Alexandria University
Faculty of Engineering

Electronics and Communication Department

Operating Systems

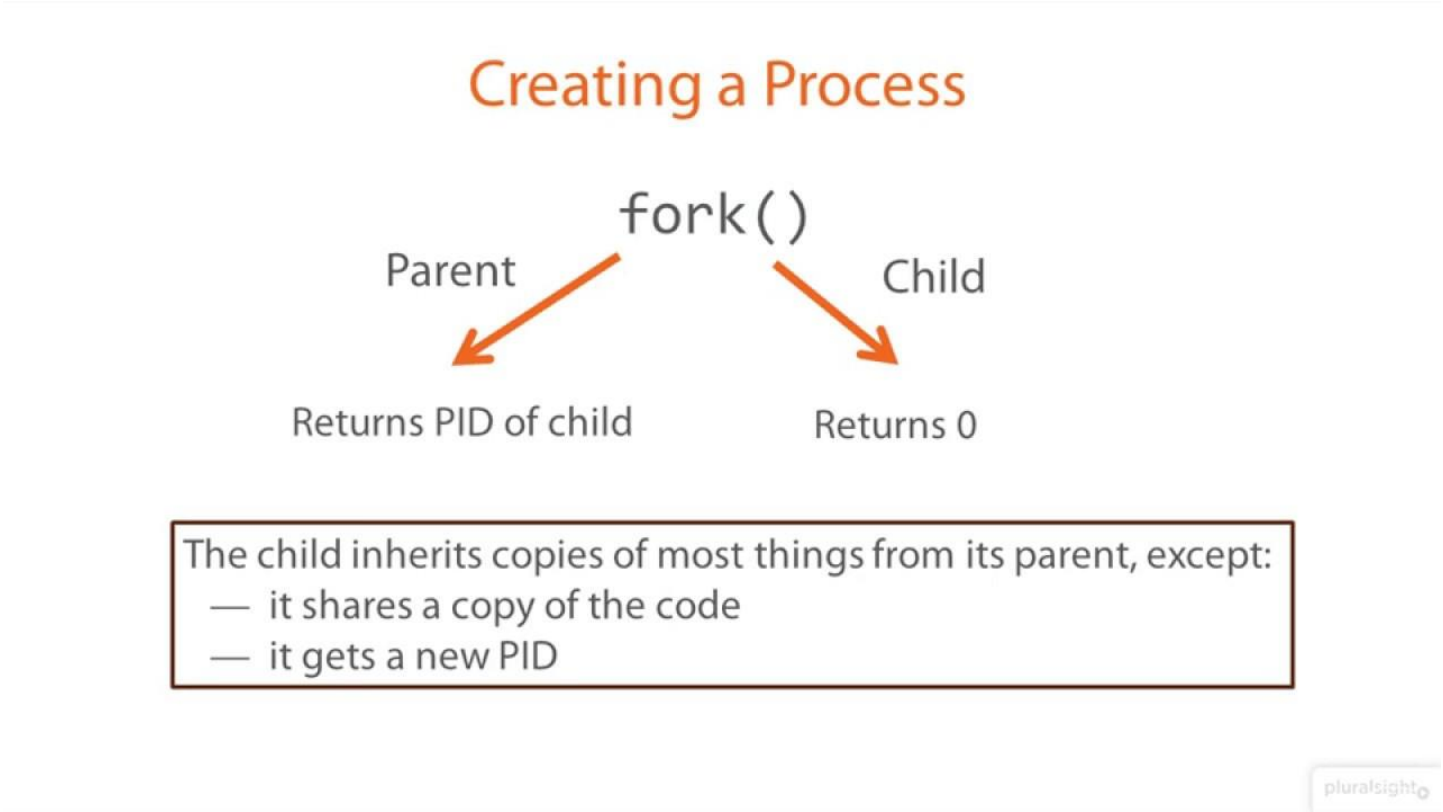
Assignment 1
Simple Shell

Name	ID
Islam Abd El-Rauf Abdo	47
Omar Tarek Abd El-Rahman	136
Mohamed Basuoni Ahmed	170
Mohamed Khaled Abd El-Monem	174
Mohamed Khaled Mohamed Atya	175

Contents:

Introduction

A shell is simply a program that conveniently allows to run other programs. We will deal with commands with no arguments like (ls, cp, rm, …) and with arguments like (ls –l , ls -a) . There are many functions that make these commands implemented easily. We can implement some internal commands like (cd , exit). One of the main concepts in the code is forking a child process and how to control it with other processes.



