Alexandria university

Faculty of engineering

Electrical engineering department

3rd year communication

Simple shell Project Report
OS Lab1

Name:	**ID:**
Alaa Mohamed Morsy	4
Alaa Hisham Mostafa	5
Ahmed Atya Ahmed Abdellatif	28
Ali Hamdy Ali Elfakharany	126

Report Content:

*Note: The subscript numbers show the code line corresponding to the topic.

Introduction

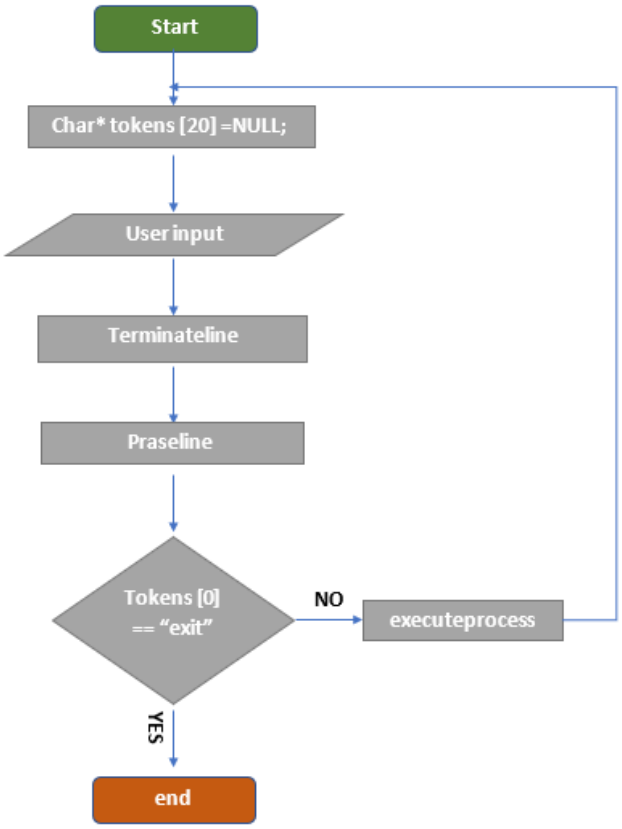
This report describes the overall organization and the major functions used in the implementation of this Unix shell program using C programming language.

This Program supports the following:

1. The internal shell command "exit" which terminates the shell
2. commands with no arguments
3. command with arguments
4. A command, with or without arguments, executed in the background using &.

On the other hand, this program does not support some shell commands as Pipelining and input output redirection operators.

The opposite figure shows the overall flow of the program represented in a flowchart.



The flow of the program contains three essential functions - which are "terminateline", " parseline" and" excecuteprocess" – and two subsidiary functions- which are "handler" and "cd". we will explain them individually in this report.

The libraries used in this program are:

- ✓ string.h
- ✓ stdio.h
- ✓ stdlib.h
- ✓ signal.h
- ✓ unistd.h
- ✓ sys/types.h

<u>Restrictions:</u>

- The program allows the user to input 20 phrases only.
- The program terminates only when the user type "exit "command.

Program overall Flow

```c
int main()
{
    int background ;
    char *buffer = NULL;
    size_t n = 0;
    char *tokens[20];
    printf(COLOR_GREEN"*****This shell is prepared by IDs: 4,5,28,126 *****\n");
    while (1) {                    /* repeat until done ....          */
            for(int j=0;j<20;j++)
              tokens[j]=NULL;
        printf(COLOR_BLUE "Shell > "COLOR_RESET );      /*   display a prompt     */
        getline(&buffer,&n,stdin);             /*   read in the command line     */
        if (strcmp(buffer, "\n") == 0) { /* is it no command ?   */
              printf(COLOR_RED  "No Command\n"COLOR_RESET );
                continue;  }
        terminateLine(buffer);
        printf("\n");
        background = parseLine(buffer,tokens);        /*   parse the line       */
         if (strcmp(tokens[0], "exit") == 0) { /* is it an "exit"?     */
              printf(COLOR_RED  "END of program"COLOR_RESET );
                exit(0);  } /*   exit if it is      */
        executeProcess(tokens,background);          /* otherwise, execute the command */
     }
    return 0;
}
```

The program starts declaring the variables we will use in shell:

- "background": holds a flag donates the ask of the user to use a process in the background (presence of "&" operator).
- "buffer": a dynamic allocated array which takes the user input written in the console [29].
- "n": a variable declared with "size_t" so, "n" can store the maximum size of a theoretically possible object of any type (array in our program).
- "tokens": an array of 20 strings which holds the command line after parsing.

