if (command->redirects[1] != NULL)  {     // for >
   int out = open(command->redirects[1], O_WRONLY | O_CREAT |
O_TRUNC, 0644);
```

```
    if (out == -1) {
      printf("Error opening redirect target file!\n");}
      dup2(out, STDOUT_FILENO); // copy file to stdout
      close(out);
  }
```

For append redirection symbol, the append source file is read and copied to the standard out file descriptor, again in the next iteration the upcoming command can access the contents easily.

```
  if (command->redirects[2] != NULL)  {   // for >>
    int out = open(command->redirects[2], O_WRONLY | O_CREAT |
O_APPEND, 0644);
    if (out == -1){
      printf("Error opening redirect target file!\n");
    }
    dup2(out, STDOUT_FILENO); // copy file to stdout
    close(out);
  }
}
```

Some test cases:
Note: For functionality, usage requires no spaces after redirection symbols.

```
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat me.txt
0
"this is gokce"
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ echo "let's see if it truncates" >me.txt
0
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat me.txt
0
"let's see if it truncates"
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ echo "not existing file redirection" >notexist.txt
0
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat notexist.txt
0
"not existing file redirection"
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat me.txt
0
"let's see if it truncates"
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ echo "let's see if it appends" >>me.txt
0
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat me.txt
0
"let's see if it truncates"
"let's see if it appends"
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ echo "does not exist file" >>donotexist.txt
0
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat donotexist.txt
0
"does not exist file"
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$
```

```
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/fall22/co
mp304/comp304-p1/comp304-p1 shellax$ cat hi
README.md
shellax
shellax-skeleton.c
text
README.md
shellax
shellax-skeleton.c
text
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/fall22/co
mp304/comp304-p1/comp304-p1 shellax$ sort <hi
README.md
README.md
shellax
shellax
shellax-skeleton.c
shellax-skeleton.c
text
text
```

b) Piping

For piping, first by copying the current command into a dummy temp variable, and taking the next on it (which holds the piped command) the total number of consecutive pipes were counted.

Then, a pipe array of duble the size of total number of pipes were opened. This was needed as the pipes have one in and one out end, thus are always 2 in size. I then thought of iterating over this pipe array by constructing a for loop that iterates over the number of pipes. Each iteration opens up the create the pipes of the assigned process accessed by the pipe array index. After this, again the code iterates over the number of pipes now opening a process by forking for the commands involved in the pipe.

```
// PART 2 - piping
if (command->next)
{
  struct command_t *tmp = malloc(sizeof(struct command_t));
  memcpy(tmp, command, sizeof(struct command_t));

  while (tmp->next)
  {
    num_pipes++;
    tmp = tmp->next;
  }
}
//printf("Number of pipes %d\n", num_pipes);
//printf("here\n");
int fd_pipes[2 * num_pipes];
for (int i = 0; i < num_pipes; i++)
{
  if (pipe(fd_pipes + i * 2) == -1)
  {
    printf("Error creating the pipe!\n");
    return UNKNOWN;
  }
}

for (int i = 0; i < num_pipes + 1; i++)
{
  pid_t pid = fork();

  if (pid == 0) // child
  {
```

As soon as we enter the child process, I check whether the current iteration indicates the last process –which in this case the copying of the process' output to standard output wont be necessary. If not the last command, it automatically copies the output contents to the stdout. Conversely, if the file is not the first command — which is the case where the command wont be taking an input from another process, thus wouldnt need a read from the pipe, the code copies the pipe input content to the stdin file descriptor. As all these are completed, by iterating over the number of files all the pipes are closed. It is important to note here that, as the fork creates copies of the variables in its own memory, all the pipes are closed, which wouldnt be reached from the upcoming fork that would read or write to the standard file descriptor.