In the code sections we will discuss the details of implementation for the shell.

Code Sections:

Section (0):

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

void interrupt_handler(int sig);
void parsing(char line[]);
void operation();
void read_str();

char *command;
char *parameters[10];
char line[100] = {0};
FILE *fp;
int flag = 0;
int spaces = 0;

```

- Explanation

  - Including some useful libraries which allow us to execute helpful functions.

  - Declaring functions we will discuss them in next sections.

  - Initializing pointer *command* : points to the string which the user will write the command in.

  - Initializing arrays of pointers:
    parameters : the array of strings which contain the arguments of the command
    line : overall line which user enters

  - Initializing file pointer *fp :* the file pointer that points to a file structure

  - Initializing variables *(flag ,spaces)* for operations

Section (1):

```
// ##########################  main  ####################
int main()
{
    signal(SIGCHLD, interrupt_handler);  ---------->  1
    remove("data.log");            // delete old log file
    fp = fopen("data.log", "a"); // open log file
    while (1)                     // terminal loop
    {
        operation();
    }
    return 0;
}
```

- <u>Explanation</u>

  - A *signal* is a software generated interrupt that is sent to a process by the OS because of when user press ctrl-c or another process tell something to this process. We included *<signal.h>* library for it

  - Signal number have symbolic names. **SIGCHLD** is number of the signal sent to the parent process when child terminates.

  - If the OS receives **SIGCHLD** signal, the OS will execute the routine (*interrupt_handler*) in the process's code to handle the signal. If no signal is received, the process continues executing its next instruction.

- the OS receives **SIGCHLD** signal :

```
void interrupt_handler(int sig)
{
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    {
        // executed if the child is terminated
        if (WIFEXITED(status))
        {

            //fp = fopen("log.txt","a");
            fprintf(fp, "child process (%d) terminated.\n", pid);
        }
    }
}
```

- Explanation

  - *waitpid* suspends the calling process until a specified process terminates. When the specified process ends, status information from the terminating process is stored in the location pointed to by *status* and the calling process resumes execution. If the specified process has already ended when the waitpid function is called and the system has status information, the return from waitpid occurs immediately. A return from the waitpid function also occurs if a signal is received and is not ignored.

  - *Waitpid(-1, &status, WNOHANG):*

    -1: the calling process waits for any child process to terminate.

    status : is a pointer to the location where status information for the terminating process is to be stored

    WNOHANG : causes the call to `waitpid` to return status information immediately, without waiting for the specified process to terminate. Normally, a call to `waitpid` causes the calling process to be blocked until status information from the specified process is available; the `WNOHANG` option prevents the calling process from being blocked. If status information is not available, `waitpid` returns a `0`.

- So we know the status information of the terminated process. to check if the specified process terminated normally we use:

  WIFEXITED(status)

  It returns true if it terminated normally so we can print the PID of the child process in log file

```
// ##########################  main  ####################
int main()
{
    signal(SIGCHLD, interrupt_handler);
    remove("data.log");          // delete old log file
    fp = fopen("data.log", "a"); // open log file
    while (1)                    // terminal loop
    {
        operation();
    }
    return 0;
}
```

2

- In the beginning of shell we delete any old log files then create another new log file and append to log file (write PID of child processes which are terminated)

Section (2):

```c
// ##########################  main  ####################
int main()
{
    signal(SIGCHLD, interrupt_handler);
    remove("data.log");            // delete old log file
    fp = fopen("data.log", "a"); // open log file
    while (1)                     // terminal loop
    {
        operation();  ---------->  3
    }
    return 0;
}
```