On executing the infinite while loop, shell clears "tokens to be able to execute every command as new [26,27].

Then, it takes the user input with getline function which is a standard library function that is used to read a string or a line from an input stream. Where it is a part of the <string.h> header[29] . The "buffer" array enters "Terminateline" function [33] further to "parseline" function which we will discuss in the following section.

In addition, we check on the user command whether it is “exit”[36,38] -to terminate the shell- or No command[30,32] or is a command, with or without arguments, executed in the background using & or not. then “executeprocess” function take the role to execute the command.

The major Functions

1) “Terminateline”:

when “getline” function fetch the user input it takes the ENTER ‘\n’ as a part of the command.
So, this function scans the whole input line and replace the ENTER character with the Null-terminated String character ‘\0’.

```
void terminateLine(char line[]){
    int i=0;
    while(line[i] != '\n')
        i++;
    line[i] = '\0';
}
```

2) “Parseline”:

the user input may contain: a command, arguments, spaces, operators, and all are concatenated within a line as a group of characters.
That is way parsing take place in this program to identify the parts of the whole command.
In order to do this operation we use “strtok” function which breaks string “line” into a series of tokens using the delimiter space charater “ “ and returns a pointer to the first token found in the string. While if there are no tokens left to retrieve, it returns a null pointer.

```
int parseLine(char line[], char*tokens[]){
   int i=0, flag=0;
   char *token;
   token = strtok(line, " "); /* get the first token */
   /* walk through other tokens */

   while( token != NULL ) {
      tokens[i] = token;
      i++;
      token = strtok(NULL," ");
   }
/*Detect background process*/
   if(strcmp(tokens[i-1], "&") == 0){
        flag = 1;
        tokens[i-1] = '\0';
   }
   return flag;
}
```

At the end of the function, we return a flag determine whether a background command will be executed or not by checking on “&” operator which must be at the end of the command to be executed correctly.

Then if this operator exists, we will replace it with the null-terminated string operator to facilitate the execution of this background command.

3) “Executeprocess”:

```c
void  executeProcess(char *argv[], int flag)
{
    pid_t  pid;
    int    status;
    signal(SIGCHLD, handler);
    if ((pid = fork()) < 0) {     /* fork a child process           */
        printf(COLOR_RED  "* ERROR: forking child process failed\n");
        exit(1);
    }
    else if (pid == 0) {          /* for the child process:         */
        if(strcmp(argv[0], "cd") == 0){
            cd(argv[1]);
        }

        else if (execvp(argv[0], argv) < 0) {     /* execute the command  */
            printf(COLOR_RED  "* ERROR: exec failed\n");
            exit(1);
        }
    }
    else {                                  /* for the parent:      */
        if (flag == 0)
            while (wait(&status) != pid);       /* wait for completion  */
    }
}
```

This function is the backbone of the whole program in which we execute the user command but to understand the flow of this function we will discuss it in a different way.

First, we declare a variable named “pid” in type “pid_t” which is a data type (a signed integer type) stands for process identification and it is used to represent process ids. It requires including of sys/types.h header.

