FAKULTI TEKNOLOGI KEJURUTERAAN ELEKTRIK DAN ELEKTRONIK
UNIVERSITI TEKNIKAL MALAYSIA MELAKA

OPERATING SYSTEMS

BEEC3453	**SEMESTER 1**	**SESI 2019/2020**

LAB 4: SYSTEM CALLS

NO.	**STUDENTS' NAME**	**MATRIC. NO.**
1.		
2.		
3.		

PROGRAMME	
SECTION / GROUP	
DATE	
NAME OF INSTRUCTOR(S)	**1.**
	2.

EXAMINER’S COMMENT(S)	**TOTAL MARKS**

Rev. No.	Date	Author(s)	Description
1.0	30 JAN 2019	1. Shamsul Fakhar 2. Noor Mohd Ariff	1. Update to new UTeM logo 2. Update faculty's name 3. Change "course" to "programme" 4. Remove verification stamp

1. LEARNING OUTCOMES

1. Differentiate the functionality among various kinds of OS components.
2. Manipulate OS theories to solve basic functional problems.
3. Perform lab and present technical report in writing.

2. REQUIREMENTS

1. PC with Linux Ubuntu installed (or any other POSIX-compliant system)
2. Knowledge of C programming language

3. SYNOPSIS & THEORY

An application program makes a system call to get the operating system to perform a service for it, like reading from a file or displaying file listings inside a directory. One nice thing about system call is that you don't have to link with a C library, so your executables can be much smaller. System calls are sometimes called kernel calls. By making system calls, commands that you usually need to manually type into terminal and press Enter to execute, can be run automatically run from a C program.

So far, all the programs we have written can be run with a single command. For example, if we compile an executable called `myprog`, we can run it from within the same directory with the command `./myprog`. But what if you want to pass information from the command line to the program you are running? Command-line arguments are very useful. After all, C functions wouldn't be very useful if you couldn't ever pass arguments to them; adding the ability to pass arguments to programs makes them that much more useful. In fact, all the arguments you pass on the command line end up as arguments to the main function in your program.