```c
void operation()
{
    read_str(); // gets user input and counts the spaces denoting the number of arguments
    parsing(line);

    if ((strcmp(command, "exit") == 0))
    {
        fclose(fp);
        exit(EXIT_SUCCESS);
    }

    else if (strcmp(command, "cd") == 0)
    {
        if (chdir(parameters[1]) == 0)
        {
        }
        else
        {
            printf("\n");
            printf("\033[0;31m"); //print in red color
            printf("path error\n\n");
        }
    }
```

- <u>Explanation</u>

  - We enter *while* loop then call *operation()* function .

  - This function calls another two functions *read_str()* then *parsing(line)* so we will explain them first

- *read_str()* function :

```c
void read_str()
{
    int i = 0;
    spaces = 0;

    printf("\033[0;32m"); //print in green color
    printf("Enter command \n");

    printf("\033[0;36m"); //print in Cyan color
    printf("~$ ");
    printf("\033[0m"); //print in reset color

    fgets(line, 100, stdin); // gets 100 cahracters from stdin
    while (line[i] != '\0')  // loop until end of line
    {
        if (line[i] == ' ') // count spaces
        {
            spaces++;
        }
        i++;
    }
}
```

- <u>Explanation</u>

  - In this function we read the user input (the command and arguments if existed)

  - We check the end of line which user enters by *(\0)* then parse if there is any space to detect the command and the arguments

  - Now we call next function *parsing(line)*

- *parsing(line)* function :

```
void parsing(char line[])
{
    if (spaces == 0) // means single parameter is passed withc
    {
        char *parse = strtok(line, "\n"); // take the first pa
        command = parse;
        parameters[0] = parse;
        parameters[1] = NULL;
        parameters[2] = NULL;
        //strcat(line,'\n');
    }
    else // means a command is run with arguments
    {
        char *parse = strtok(line, " "); // returns the string

        int i = 0;
        while (parse != NULL)
        {
            if (!(strcmp(parse, "&"))) // if ampersand is foun
            {
                flag = 1; // indicates background task
                parameters[i] = NULL;
            }
            else // means this is a parameter or an argument
            {
                parameters[i] = parse;
            }
            parse = strtok(NULL, " \n");
            i++;
        }
        command = parameters[0];
        parameters[i] = NULL;
    }
    //##########################################################################3
}
```

- <u>Explanation</u>

  - In this function there are two cases

    1.spaces == 0 :

    so the line is command without arguments ,then we use *strtok* to modify the string and delete the (/n) from the end of line

    2.spaces != 0 :

    So the line is command with arguments, then we use *strtok* to modify the string and broke it into smaller strings between each others space(“ “) the first string is the command and others are arguments

- There is a special case if there is ampersand '&' which indicates background process not argument so it was handled by *if condition* ans setting variable *flag*

- Also in this case we used strtok to delete (\n) at the end of line

- Now we return to main function *operation():*

```
void operation()
{
    read_str(); // gets user input and counts the spaces denoting the number of arguments
    parsing(line);

    if ((strcmp(command, "exit") == 0))
    {
        fclose(fp);
        exit(EXIT_SUCCESS);
    }

    else if (strcmp(command, "cd") == 0)
    {
        if (chdir(parameters[1]) == 0)
        {
        }
        else
        {
            printf("\n");
            printf("\033[0;31m"); //print in red color
            printf("path error\n\n");
        }
    }
```

- <u>Explanation</u>

  - Now we have the command and arguments(if existed)

  - We compare the command with more than string:

    1. *strcmp(command, "exit"):* if returns zero, so the command is *exit* so we will terminate the program using *exit()* function and close log file using *fclose* so it is *exit_success*(the user want that)

    2. *strcmp(command, "cd"):* if returns zero, so the command is *cd* so we will check the argument with it and use *chdir()* function to change directory to *parameters[1]* which stores the destination.

       if the destination is not valid or no change directory happened, program will print *"path error"*

if previous two cases didn't happen :

