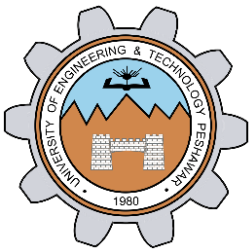
Command Line Arguments

LAB # 07



Spring 2023

CSE-204L Operating Systems Lab

Submitted by: **Ali Asghar**

Registration No.: **21PWCSE2059**

Class Section: **C**

"On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work."

Submitted to:

Engr. Madiha Sher

Date:

19th May 2023

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

Objectives:

- To understand the concept of command-line arguments and their usage in C programming.
- To learn how to access and process command-line arguments in a C program.
- To explore process creation and execution using command-line arguments.
- To create multiple child processes and execute different commands in each child process.

COMMAND LINE ARGUMENTS:

Command-line arguments are essential for providing inputs and additional information to a program during runtime. In C programming, the main() function is used to access and process command-line arguments. The argc parameter represents the count of command-line arguments, and the argv parameter is an array of strings that holds the arguments themselves.

When executing a program from the command line, arguments can be passed after the program name. These arguments are separated by spaces. For example, executing a program named "myprogram" with three arguments would look like: ./myprogram arg1 arg2 arg3.

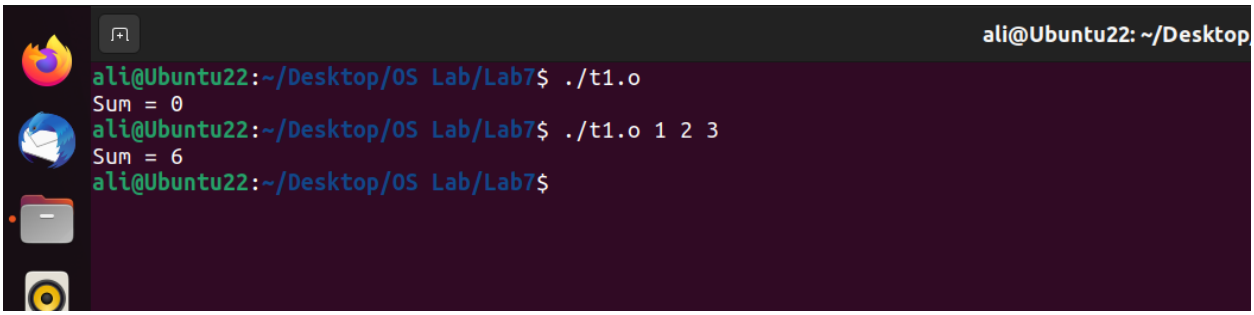
The argc parameter in the main() function will have a value of 4, indicating the program name plus three additional arguments. The argv parameter is an array where argv[0] holds the program name, argv[1] holds "arg1", argv[2] holds "arg2", and argv[3] holds "arg3".

By accessing the argv array, a C program can process and utilize the command-line arguments as needed. This allows for dynamic behavior and customization of the program based on user inputs.

Overall, command-line arguments provide flexibility and interactivity to programs, allowing users to provide inputs and customize program behavior without modifying the source code.

Task1:

Write a C program that finds the sum of all CLA's.



Task2:

Write a C program that creates a child process & execute task 1 in child process using execlp() system call. Parent process shall wait for the child process.

Open

t2.c
~/Desktop/OS Lab/Lab7

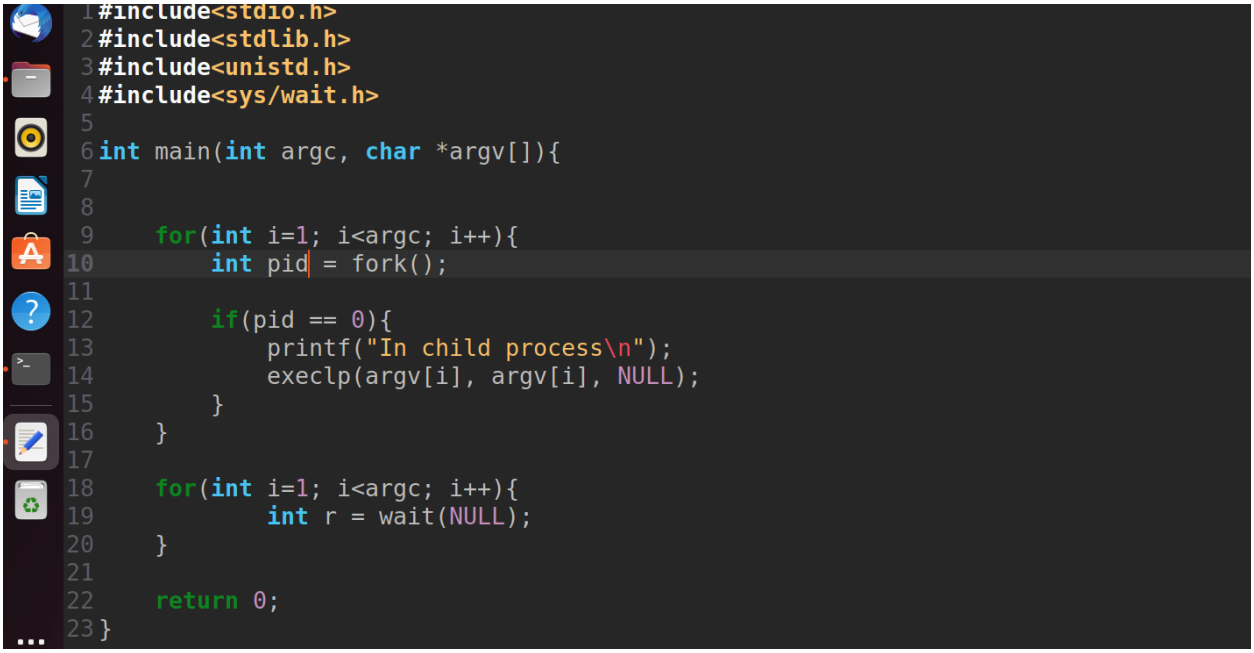
```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int main(int argc, char *argv[]){

    int pid=fork();

    if(pid == 0){
        printf("In child process\n");
        execlp("./t1.o","t1.o", argv[1], argv[2],argv[3], NULL);
    }

    else if(pid > 0){
        int r = wait(NULL);
        printf("Child process terminated\n");
    }

    return 0;
}
```