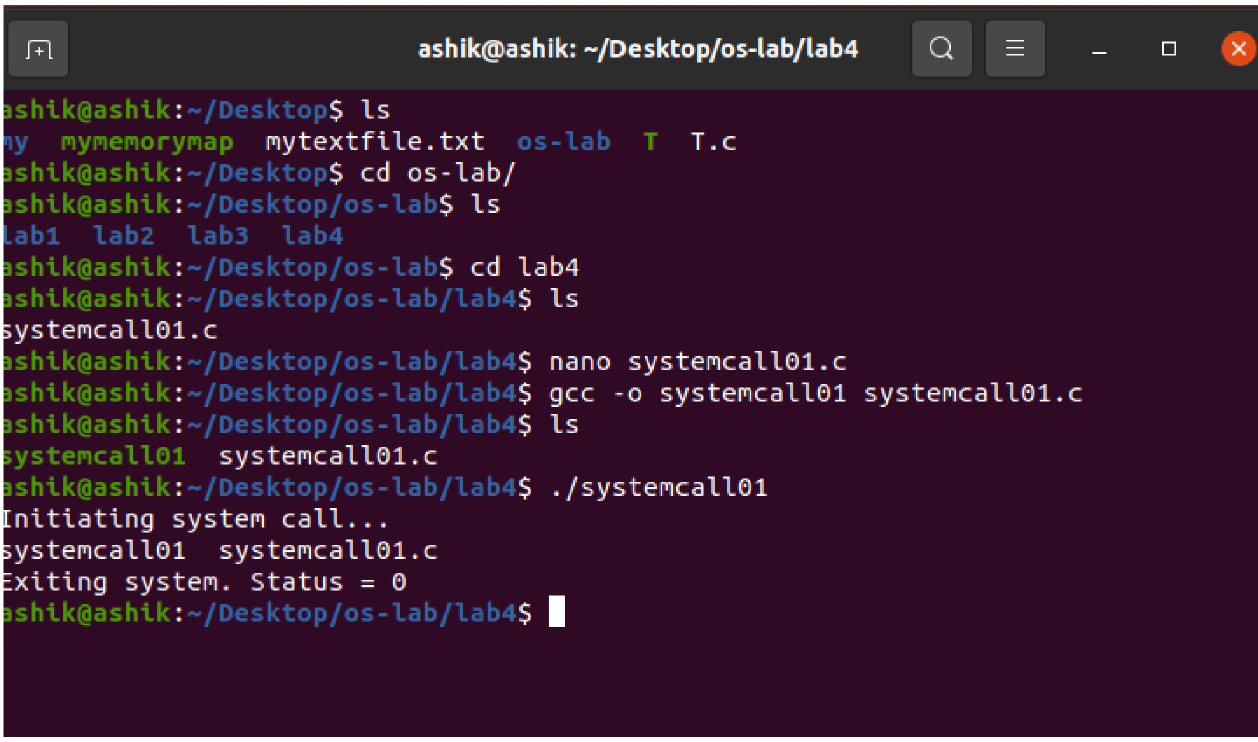
4. PROCEDURE

PART 1: System Calls

1. In terminal open a **text editor**.
2. **Save** the file as "systemcall01.c" on your desktop.
3. **Write** the code below into "systemcall01.c":

```
#include <stdio.h>
#include <stdlib.h>

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

    printf("Initiating system call...\n");

    // this can be any Linux command:
status = system("ls");

    printf("Exiting system. Status = %d\n", status);

    return 0;
}
```

4. **Compile** the program.
5. **Run** the output file.
6. **Observe** the output. This program effectively runs the "`ls`" command programmatically instead of having the user type it manually from the terminal. Status will always be 0 if program is run with no error.

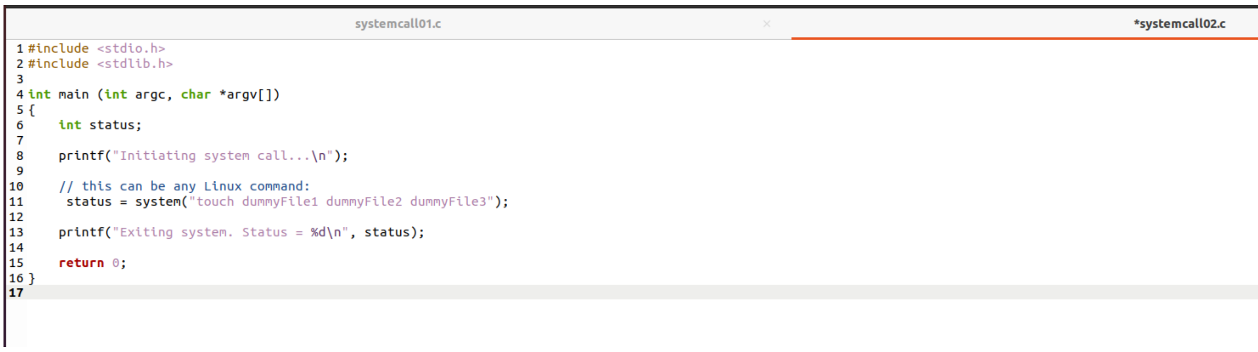


```
ashik@ashik: ~/Desktop/os-lab/lab4

ashik@ashik:~/Desktop$ ls
my  mymemorymap  mytextfile.txt  os-lab  T  T.c
ashik@ashik:~/Desktop$ cd os-lab/
ashik@ashik:~/Desktop/os-lab$ ls
lab1  lab2  lab3  lab4
ashik@ashik:~/Desktop/os-lab$ cd lab4
ashik@ashik:~/Desktop/os-lab/lab4$ ls
systemcall01.c
ashik@ashik:~/Desktop/os-lab/lab4$ nano systemcall01.c
ashik@ashik:~/Desktop/os-lab/lab4$ gcc -o systemcall01 systemcall01.c
ashik@ashik:~/Desktop/os-lab/lab4$ ls
systemcall01  systemcall01.c
ashik@ashik:~/Desktop/os-lab/lab4$ ./systemcall01
Initiating system call...
systemcall01  systemcall01.c
Exiting system. Status = 0
ashik@ashik:~/Desktop/os-lab/lab4$
```