```
else
{
    pid_t child_pid = fork(); // create a child process

    if (child_pid == 0)
    {
        // child continues here
        int status_code = execvp(command, parameters);
        if (status_code == -1)
        {
            printf("\033[0;31m"); //print in red color
            printf("Terminated Incorrectly\n");
            exit(EXIT_FAILURE);
        }
    }
    else if (child_pid > 0)
    {
        // parent continues here
        // printf("\n");
        // printf("from parent: pid=%d child_pid=%d\n",

        // Wait until child process exits or terminates

        printf("\033[0;36m"); //print in Cyan color
        printf("\n");

    if (flag == 0)
    {
        int status;
        pid_t waited_pid = waitpid(child_pid, &status, 0); // wait for child to

        if (waited_pid < 0) // means terminated incorrectly
        {
            fprintf(fp, "child process (%d) terminated.\n", child_pid);
            perror("waitpid() failed");
            exit(EXIT_FAILURE);
        }
        else if (waited_pid == child_pid) // means terminated correctly
        {
            if (WIFEXITED(status))
            {
                /* WIFEXITED(status) returns true if the child has terminated
         * normally. In this case WEXITSTATUS(status) returns child's
         * exit code.
         */
                fprintf(fp, "child process (%d) terminated.\n", child_pid);
                printf("\033[0m"); // reset color
                printf("\n");
                //printf("from parent: child exited with code %d\n",
                      // WEXITSTATUS(status));
            }
        }
    }
    else
    {
        flag = 0;
         operation();
    }
}
}
}
```

- Explanation

  - In this section of code we will create a child process using *fork()* to use *execvp*() function

execvp() function:

This function will give the control of the current process (C program) to the command. So, the C program is instantly replaced with the actual command.
So, anything that comes after execvp() will NOT execute, since our program is taken over completely!
However, if the command fails for some reason, execvp() will return -1.
So, whenever you use execvp() if you want to maintain your C program, you generally use *fork()* to first spawn a new process, and then use execvp() on that new process.
The exec type system calls allow a process to run any program files, which include a binary executable or a shell script

- If the child process created successfully (child_pid =0),We pass the command and arguments to execp() function then to check that it is successful operation there is if condition . if it is not successful, program will terminate

- If the parent process(child_pid >0), we won't use execvp() function because what we said above. We will handle if there is background process

- Variable flag= 0 means no background process so here we will deal with processes using waitpid normally and waits until child process terminates and when it terminates correctly ,the pid of it will be written in log file

  But if flag = 1 that means thar there is background process so we don't wait child to terminate . we reset flag and go to operation() function to perform another function

Overall idea of code

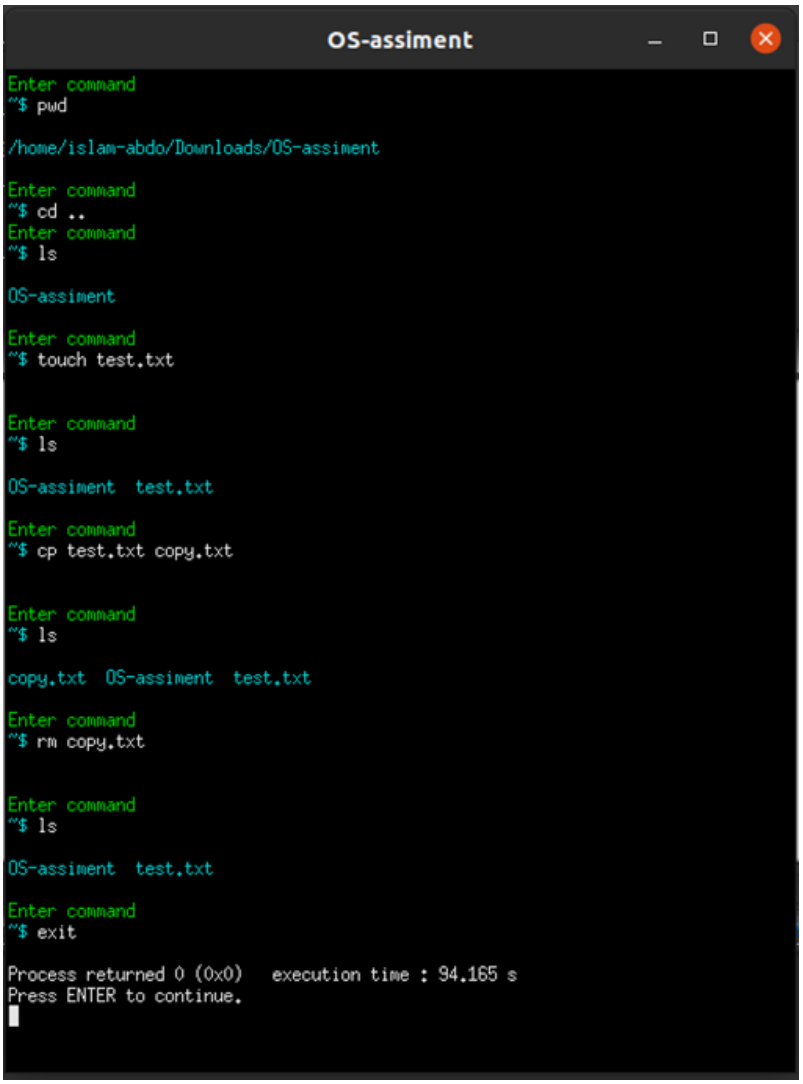
1.we read string command from user using read_str function
2.we parse this string and get the command and the rest parameters using parsing function in which we check if there is a '&' parameter to enable or disable the parent wait using flag

Sample runs:

1.A command with no arguments & “exit” command

Commands used in figure are:

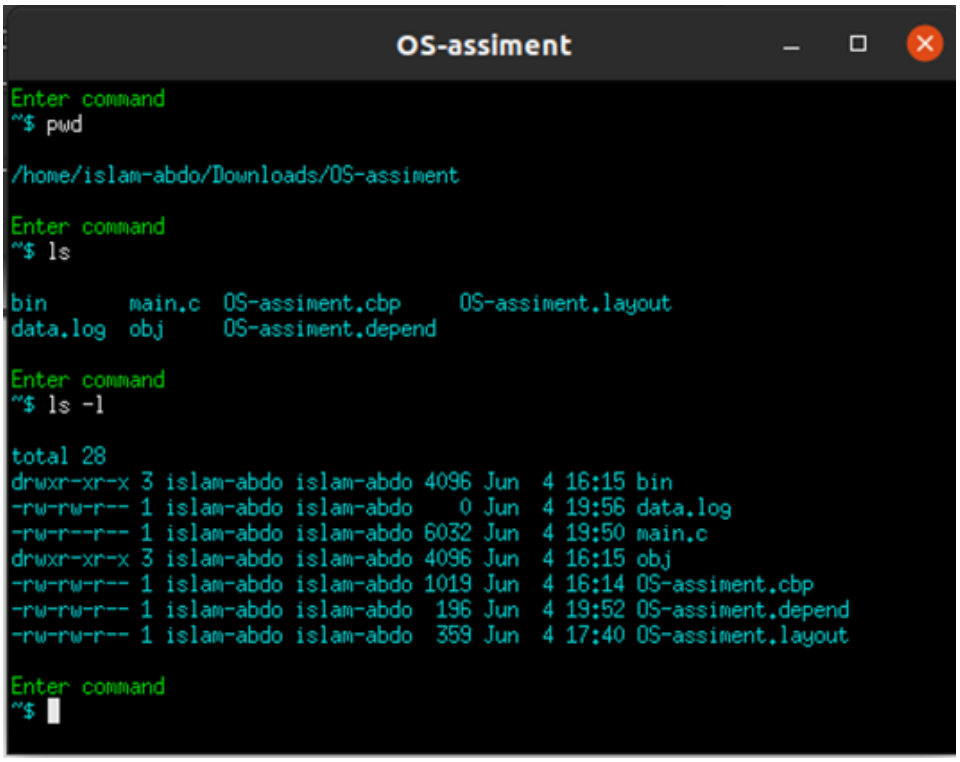
- “pwd”
- “cd ..”
- ”ls”