Activities Terminal

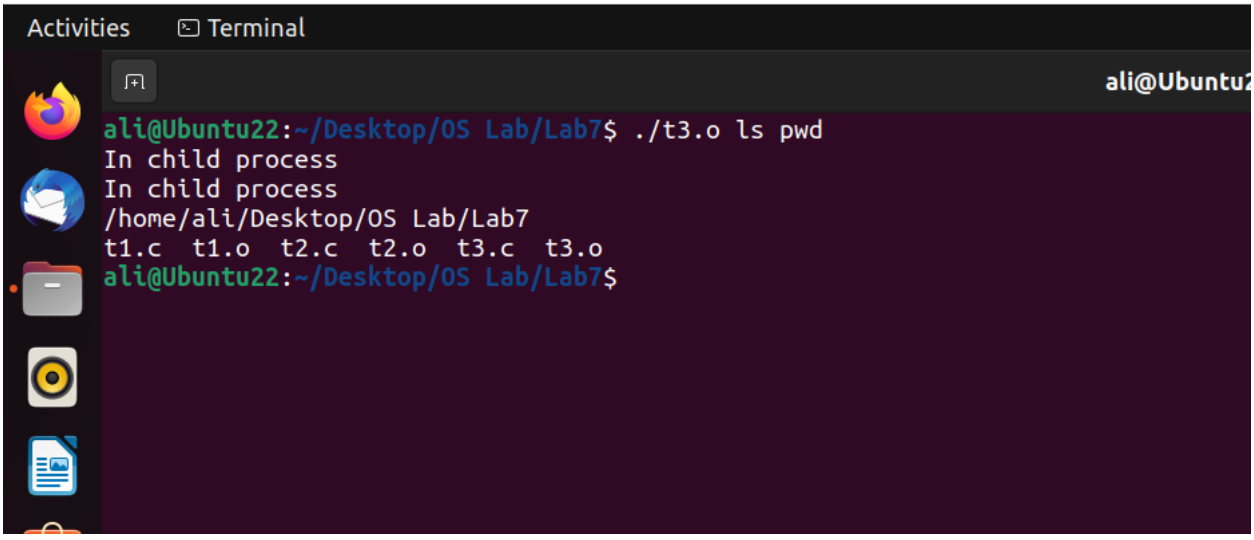
```
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$ ./t1.o
Sum = 0
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$ ./t1.o 1 2 3
Sum = 6
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$ gedit t2.c&
[1] 27985
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$ ./t2.o
In child process
Sum = 0
Child process terminated
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$ ./t2.o 1 2 3
In child process
Sum = 6
Child process terminated
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$
```

Task3:

Write a C program that takes built-in command on CLA's and create separate child process for each command & execute these commands in child process. Parent shall wait for the child processes.



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

int main(int argc, char *argv[]){


    for(int i=1; i<argc; i++){
        int pid = fork();

        if(pid == 0){
            printf("In child process\n");
            execlp(argv[i], argv[i], NULL);
        }
    }

    for(int i=1; i<argc; i++){
            int r = wait(NULL);
    }

    return 0;
}
```



```
Activities   Terminal
ali@Ubuntu2
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$ ./t3.o ls pwd
In child process
In child process
/home/ali/Desktop/OS Lab/Lab7
t1.c  t1.o  t2.c  t2.o  t3.c  t3.o
ali@Ubuntu22:~/Desktop/OS Lab/Lab7$
```