7. Let's add a few more files to the desktop so that we can properly see the file listing.

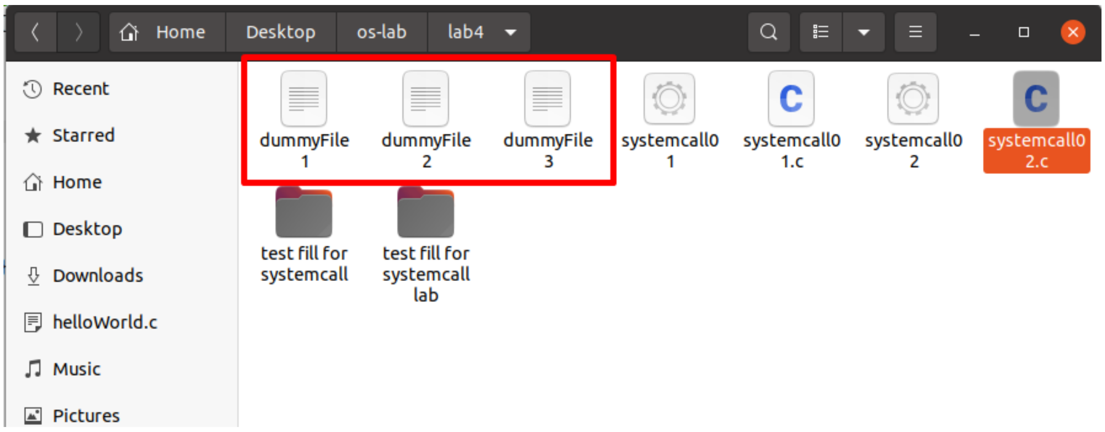
8. Create a new file named "**systemcall02.c**" on your desktop.

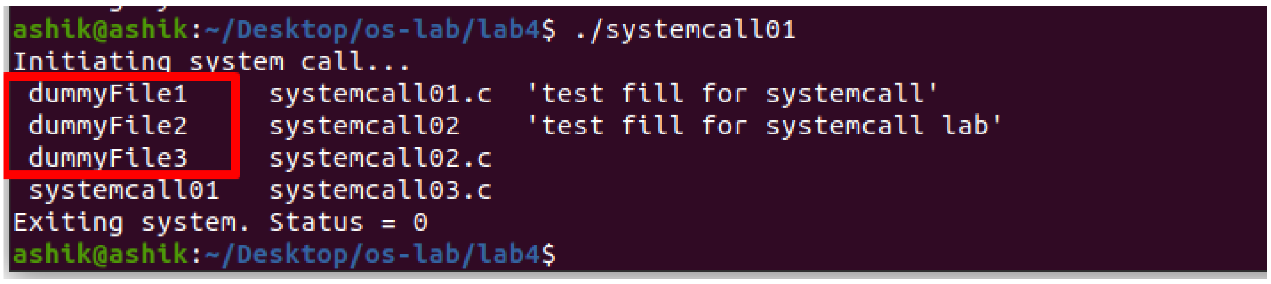
9. Write the code as in step 3, but instead of running the "`ls`" command, this time change it to execute "**`touch dummyFile1 dummyFile2 dummyFile3`**".



```c
#include <stdio.h>
#include <stdlib.h>

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

    printf("Initiating system call...\n");

    // this can be any Linux command:
     status = system("touch dummyFile1 dummyFile2 dummyFile3");

    printf("Exiting system. Status = %d\n", status);

    return 0;
}
```