```
if (i != num_pipes)
{
  if (command->next)
  {
    dup2(fd_pipes[2 * i + 1], STDOUT_FILENO);
  }
}
// if not first command, without a read
if (i != 0)
{
  dup2(fd_pipes[2 * i - 2], STDIN_FILENO);
}

for (int j = 0; j < 2 * num_pipes; j++)
{
  close(fd_pipes[j]);
}
```

Finally, after the child process terminates, this time the parent copies of the pipes are closed. Then, the new iteration to read/write from the pipes are allowed for the upcoming command.

```
for (int j = 0; j < 2 * num_pipes; j++)
{
  close(fd_pipes[j]);
}
for (int k = 0; k < num_pipes + 1; k++)
{ // wait for child process to finish
  wait(&status);
}
return SUCCESS;
```

Some test cases:

```
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/fall22/co
mp304/comp304-p1/comp304-p1 shellax$ cat hi | sort
README.md
README.md
shellax
shellax
shellax-skeleton.c
shellax-skeleton.c
text
text
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/fall22/co
mp304/comp304-p1/comp304-p1 shellax$ cat hi | sort | wc -l
8
```

Part III

a) Uniq (myuniq)

Uniq implementation is handled by using “myuniq” command. To store the contents of text file an array is created and the content of the text file is read by using

```
read(STDIN_FILENO , buf ,  sizeof(buf));
```

This buffer(buf) is splitted by new line characters, in order to obtain strings rather than characters in each line.

```
split_request = strtok(buf,"\n");
```

Then the content of the buf is stored in the previous mentioned array.

```
while(split_request != NULL) {
          strcpy(line[l],split_request);
          l++;
          i++;
          split_request = strtok(NULL,"\n");
       }
```

In order to eliminate the duplicate ones, for loop is created with a flag variable called “check”. If check is equal to 1, this means consecutive elements are not equal and the string is demonstrated using printf; if check is equal to 0, this means the consecutive elements are equal, hence the element is not printed.

Furthermore, -c and –count features are also added by usage of count.

```
int tot = i;
int check = 0; //flag var
int count2 = 0; //for number of occurrences
for (int i = 0; i <= tot; ++i){
   if(check!=0){
       if((command->arg_count>0) &&
(((strcmp(command->args[0],"-c")==0)) ||
(strcmp(command->args[0],"--count")==0))){
           printf("%d %s\n",count2,line[i-1]);
```

```
            }
            else{
               printf("%s\n",line[i-1]);
             }
            count2 = 1;
        }
            if(check==0){
               count2++;
            }
         check = strcmp(line[i],line[i+1]); //check if consecutive
    strings are equal
    }
```

```
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat input.txt
fefe
gok
gok
gok
maco
maco
mel
mel

parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat input.txt | myuniq
fefe
gok
maco
mel
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat input.txt | myuniq -c
1 fefe
3 gok
2 maco
2 mel
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ cat input.txt | myuniq --count
1 fefe
3 gok
2 maco
2 mel
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$
```

b) Chatroom

First, the tmp/chatroom-<roomname> folder creation is handled by implementing appopriate checks for the case of folders already existing or not. Also, the named pipes are created again by checking if they already exist by the username given by the user.

```c
int chatroom(struct command_t *command)
{
  char chatroom_dir[100];
  strcpy(chatroom_dir, "/tmp");
  // create tmp folder if doesnt exist
  if (mkdir(chatroom_dir, S_IRWXU | S_IRWXG | S_IRWXO) == -1)
  {
    if (errno != EEXIST)
    {
      printf("Chatroom error tmp folder: %s\n", strerror(errno));
    }
  }
  strcat(chatroom_dir, "/chatroom-");
  strcat(chatroom_dir, command->args[0]);
  strcat(chatroom_dir, "/");

