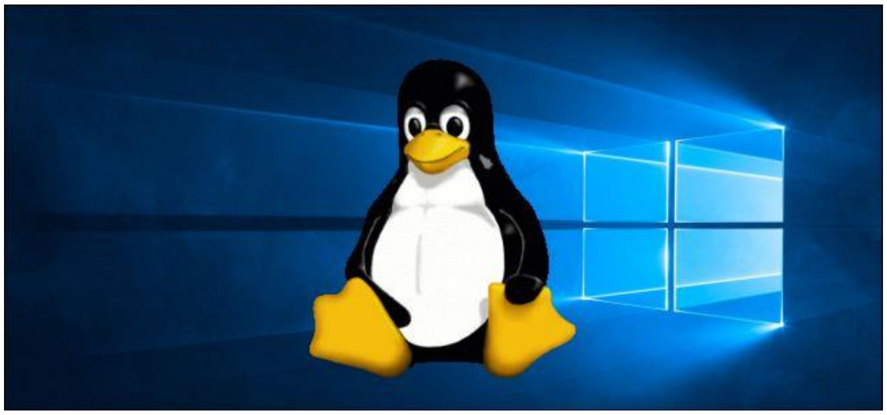
UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING

SOFTWARE ENGINEERING DEPARTMENT



Operating Systems

Experiment 09

C Programming in Linux & exec() System Call Variants*

CLO 2. Use modern tools and languages.
CLO 3. Demonstrate an original solution of problem under discussion.
CLO 4. Work individually as well as in teams

I. Command Line Arguments in C:

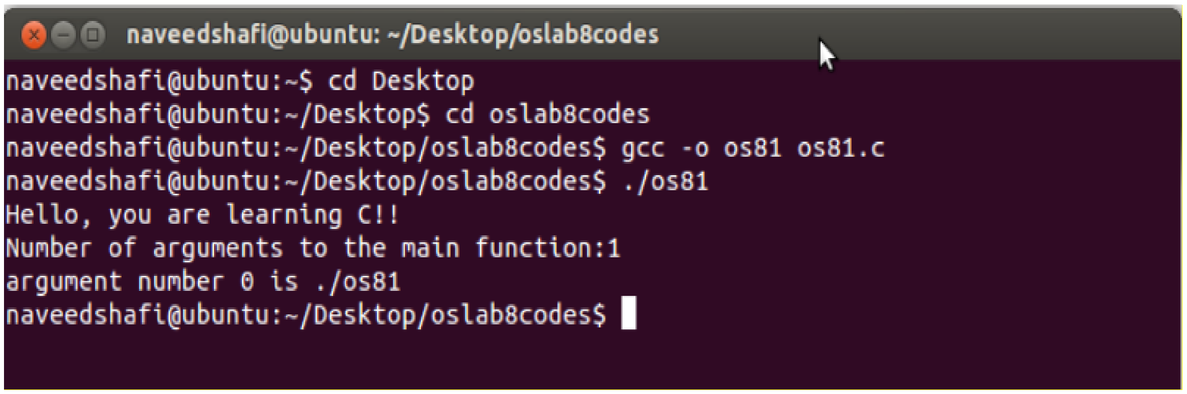
C language is like **Latin-** it is finite and has not changed for years. C is built right into the core of UNIX and Linux. C remains particularly popular in the world of Unix-like operating systems, and, for example, most of the Linux *kernel* (i.e., the core of the operating system) is written in C.