- “touch test.txt” : create empty text file
- ”cp test.txt copy.txt” : copy test.txt file to anther text file with name copy.txt
- “rm copy.txt” : remove copy.txt file
- ”exit”



2.A command with arguments

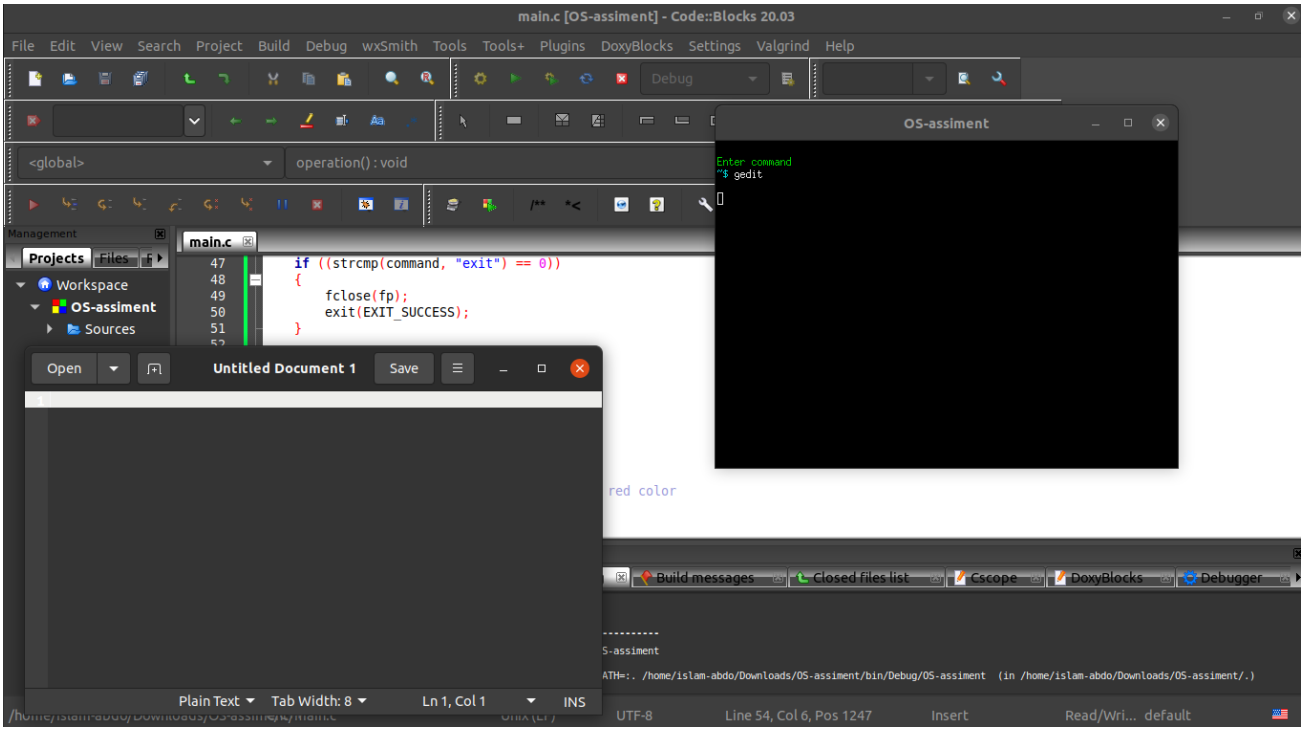
Commands used in figure are:

- “pwd”
- “ls"
- “ls -l”



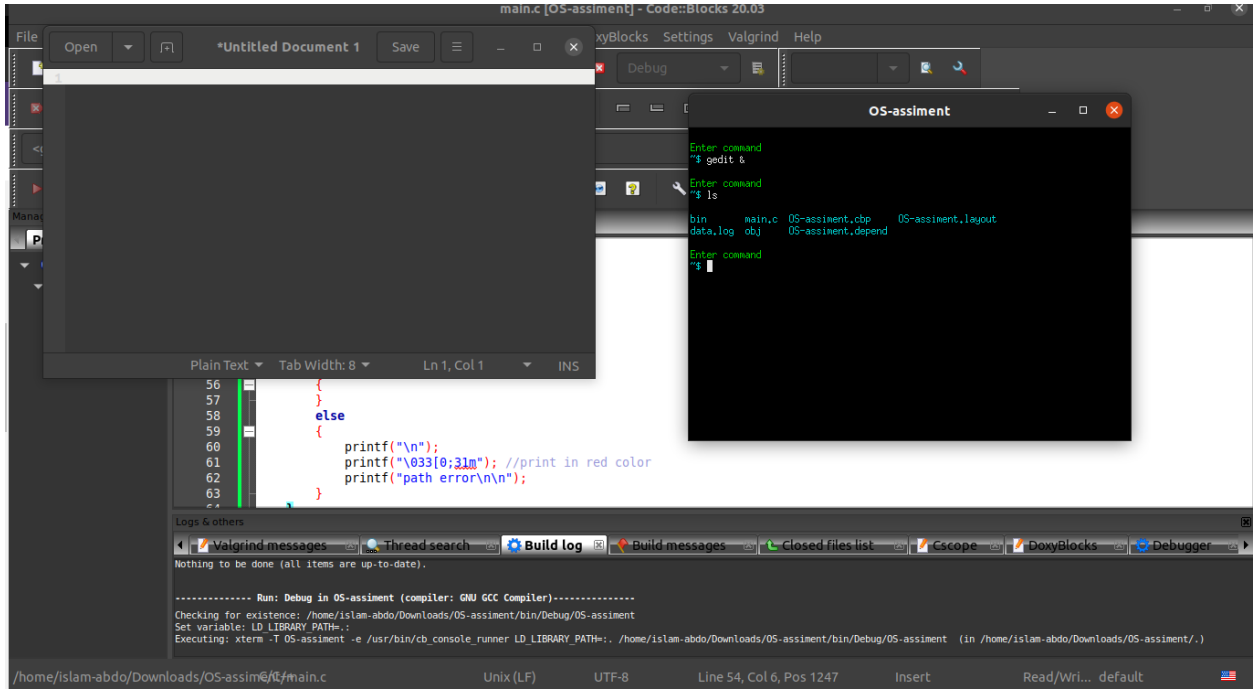
3.A command without executed in the background

- “gedit” : open text editor

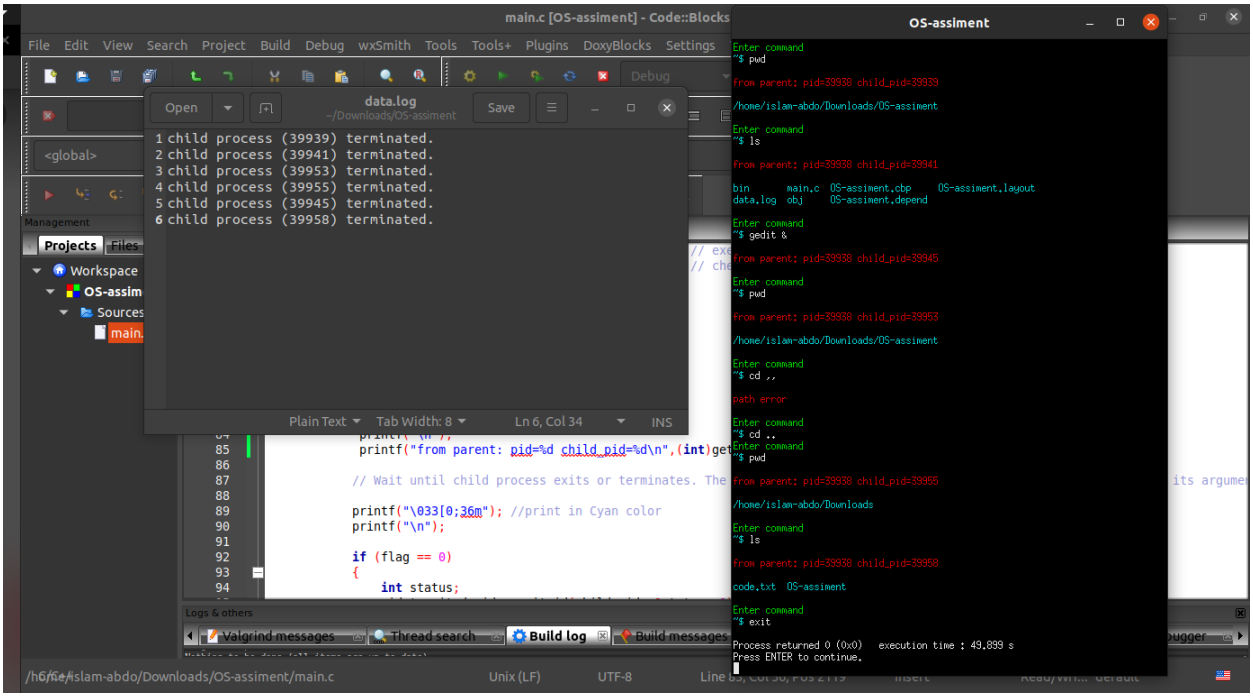


4.A command with executed in the background “&”

- “gedit &” : open text editor in background and do any other command like “ls” in this figure.



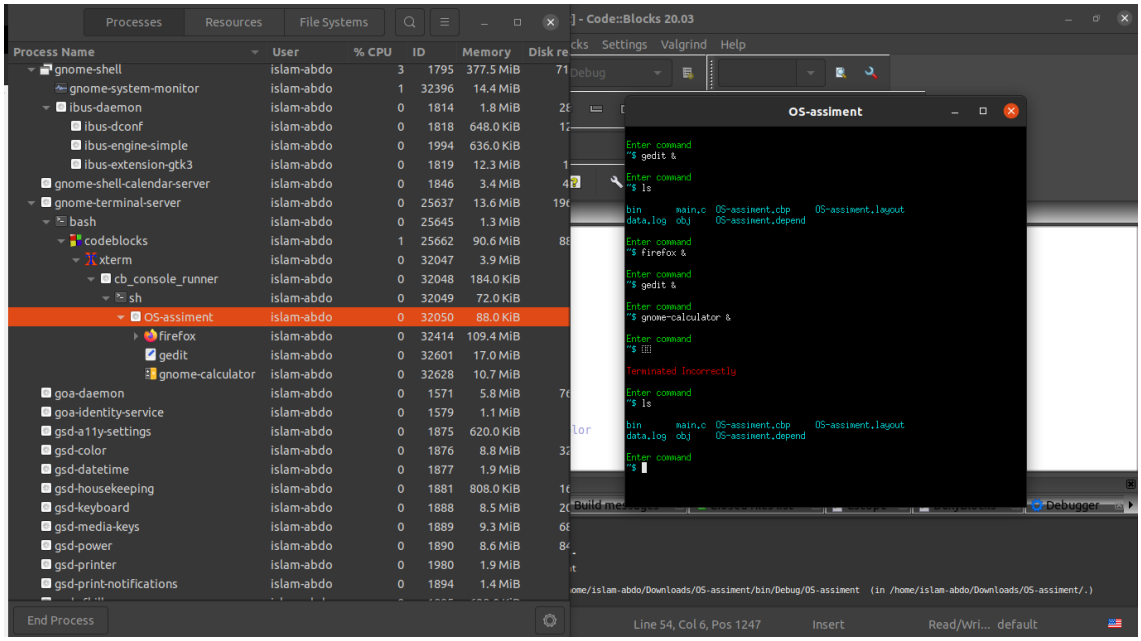
5.Log file after exit shell process



- Processes hierarchy

1.Firefox, Calculator and Gedit are child processes to the Simple-Shell process

- “gedit &”
- “firefox &”
- “gnome-calculator &” : to open calculator



2.Exit all child processes to the Simple-Shell process