10. This will create 3 empty files named dummyFile1, dummyFile2 and dummyFilei3.



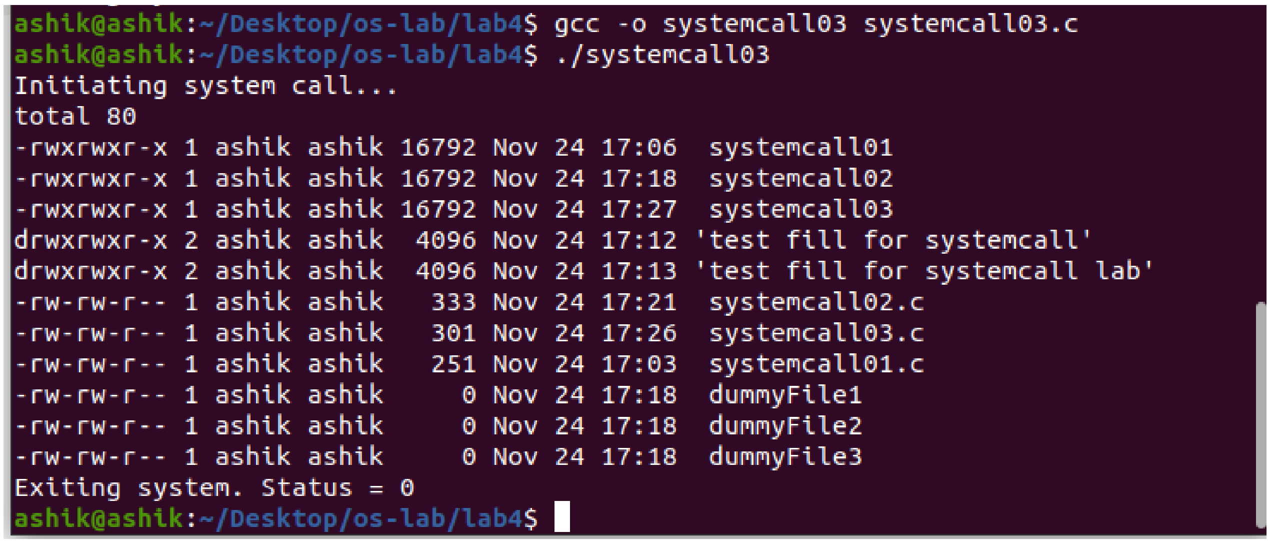


```
ashik@ashik:~/Desktop/os-lab/lab4$ ./systemcall01
Initiating system call...
 dummyFile1    systemcall01.c  'test fill for systemcall'
 dummyFile2    systemcall02    'test fill for systemcall lab'
 dummyFile3    systemcall02.c
 systemcall01  systemcall03.c
Exiting system. Status = 0
ashik@ashik:~/Desktop/os-lab/lab4$
```

11. Create a new file named "**systemcall03.c**" on your desktop.

12. Write the code as in step 3, but instead of running the `"ls"` command, this time change it to execute **`"ls -lS"`**.

13. This will list down all the files on the desktop and arrange it according it to file size.



```
ashik@ashik:~/Desktop/os-lab/lab4$ gcc -o systemcall03 systemcall03.c
ashik@ashik:~/Desktop/os-lab/lab4$ ./systemcall03
Initiating system call...
total 80
-rwxrwxr-x 1 ashik ashik 16792 Nov 24 17:06  systemcall01
-rwxrwxr-x 1 ashik ashik 16792 Nov 24 17:18  systemcall02
-rwxrwxr-x 1 ashik ashik 16792 Nov 24 17:27  systemcall03
drwxrwxr-x 2 ashik ashik  4096 Nov 24 17:12 'test fill for systemcall'
drwxrwxr-x 2 ashik ashik  4096 Nov 24 17:13 'test fill for systemcall lab'
-rw-rw-r-- 1 ashik ashik   333 Nov 24 17:21  systemcall02.c
-rw-rw-r-- 1 ashik ashik   301 Nov 24 17:26  systemcall03.c
-rw-rw-r-- 1 ashik ashik   251 Nov 24 17:03  systemcall01.c
-rw-rw-r-- 1 ashik ashik     0 Nov 24 17:18  dummyFile1
-rw-rw-r-- 1 ashik ashik     0 Nov 24 17:18  dummyFile2
-rw-rw-r-- 1 ashik ashik     0 Nov 24 17:18  dummyFile3
Exiting system. Status = 0
ashik@ashik:~/Desktop/os-lab/lab4$
```