Program 9-1: Arguments to main function.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
      int i=0;
      printf("Hello, you are learning C!!\n");
      printf("Number of arguments to the main function:%d\n", argc);
      for(i=0;i<argc;i++)
      {
            printf("argument number %d is %s\n", i, argv[i]);
      }
        return 0;
}
```

- ✓ The "**`int main(int argc, char *argv[])`**" part is the start of the actual program. This is an entry point.

- ✓ The "**`int argc`**" is an argument to the function "main" which is an integer of the number of character string arguments passed in "**`char *argv[]`**" (a list of pointers to character strings) that might be passed at the command line when we run it.

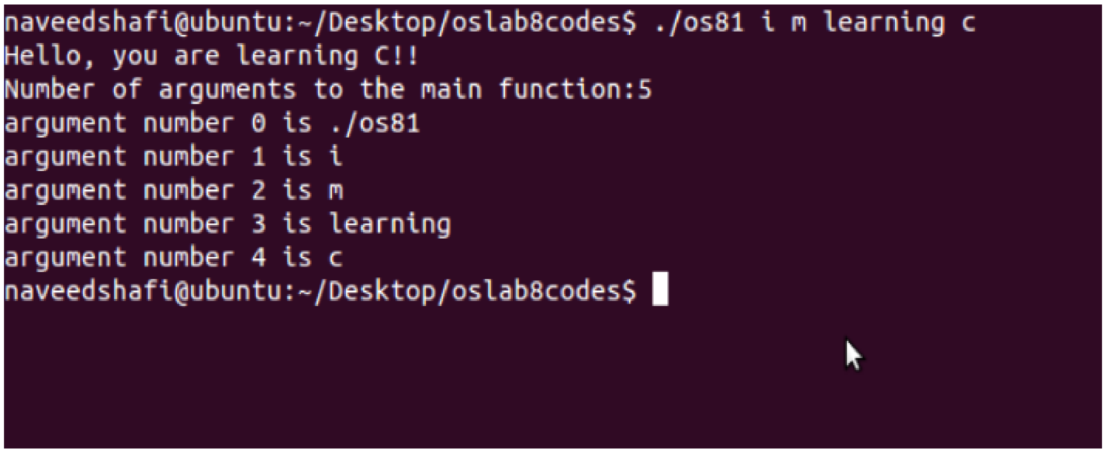


```
naveedshafi@ubuntu: ~/Desktop/oslab8codes
naveedshafi@ubuntu:~$ cd Desktop
naveedshafi@ubuntu:~/Desktop$ cd oslab8codes
naveedshafi@ubuntu:~/Desktop/oslab8codes$ gcc -o os81 os81.c
naveedshafi@ubuntu:~/Desktop/oslab8codes$ ./os81
Hello, you are learning C!!
Number of arguments to the main function:1
argument number 0 is ./os81
naveedshafi@ubuntu:~/Desktop/oslab8codes$
```

Program 9-2: Environment variables.

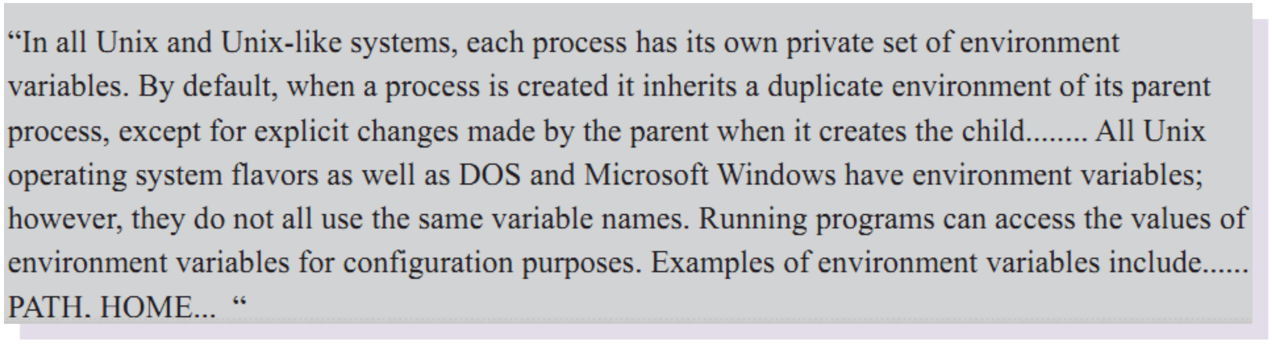
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
{
     int i=0;
     printf("Hello, you are learning C!!\n");
     printf("Number of arguments to the main function:%d\n", argc);
     for(i=0;i<argc;i++)
     {
          printf("argument number %d is %s\n", i, argv[i]);
     }
     i = 0;
     printf("Environment variables:\n");
     while(env[i])
     {
          printf("env[%d] = %s\n", i, env[i]);
          i++;
     }
     return 0;
}
```



```
naveedshafi@ubuntu:~/Desktop/oslab8codes$ ./os81 i m learning c
Hello, you are learning C!!
Number of arguments to the main function:5
argument number 0 is ./os81
argument number 1 is i
argument number 2 is m
argument number 3 is learning
argument number 4 is c
naveedshafi@ubuntu:~/Desktop/oslab8codes$
```