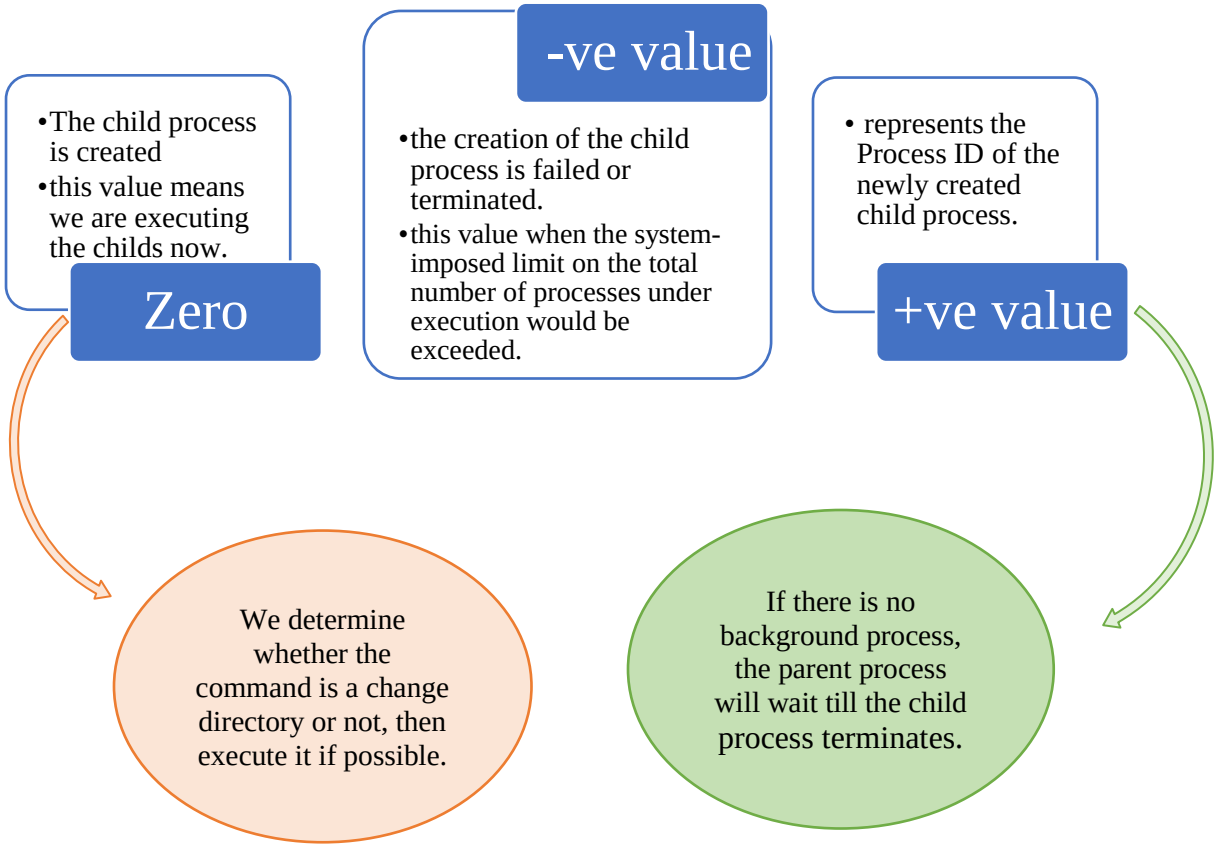
Then we use “fork” system call which is used for creating a new process (child process) by duplicating the calling process (parent process) and both will run concurrently within the program.

NOTE the usage of “signal” function which is from “signal.h” header file.

A signal is an event which is generated to notify a process that some important situation has arrived. When a process or thread has received a signal, the process or thread will stop what it is doing and take some action which is useful for inter-process communication.

SIGCHLD is a symbolic name for a number signal sent to the parent process when child terminates. If this signal is received, we execute the “handler” function which will be discussed in the following section.

“pid” variable can take values represented in the following graph:



When the child process is created i.e. “pid” =0, we check if the command is “cd” -change directory- or not. For the change directory command, we pass the location of path -written in the string following the command- to “cd” function, will be discussed in the next section, to change the directory of both parent and child processes.

On the other hand, “execvp” function – require “unistd.h” header file – executes the other commands if the user input command was wrong the function returns a negative value.

The subsidiary functions

1) "Handler":

When the child process we created through fork function is terminated, it automatically sends a signal to the parent process called SIGCHILD as we discussed.

```
void handler(int sig)
{
  pid_t pid;
  pid = wait(NULL);
  FILE *pointer;
  pointer=fopen("logfile.txt","a+");
  fprintf(pointer,"child %d is terminated \n",pid);
  fclose(pointer);
}
```

So, we try to catch this signal and record it in "logfile.txt"

[47] Using Function "fopen" we create a file called "logfile" then we declare the mode for using this file which is "a+" that means Opens a file for reading and appending.

We save the notation of termination in the file through the function "fprintf".

2) "cd":

this function is used to change the working directory by the value entered after "cd" command then print the new directory [89] .

```
/*Change directory*/
void cd(const char* path)
{
    char s[100];
    if (chdir(path)==0)
        printf("%s\n",getcwd(s,100));
    else
    printf(COLOR_RED  "change directory failed\n");
  exit(0);
}
```

[106] Using "chdir" command which is a system function (system call) used to change the current working directory. This command returns zero (0) on success. -1 is returned on an error where it is declared in "unistd.h".