  // create room folder if doesnt exist
  if (mkdir(chatroom_dir, S_IRWXU | S_IRWXG | S_IRWXO) == -1)
  {
    if (errno != EEXIST)
    {
      printf("Chatroom error room folder: %s\n", strerror(errno));
    }
  }

  char pipe_path[100];
  // create user named pipe if doesnt exist
  strcpy(pipe_path, chatroom_dir);
  strcat(pipe_path, command->args[1]);

  int fd;
  char buff_in[140];
  char buff_out[140];

  mkfifo(pipe_path, 0666);
  if (errno != EEXIST)
  {
    printf("Chatroom error named pipe: %s\n", strerror(errno));
  }
```

Then, I opened a child process which was necessary to first write to the other users' allocated named pipes. This is done in a while(1) construct to ensure continues write abilities to the user. To access to the other users ' named pipes in the chatroom, I open

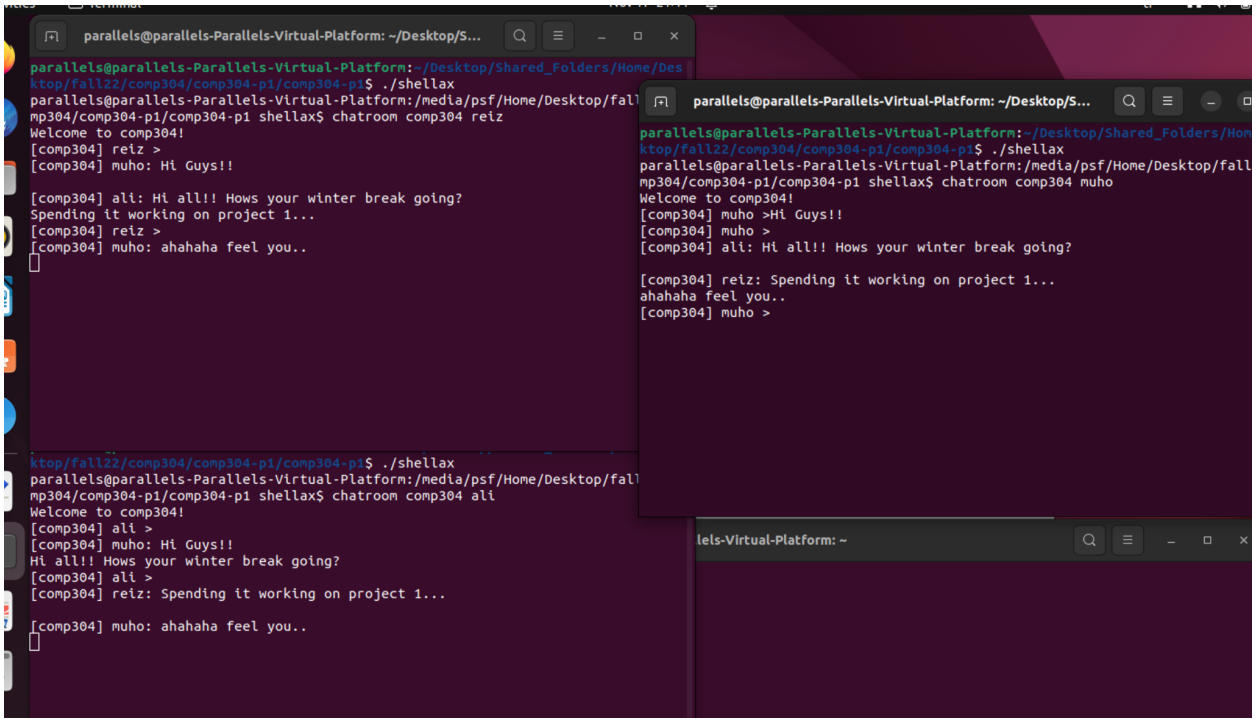
the directory and open and write to their named pipes iteratively.

```
pid_t pid = fork();
if (pid == 0)
{
  while (1)
  {
    printf("[%s] %s >", command->args[0], command->args[1]);

    // get user message
    fgets(buff_in, 140, stdin);

    char temp_buff[180];
    strcpy(temp_buff, "[");
    strcat(temp_buff, command->args[0]);
    strcat(temp_buff, "] ");
    strcat(temp_buff, command->args[1]);
    strcat(temp_buff, ": ");
    strcat(temp_buff, buff_in);