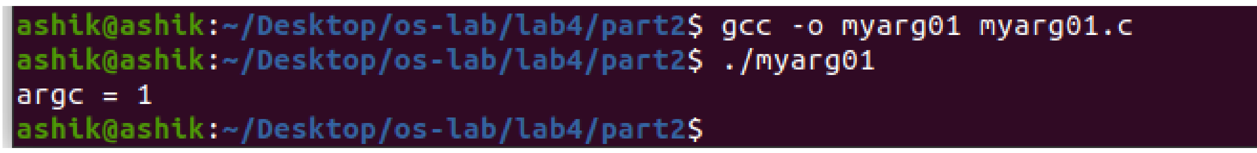
PART 2: Argument Handling

1. Write the code below into a new file "myarg01.c":

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("argc = %d\n", argc);

    return 0;
}
```

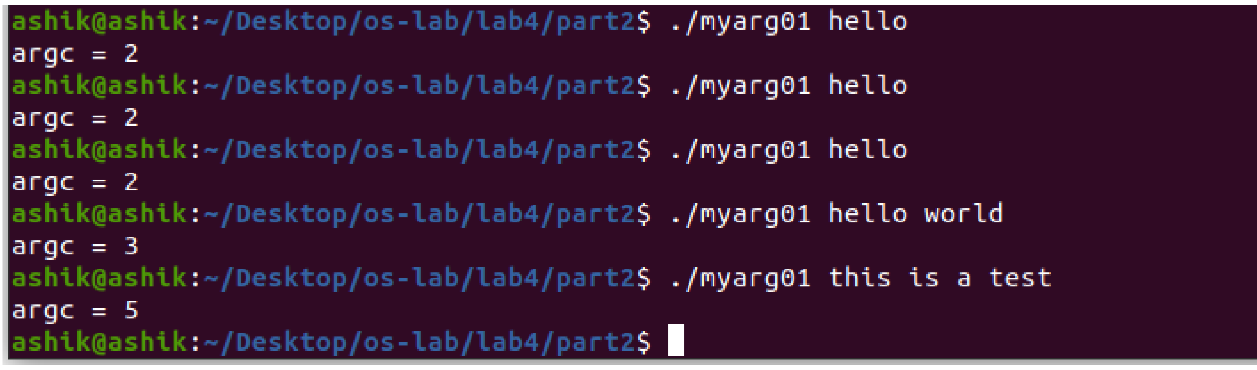
2. **Compile** the program.

3. **Run** the output file.


```
ashik@ashik:~/Desktop/os-lab/lab4/part2$ gcc -o myarg01 myarg01.c
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg01
argc = 1
ashik@ashik:~/Desktop/os-lab/lab4/part2$
```

4. What exactly does the variable **argc** represents?

5. Try running the program again by giving a few extra parameters such as **"./myarg01 hello"** **or "./myarg01 hello world"** or **"./myarg01 this is a test"**.