We use the "getcwd" function to check the current working directory where it returns a pointer which points to a character array where the path of current working directory is stored then print into the console.

The Program outputs

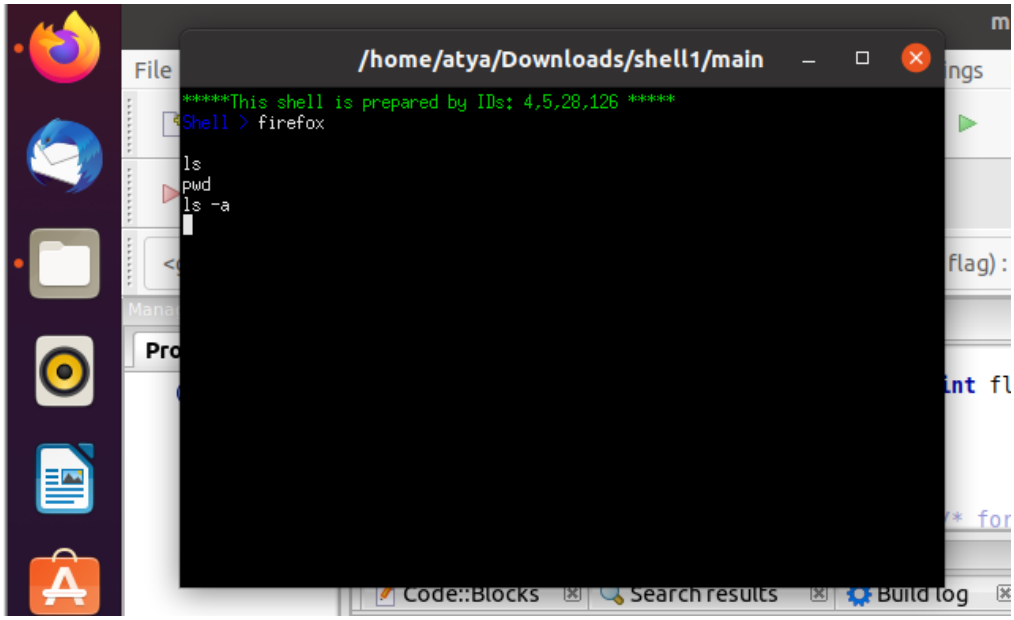
- Sample runs:

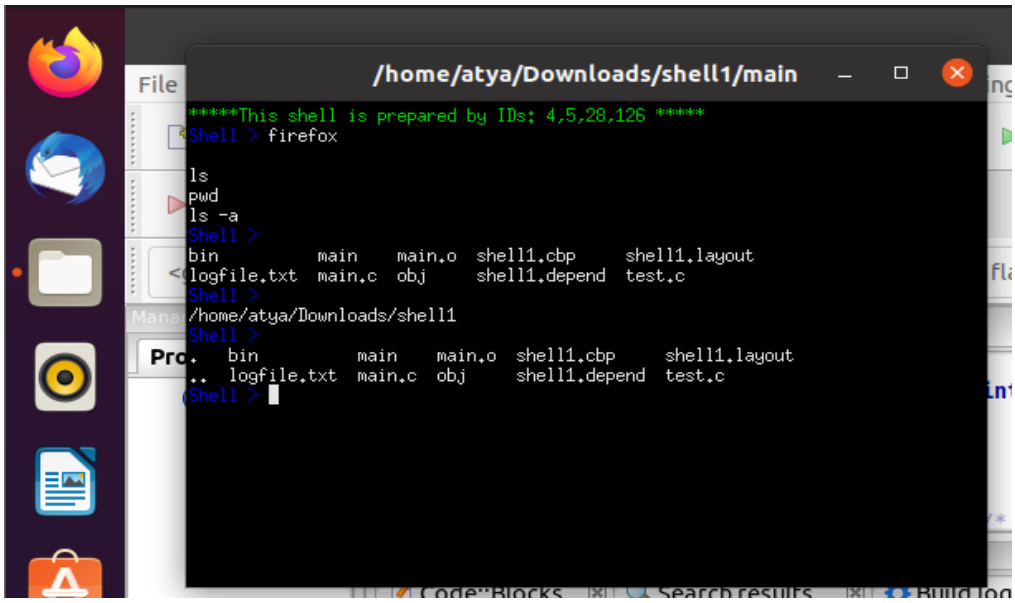
```
Os_project_shell1
*****This shell is prepared by IDs: 4,5,28,126 *****
Shell > ls

bin     obj                     Os_project_shell1.depend
main.c  Os_project_shell1.cbp   Os_project_shell1.layout
Shell > ls -l

total 10
drwxrwxrwx 1 alaa_morsy alaa_morsy    0 Jun  1 22:34 bin
-rwxrwxrwx 1 alaa_morsy alaa_morsy   24 Jun  3 20:36 logfile.txt
-rwxrwxrwx 1 alaa_morsy alaa_morsy 3065 Jun  3 02:48 main.c
drwxrwxrwx 1 alaa_morsy alaa_morsy    0 Jun  1 22:34 obj
-rwxrwxrwx 1 alaa_morsy alaa_morsy 1086 Jun  1 22:25 Os_project_shell1.cbp
-rwxrwxrwx 1 alaa_morsy alaa_morsy  192 Jun  2 02:38 Os_project_shell1.depend
-rwxrwxrwx 1 alaa_morsy alaa_morsy  356 Jun  2 02:54 Os_project_shell1.layout
Shell > pwd

/media/alaa_morsy/Data 1/2nd term-3rd Comm/Operating SyS/LaBs/OS_lab1/Os_project
_shell1
Shell > cd ..

/media/alaa_morsy/Data 1/2nd term-3rd Comm/Operating SyS/LaBs/OS_lab1
Shell >
```

```
Os_project_shell1
*****This shell is prepared by IDs: 4,5,28,126 *****
Shell > firefox &

Shell > ls -l

total 9
drwxrwxrwx 1 alaa_morsy alaa_morsy    0 Jun  1 22:34 bin
-rwxrwxrwx 1 alaa_morsy alaa_morsy 3037 Jun  3 22:28 main.c
drwxrwxrwx 1 alaa_morsy alaa_morsy    0 Jun  1 22:34 obj
-rwxrwxrwx 1 alaa_morsy alaa_morsy 1086 Jun  1 22:25 Os_project_shell1.cbp
-rwxrwxrwx 1 alaa_morsy alaa_morsy  219 Jun  3 21:33 Os_project_shell1.depend
-rwxrwxrwx 1 alaa_morsy alaa_morsy  356 Jun  2 02:54 Os_project_shell1.layout
Shell > ls

bin          main.c  Os_project_shell1.cbp     Os_project_shell1.layout
logfile.txt  obj     Os_project_shell1.depend
Shell > exit

END of program
Process returned 0 (0x0)   execution time : 111.626 s
Press ENTER to continue.
```

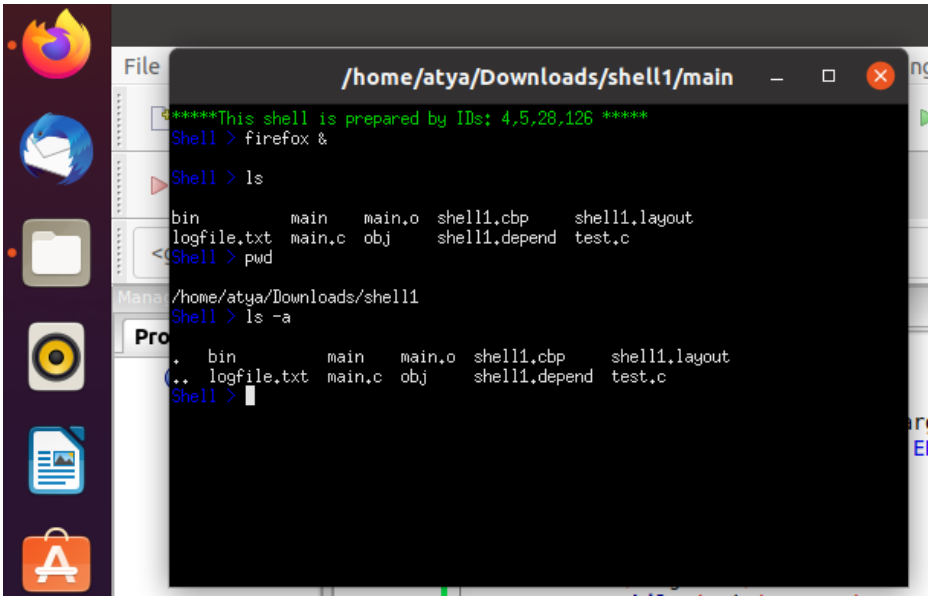
```
Os_project_shell1
*****This shell is prepared by IDs: 4,5,28,126 *****
Shell > xxxx

* ERROR: exec failed
Shell >
No Command
Shell > cd kkkkk

change directory failed
Shell > exit

END of program
Process returned 0 (0x0)   execution time : 31.205 s
Press ENTER to continue.
```