    DIR *d;
    struct dirent *dir;
    d = opendir(chatroom_dir);
    char *write_pipe = (char *)malloc(1 * sizeof(char));
    write_pipe[0] = '\0';
    char *temp = (char *)malloc((strlen(write_pipe) + 1) * sizeof(char));
    if (d)
    {
      while ((dir = readdir(d)) != NULL)
      {

        if ((strcmp(dir->d_name, command->args[1]) != 0) && dir->d_name[0] != '.')
        {
          // each time keep the room directory, append the username
          temp = (char *)realloc(temp, (strlen(chatroom_dir) + strlen(dir->d_name) + 1) * sizeof(char));
          strcat(temp, chatroom_dir);
          strcat(temp, dir->d_name);
          printf("%s \n", temp);

          fd = open(temp, O_WRONLY);
          // write to other pipes
          write(fd, temp_buff, strlen(temp_buff) + 1);
```

Then each named piper is closed after the writing to ensure functionality in the child process. As if not the pipes would hang. A crucial implementation detail in this part, is the parent process also reads continuously from the users' named pipe. This ensures that the pipes dont hang, because although closed, the writing solely isnt enough to not hang the pipe. Reading from the pipe is a must to not hang the pipe. Again then, the named pipe is closed.

```
          close(fd);
          temp[0] = '\0';
        }
      }
      closedir(d);
    }
  }
}
else
{
  while (1)
  {
    // read from YOUR pipe only
    fd = open(pipe_path, O_RDONLY);
    read(fd, buff_out, sizeof(buff_out));
    printf("%s", buff_out);
    close(fd);
  }
}
}
```

Some test cases:
(For functionality, all users in the chatroom must be online)



c) Wiseman

Wiseman is implemented by using crontab. Initially a file pointer is created to open cronjob.txt in write mode.

```
FILE *cronjob_hw_file = fopen("cronjob.txt", "w");
```

Then the shell and path is given into cronjob.txt, where SHELL is:

```
SHELL=/bin/bash
```

, and the path is:

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
*/1* * * * *  fortune | espeak
```

Wiseman speaks every x many minutes, which is given by user. (Above in PATH, “1” is given by me for testing.)

```
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ wiseman 1
wiseman will speak for every 1 minutes.
```

d) Custom Commands

- **Pomodoro Timer** *(implemented by Melis Oktayoğlu)*

  I implemented a pomodoro timer that the cycles and the length of work and break cycles can be given by the user in minutes' scale. The method also prompts some motivational writing to the users' terminal as the user enters the first and last cycle, and the last five minutes of the study.

  The design follows a simple construct by assigning the study cycle to the child process and the break cycle to the parent process. The cycles given by the user iteratively creates child processes the keep track of the study time. As the user enters the study cycle the method prompts a motivation, and waits until the end of the study cycle using the sleep() command.

```
int pomodoro(struct command_t *command)
{
  int num_cycles = atoi(command->args[0]);
  for (int i = 1; i <= num_cycles; i++)
  {
    printf("\aEntering pomodoro cycle %d ... ", i);

    motivation_prompt(i, num_cycles);

    pid_t pid = fork();
    if (pid == 0)
    {
      int study_min = atoi(command->args[1]);
      // printf("study ID: %d\n",getpid());
      if ((study_min - 5) > 0)
      {
        sleep((study_min - 5) * 60);
        printf("Last five minutes... little goes a long way!\n");
        sleep(5 * 60);
      }
      else
      {
        sleep(study_min * 60);
      }

      exit(0);
    }
    else
```

Meanwhile the parent process waits for the child to terminate, then starts its sleep cycle for the given break cycle length. Also, prompts relevant information about the completed cycle number. It also notifies the user with a subtle “beep” sound with the “/a” escape character in the printf command when entering the break cycle. This sound is also present when entering the study cycle.

```
{
  wait(0);
  if (i == num_cycles)
  {
    printf("Great Job! You completed ALL your cycles ( ⁀o⁀)人(⁀-⁀\n");
  }
  else
  {
    printf("Nice... You completed %d cycle(s) \n", i);
    // printf("break ID: %d\n",getpid());
    printf("\aNow its time for a %s minute break...\n", command->args[2]);
    int break_mins = atoi(command->args[2]);
    sleep(break_mins * 60);
  }
}
```