- ✓ **What Are Environment Variable:**



"In all Unix and Unix-like systems, each process has its own private set of environment variables. By default, when a process is created it inherits a duplicate environment of its parent process, except for explicit changes made by the parent when it creates the child........ All Unix operating system flavors as well as DOS and Microsoft Windows have environment variables; however, they do not all use the same variable names. Running programs can access the values of environment variables for configuration purposes. Examples of environment variables include...... PATH. HOME... "

Figure 1: Environment variables defined by Wikipedia

II. *exec*()* System Call:

- Typically, the exec system call is used after a fork system call by one of the two processes to replace the process' memory space with a new executable program.
- The new process image is constructed from an ordinary, executable file.
- There can be no return from a successful exec because the calling process image is overlaid by the new process image.
- Commonly a process generates a child process because it would like to transform the child process by changing the program code the child process is executing.
- The text, data and stack segment of the process are replaced and only the **u** (user) area of the process remains the same.
- There are **six variations** of "exec" system call.

```
 #include <unistd.h>

 extern char **environ;

 int execv(const char *path, char *const argv[]);
 int execvp(const char *file, char *const argv[]);
 int execve(const char *path, char *const argv[], char *const envp[]);
 int execl(const char *path, const char *arg, ... /*(char *) 0*/);
 int execlp(const char *file, const char *arg, ... /*(char *) 0*/);
 int execle(const  char  *path,  const  char  *arg  , ... /*(char *)
0*/, char * const envp[]);
```

The naming convention:

- ✓ **'l'** indicates a list arrangement (a series of null terminated arguments)
- ✓ **'v'** indicates the array or vector arrangement.
- ✓ **'e'** indicates the programmer will construct (in the array/vector format) and pass their own environment variable list.
- ✓ **'p'** indicates the current PATH string should be used when the system searches for executable files.

NOTE:

- ✓ In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

exec system call functionality:

Library Call Name	**Argument List**	**Pass Current Environment Variables**	**Search PATH automatic?**
execl	list	yes	no
execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

execl

- This system call is used when the number of arguments to be passed to the program to be executed is known in advance

execv

- This system call is used when the numbers of arguments for the program to be executed is dynamic.

DESCRIPTION

- ✓ When a C-language program is executed as a result of *exec*()* call, it shall be entered as a C-language function call as follows:

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

- ✓ Where **`argc`** is the argument count and **`argv`** is an array of character pointers to the arguments themselves. In addition, the following variable:

- ✓ **`extern char **environ;`**
  is initialized as a pointer to an array of character pointers to the environment strings. The **`argv`** and **`environ`** arrays are each terminated by a null pointer. The null pointer terminating the **`argv`** array is not counted in **`argc`**.

exec system call Arguments:

1st Argument: Either *file* or *path*

- ✓ ***file***
  It is the filename of the file that contains the executable image of the new process.

- ✓ ***path***
  It identifies the location of the new process image within the hierarchical file system (HFS). If the `path` argument contains a slash (/), it is assumed that either an absolute or a relative pathname has been specified. If the `path` argument does not contain a slash, the directories specified by the `PATH` environment variable are searched in an attempt to locate the file.

2nd Argument: Either argv[] or argument list.