```
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg01 hello
argc = 2
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg01 hello
argc = 2
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg01 hello
argc = 2
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg01 hello world
argc = 3
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg01 this is a test
argc = 5
ashik@ashik:~/Desktop/os-lab/lab4/part2$ 
```

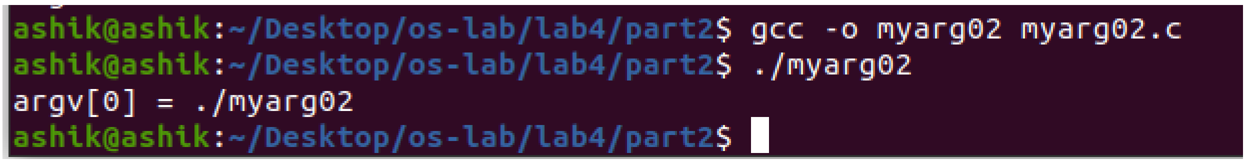
6. Now do you get what **argc** is for?

Yeah, it show number of space I type.

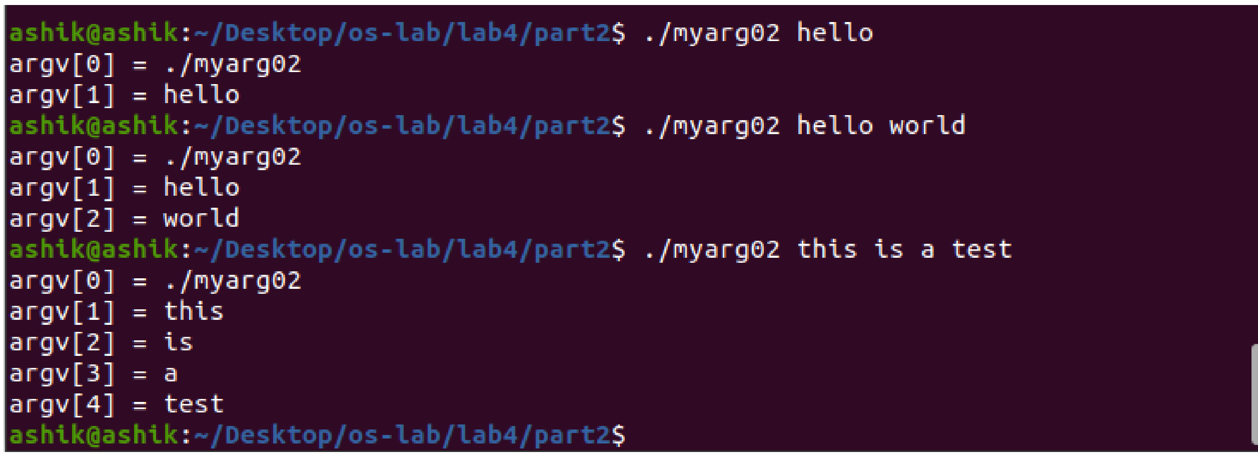
```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=0; i<argc; i++)
    {
        printf("argv[%d] = %s\n", i, argv[i]);
    }

    return 0;
}
```

7. **Compile** the program.

8. **Run** the program.


```
ashik@ashik:~/Desktop/os-lab/lab4/part2$ gcc -o myarg02 myarg02.c
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg02
argv[0] = ./myarg02
ashik@ashik:~/Desktop/os-lab/lab4/part2$ 
```

9. Try running the program again by giving a few extra parameters such as **"./myarg02 hello"** or **"./myarg02 hello world"** or **"./myarg02 this is a test"**.



```
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg02 hello
argv[0] = ./myarg02
argv[1] = hello
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg02 hello world
argv[0] = ./myarg02
argv[1] = hello
argv[2] = world
ashik@ashik:~/Desktop/os-lab/lab4/part2$ ./myarg02 this is a test
argv[0] = ./myarg02
argv[1] = this
argv[2] = is
argv[3] = a
argv[4] = test
ashik@ashik:~/Desktop/os-lab/lab4/part2$
```

10. What exactly does the variable **argv** represents?

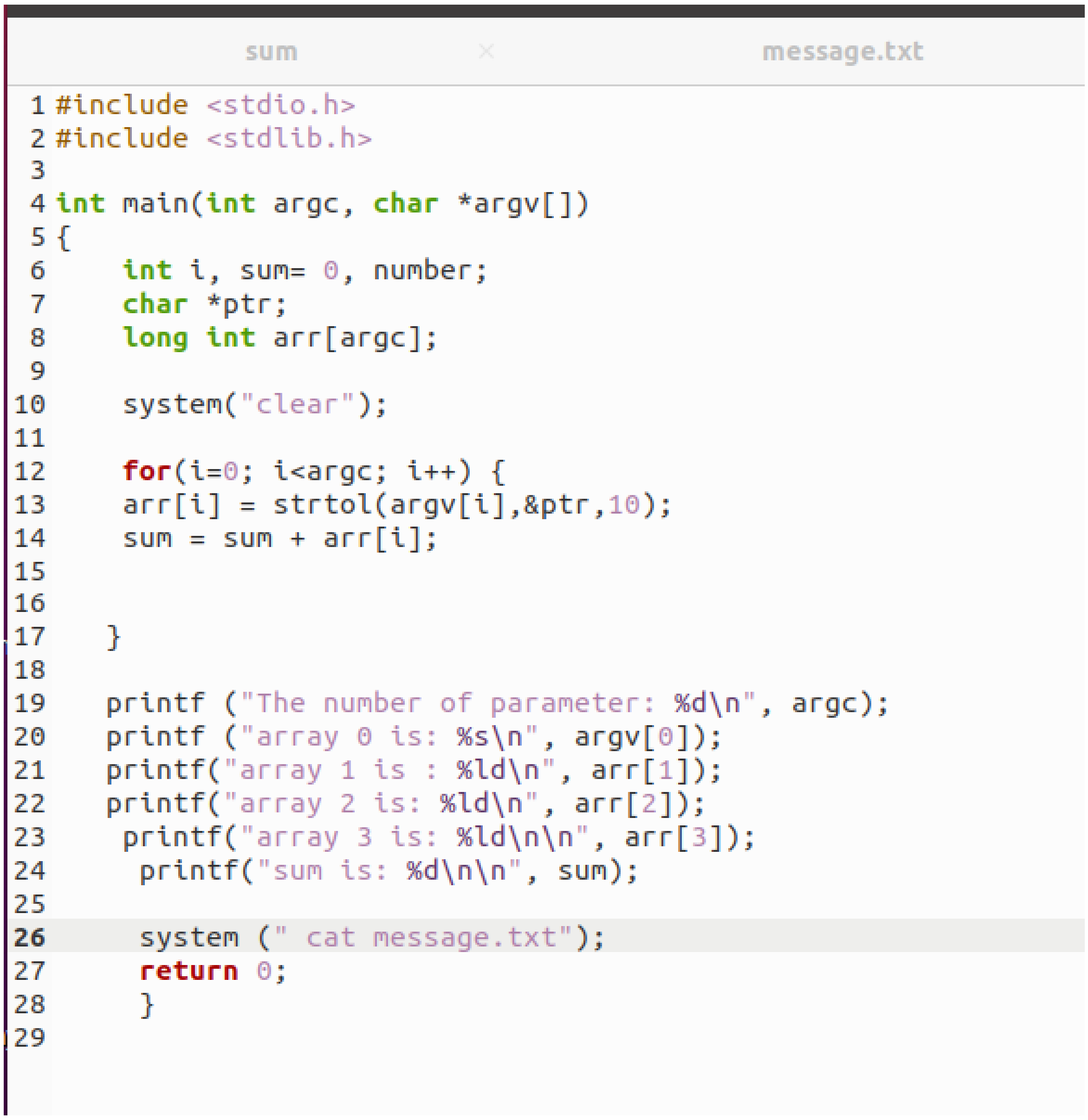
PART 3: Complete The Task Below

Create a text file named **"message.txt"** on the desktop. Inside the file, write the message **"Program completed successfully."**

Write a **C program** that **sums up** all the integer arguments supplied by user. Arguments supplied are all **numbers**. Display the final **sum** to console screen. Before doing any calculations, run the command to programmatically **clear** all display on screen. After the total sum is displayed on screen, run the command to programmatically show the **contents** inside of "**message.txt**" to screen.

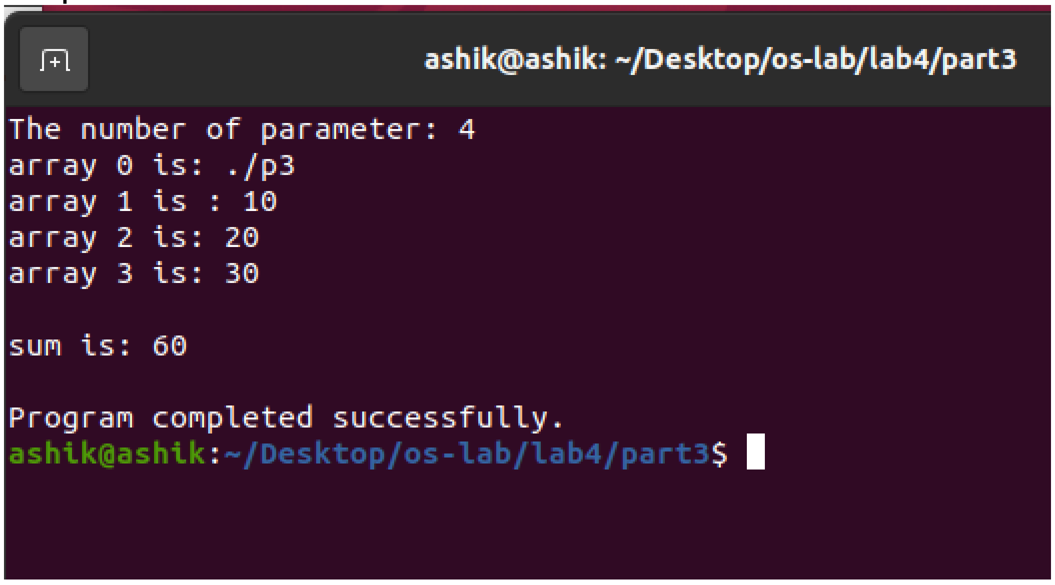
Your **output** should look like this:

```
argv[0] = ./mytask
argv[1] = 10
argv[2] = 30
argv[3] = 10

Sum is 50

Program completed successfully.

shamsul@shamsul-HP-Compaq-dc5850-Microtower:~/Desktop$
```



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i, sum= 0, number;
    char *ptr;
    long int arr[argc];

    system("clear");

    for(i=0; i<argc; i++) {
    arr[i] = strtol(argv[i],&ptr,10);
    sum = sum + arr[i];


   }

   printf ("The number of parameter: %d\n", argc);
   printf ("array 0 is: %s\n", argv[0]);
   printf("array 1 is : %ld\n", arr[1]);
   printf("array 2 is: %ld\n", arr[2]);
    printf("array 3 is: %ld\n\n", arr[3]);
     printf("sum is: %d\n\n", sum);

     system (" cat message.txt");
     return 0;
     }

```

Output



```
The number of parameter: 4
array 0 is: ./p3
array 1 is : 10
array 2 is: 20
array 3 is: 30

sum is: 60

Program completed successfully.
ashik@ashik:~/Desktop/os-lab/lab4/part3$ 
```

5. QUESTIONS

1. What is the advantage of running command lines programmatically instead of manually?

   One of the main advantages of a command line interface is that it allows users to type in commands that can produce immediate results.

   advantages of a command-line interface include:

   - If you know the commands, a CLI can be a lot faster and efficient than any other type of interface. It can also handle repetitive tasks easily.
   - A CLI requires less memory to use in comparison to other interfaces. It also does not use as much CPU processing time as other interfaces.
   - A CLI doesn't require Windows and a low-resolution monitor can be used. Meaning, it needs fewer resources, yet is highly precise.

   While in running program lines manually it will take you much more time to finish your work.

2. What does the "`touch`" command do?

   In computing, **touch** is a command used to update the access date and/or modification date of a computer file or directory.

3. What does the "`ls -lS`" command do?

The **ls** command is used to list files. "ls" on its own lists all files in the current directory except for hidden files. "ls *.tex" lists only those files ending in ".tex". There are a large number of options; here are some of the most useful. Options can be combined (this is a general principle of Unix commands) - for example "ls -la" gives a long listing of all files.

4. What is "`argc`"?

argc (ARGument Count) is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name). The value of argc should be non negative.

5. What is "`argv`"?

argv(ARGument Vector) is array of character pointers listing all the arguments. If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings. Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments

6. DISCUSSIONS & CONCLUSION

(What have you learned from this experiment?)