Some test cases:

(For easy visualization very small cycle lengths are selected)

Usage: pomodoro <number of cycles> <length of study cycle in mins> <length of break cycle in mins>

```
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/fall22/comp304/comp304-p1/comp304-p1 shellax$
pomodoro 3 1 1
Entering pomodoro cycle 1 ... Good luck working, keep it zen ⊹(⌣‿⌣)ノ⊹
Nice... You completed 1 cycle(s)
Now its time for a 1 minute break...
Entering pomodoro cycle 2 ... Entering pomodoro cycle 2 ... Nice... You completed 2 cycle(s)
Now its time for a 1 minute break...
Entering pomodoro cycle 3 ... Last cycle! You got this ٩(◡‿◡”)۶
Great Job! You completed ALL your cycles ( ⁀o⁀)人(⁀-⁀
```

- **Fibonacci Guessing Game** *(implemented by Gökçe Sevimli)*

I implemented a fun fibonacci guessing game. In this implementation, I initially wrote a recursive method for getting the nth fibonacci number.

```
int fib(int n) {
      if (n <= 1)
         return n;
    return fib(n - 1) + fib(n - 2); //recursive function call
}
```

Then I defined a method called fibonacci_game to handle the game processes. In this method integer n is defined randomly. Then, inputs (guesses) are taken from two users. If the guess of user 1 is closer to the real value than the guess of user 2, user 1 is the winner and the real value is printed. Similarly, if the guess of user 2 is closer, user 2 is the winner, and real value is printed. If both users are equally closer to the real value, they are both winners.

```
printf("\n\nWelcome to the game of Fibonacci!\n\n");
    int n = (rand() % GAME_ARRAY_SIZE); //randomly assign n

    int guess1;
    int guess2;
    int num_user=2;
    int fibonacci = fib(n); //calling fib function
    printf("Guess the %d th term\n",n);

    printf("Number of users %d\n",num_user); //set to 2

    printf("Guess of user 1:");
    fflush(stdout);
    scanf("%d",&guess1);
    printf("Guess of user 2:");
    fflush(stdout);
    scanf("%d",&guess2);

       if (abs(fibonacci - guess1) < abs(fibonacci - guess2)) {
            printf("Congratulations User 1!\n");
       }
       else if (abs(fibonacci - guess1)> abs(fibonacci -
guess2)) {
            printf("Congratulations User 2!\n");
       }
       else {
            printf("Congratulations User 1 and User 2! You both
win:))\n");
       }
       printf("The value of fib(%d) is %d", n, fibonacci);
       printf("\n");
```

When a user types ‘fib’ in Shellax, the fibonacci_game method is called in process_command, and the game starts.

Some test cases:

```
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ fib


Welcome to the game of Fibonacci!

Guess the 13 th term
Number of users 2
Guess of user 1:200
Guess of user 2:188
Congratulations User 1!
The value of fib(13) is 233
parallels@parallels-Parallels-Virtual-Platform:/home/parallels/comp304-p1/comp304-p1 shellax$ fib


Welcome to the game of Fibonacci!

Guess the 16 th term
Number of users 2
Guess of user 1:400
Guess of user 2:500
Congratulations User 2!
The value of fib(16) is 987
```

Part IV - psvis

In this part of the project, first I wrote a kernel module psvis.c. In this kernel the init takes the pid given to the user to shellax, and is passed through shellax. Then calls the tree traversal method traverse_proc_tree with recursively calls the children of the tasks until reaches leaf nodes. Printing of the tree is also handles during the traversal.

```c
int psvis_init(void){

   struct task_struct *ts;
   ts = get_pid_task(find_get_pid(pid), PIDTYPE_PID);

   if (ts != NULL){
        traverse_proc_tree(ts, 0);
   }else{
        printk("Given PID: %d doesnt exist\n", pid);
        return 1;
   }
   return 0;
}
```

```c
void traverse_proc_tree(struct task_struct* ts, int depth){
   struct list_head *list;
   struct task_struct *child;

   int i;
   for(i = 0; i < depth; i++){
        printk(KERN_CONT "---");
        if (i == depth -1 ){
        printk(KERN_CONT ">");
        }
   }


   printk(KERN_CONT "%d - PID: %d (start time: %lld)\n", depth, ts->pid, ts->start_time);

   list_for_each(list, &ts->children){
         child = list_entry(list, struct task_struct, sibling);
         traverse_proc_tree(child, depth+1);
   }

}
```