- ✓ ***argv***
  It is a pointer to an array of pointers to null-terminated character strings. A NULL pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The **first argument, argv[0],** is required and must contain the name of the executable file for the new process image.
  .
- ✓ ***arg0, ..., NULL***
  It is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a NULL pointer. The first argument, arg0, is required and must contain the name of the executable

file for the new process image. **If the new process image is a normal C `main` program, the list of arguments will be passed to `argv` as a pointer to an array of strings. The number of strings in the array is passed to the `main()` function as `argc`.**

3rd Argument:

- ✓ ***envp***

  It is a pointer to an array of pointers to null-terminated character strings. A `NULL` pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an **`environment variable`** to the new process image.

Things to remember about exec*:

- ✓ This system call simply replaces the current process with a new program -- the pid does not change
- ✓ The exec () is issued by the calling process and what is expected is referred to as the new program -- not the new process since no new process is created
- ✓ It is important to realize that control is not passed back to the calling process unless an error occurred with the exec () call
- ✓ In the case of an error, the exec () returns a value back to the calling process
- ✓ If no error occurs, the calling process is lost .

1. **EXAMPLE (*execv Usage*):**

The following example illustrates the use of `execv` to execute the `ls` shell command:

Program 9-3: using execv to execute ls command

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>

main()
{
   pid_t pid;
   char *const parmList[] = {"/bin/ls", "-l", NULL};

   if ((pid = fork()) == -1)
      perror("fork error");
   else if (pid == 0) {
      execv("/bin/ls", parmList);
      printf("Return not expected. Must be an execv error.n");
   }
}
```

2. EXAMPLE (*execvp Usage*):

The following example illustrates the use of `execvp` to execute the `ls` shell command:

Program 9-4: using execvp to execute ls command

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
main()
{
   pid_t pid;
   char *const parmList[] = {"/bin/ls", "-l", "/", NULL};

   if ((pid = fork()) == -1)
   perror("fork() error");
   else if (pid == 0) {
      execvp("ls", parmList);
      printf("Return not expected. Must be an execvp() error.n");
   }
}
```

3. EXAMPLE (*execl Usage*):

The following example illustrates the use of `execl` to execute another program (sum.c)

Program 9-5: To add two numbers taken as argument from command line. (sum.c)

```
#include<stdio.h>
#include<stdlib.h>
void main(int argc , char * argv[])
{
    int i,sum=0;
    if(argc!=3)
      {
      printf("Arguments not valid!");
      exit(1);
      }
    printf("The sum is : ");
    for(i=1;i<argc;i++)
        sum = sum + atoi(argv[i]);
    printf("%d",sum);
}
```

Program 9-6: Child is overlaid by "Program 9-5" using execl command after fork.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int main()
{
 pid_t pid;
 pid=fork();
 if(pid<0)
{
 perror("fork error");
 return 1;
}
 else if(pid ==0)
{
if (execl("/home/sidra/lab/sum","sum","5","6", NULL) <0 )
{
 perror("child fail to execute");
 return 1;}}
return 0;
}
```

UNIVERSITY OF ENGINEERING AND TECHNOLOGY, TAXILA

FACULTY OF TELECOMMUNICATION AND INFORMATION ENGINEERING

SOFTWARE ENGINEERING DEPARTMENT

TASK 1:

➔ Compile and execute the "Program 9-1" and "Program 9-2" as given below. Examine the output.
- gcc –o args 9-1.c
- ./args Operating system C programming

TASK 2:

➔ Compile and execute the "Program 9-3"and "Program 9-4". What is the difference between them?

➔ Compile and execute the "Program 9-5"and "Program 9-6".
- Firstly compile "Program 9-5" with the object name as "sum".
- In "Program 9-6", use exec() system call instead of excel(), and then compile and execute

TASK 3:

➔ Keeping in mind "Program 9-5", multiply 4 integers taken from command line.

➔ Keeping in mind "Program 9-5" & "Program 9-6", Compute power of 2 (2^n where n is taken as argument from exec system call).