- **Blocking background process:**

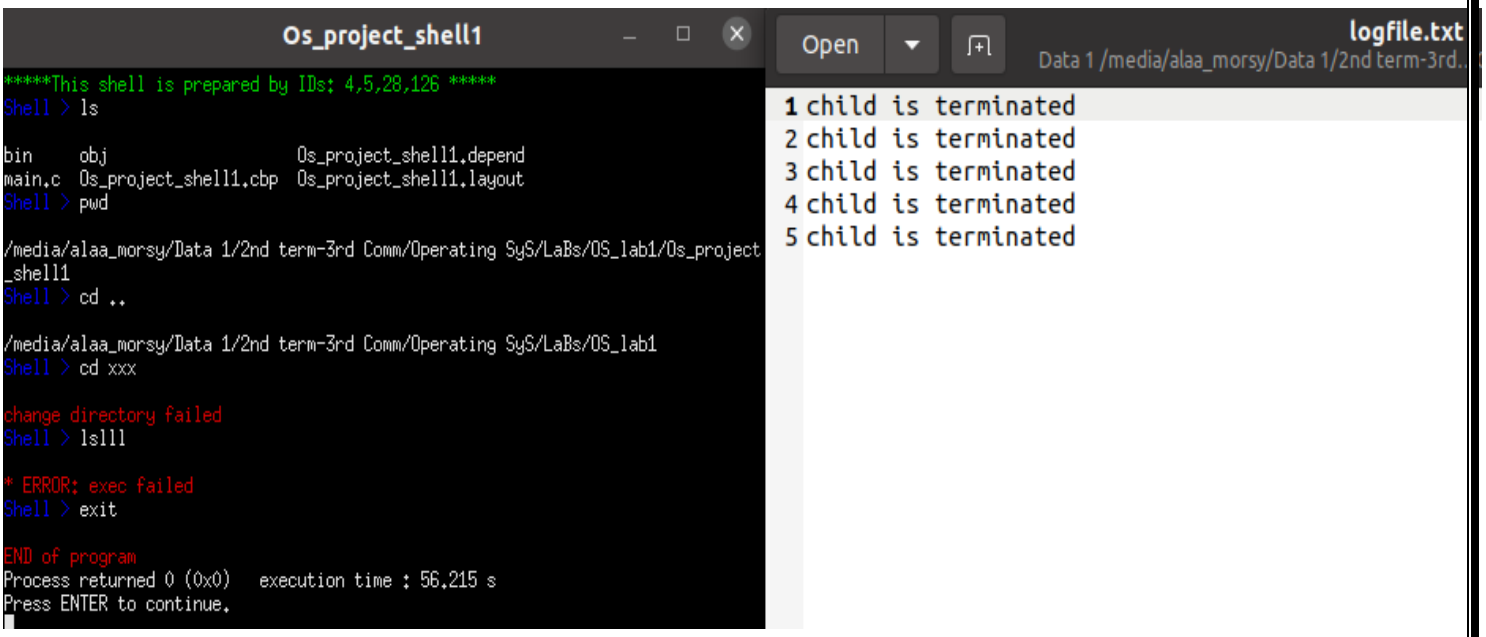




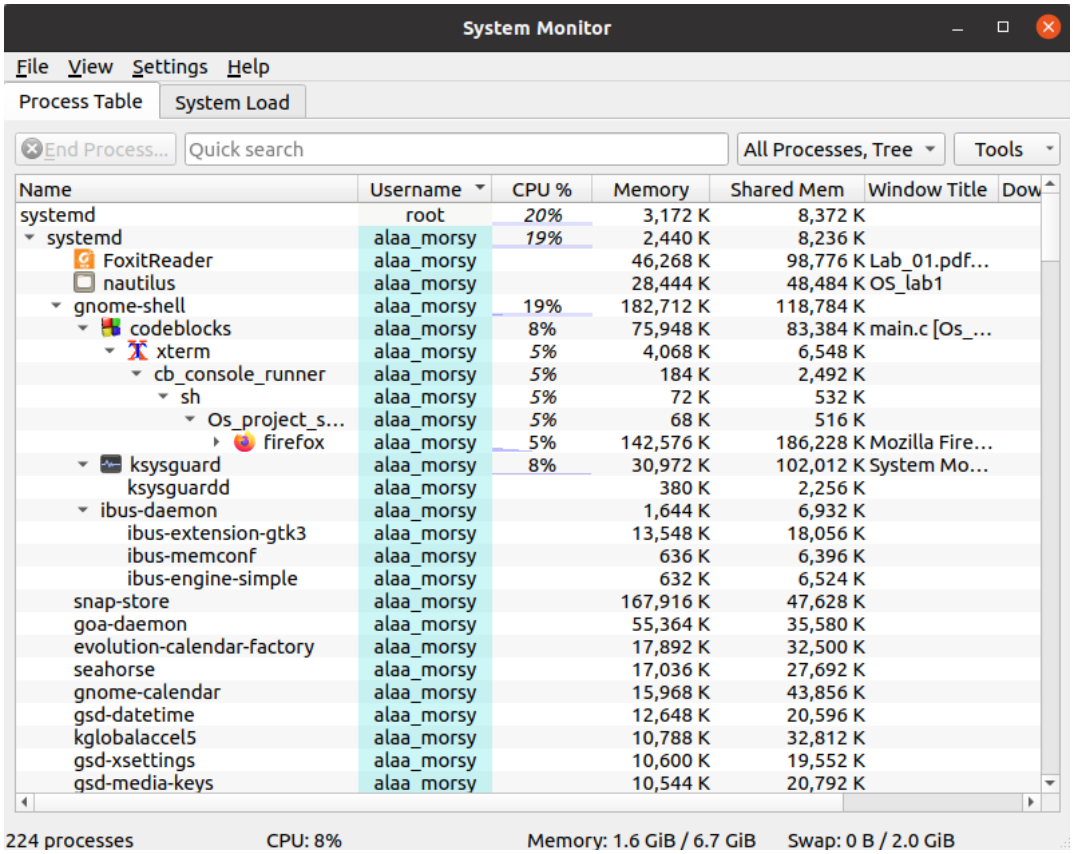
➢ **Unblocking background process:**



- Log file:



- Processes hierarchy in KSysguard during the execution of our shell program:



Name	Username	CPU %	Memory	Shared Mem	Window Title	Dow
systemd	root	20%	3,172 K	8,372 K		
systemd	alaa_morsy	19%	2,440 K	8,236 K		
FoxitReader	alaa_morsy		46,268 K	98,776 K	Lab_01.pdf...	
nautilus	alaa_morsy		28,444 K	48,484 K	OS_lab1	
gnome-shell	alaa_morsy	19%	182,712 K	118,784 K		
codeblocks	alaa_morsy	8%	75,948 K	83,384 K	main.c [Os_...	
xterm	alaa_morsy	5%	4,068 K	6,548 K		
cb_console_runner	alaa_morsy	5%	184 K	2,492 K		
sh	alaa_morsy	5%	72 K	532 K		
Os_project_s...	alaa_morsy	5%	68 K	516 K		
firefox	alaa_morsy	5%	142,576 K	186,228 K	Mozilla Fire...	
ksysguard	alaa_morsy	8%	30,972 K	102,012 K	System Mo...	
ksysguardd	alaa_morsy		380 K	2,256 K		
ibus-daemon	alaa_morsy		1,644 K	6,932 K		
ibus-extension-gtk3	alaa_morsy		13,548 K	18,056 K		
ibus-memconf	alaa_morsy		636 K	6,396 K		
ibus-engine-simple	alaa_morsy		632 K	6,524 K		
snap-store	alaa_morsy		167,916 K	47,628 K		
goa-daemon	alaa_morsy		55,364 K	35,580 K		
evolution-calendar-factory	alaa_morsy		17,892 K	32,500 K		
seahorse	alaa_morsy		17,036 K	27,692 K		
gnome-calendar	alaa_morsy		15,968 K	43,856 K		
gsd-datetime	alaa_morsy		12,648 K	20,596 K		
kglobalaccel5	alaa_morsy		10,788 K	32,812 K		
gsd-xsettings	alaa_morsy		10,600 K	19,552 K		
gsd-media-keys	alaa_morsy		10,544 K	20,792 K		