Before invoking it in the shellax, first the user should create the kernel module using the Makefile provided. Then, in the shellax first a child process is created to invoke “sudo dmesg -c” which cleans up the kernel stack. This is done to redirect the output here in a clear manner without giving the user unnecessary kernel info in the output file for pid tree visualization. Meanwhile the parent process waits for the stack cleaning to be finished then executes the command for loading the precompiled kernel module by invoking the command “sudo insmod psvis.ko pid=<arg from user>”. Here also the pid argument from the user is passed to the kernel module.

```
if(command->arg_count == 2){

     pid_t pid_dmesg = fork();

     if(pid_dmesg == 0){
        char *clear_msg_cmd[] = {"sudo", "dmesg", "-c", NULL};
        execvp(clear_msg_cmd[0], clear_msg_cmd);


     }else{
        wait(0);
        pid_t pid_ps = fork();

        if (pid_ps==0){//Loading the psvis module

                char pass_pid[20];
                strcpy(pass_pid, "pid=");
                strcat(pass_pid,command->args[0]);
                char *cmd_invoke[]={"sudo", "insmod", "psvis.ko", pass_pid, NULL};
                //printf("invoke\n");
                execvp(cmd_invoke[0], cmd_invoke);


        }else{
                wait(0);

                pid_t pid = fork();
```

Then, the parent process again waits for the kernel load, then removes the kernel in a child process it creates. Parent again waits for this process to finish then it invokes “sudo dmesg” which helps visualizing the kernel stack. And, it redirects this output to the specified output file by the user.

```
            //printf("invoke\n");
            execvp(cmd_invoke[0], cmd_invoke);


        }else{
            wait(0);

            pid_t pid = fork();

            if (pid == 0){
            char *com_remove[]={"sudo", "rmmod", "psvis", NULL};
            //remove kernel

            execvp(com_remove[0],com_remove);

            }else{
                    wait(0);
                    //redirect output
                    int out = open(command->args[1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
                    if (out == -1){

                            printf("Error opening redirect target file!\n");
                    }
                    dup2(out, STDOUT_FILENO); // copy file to stdout
                    close(out);
                    char *dmesg_out[]={"sudo", "dmesg",NULL};
                    execvp(dmesg_out[0], dmesg_out);


            }
```

Some test cases:

```
fall22/comp304/comp304-p1/comp304-p1$ ./shellax
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/
fall22/comp304/comp304-p1/comp304-p1 shellax$ psvis 2853 out_tree
[sudo] password for parallels:
[118453.522732] 0 - PID: 2853 (start time: 90631542503)
[118453.522748] --->1 - PID: 2858 (start time: 90643919845)
[118453.522753] ------>2 - PID: 2932 (start time: 90905562679)
[118453.522757] ------>2 - PID: 2933 (start time: 90906853647)
[118453.522761] ------>2 - PID: 3129 (start time: 92549805882)
[118453.542060] Leaving psvis...
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/
```

out_psvis_tree

```
1 [118073.877106] 0 - PID: 2853 (start time: 90631542503)
2 [118073.877111] --->1 - PID: 2858 (start time: 90643919845)
3 [118073.877113] ------>2 - PID: 2932 (start time: 90905562679)
4 [118073.877131] ------>2 - PID: 2933 (start time: 90906853647)
5 [118073.877133] ------>2 - PID: 3129 (start time: 92549805882)
6 [118073.888705] Leaving psvis...
```

pstree -p output: