**Operating Systems Principles and Programming Tutorial IV Semester Div A & F**

**Linux Exec System Call**

The exec system call is used to execute a file which is residing in an active process. When exec is called the previous executable file is replaced and new file is executed.

More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.

The user data segment which executes the exec() system call is replaced with the data file whose name is provided in the argument while calling exec().

The new program is loaded into the same process space. The current process is just turned into a new process and hence the process id PID is not changed, this is because we are not creating a new process we are just replacing a process with another process in exec.

If the currently running process contains more than one thread then all the threads will be terminated and the new process image will be loaded and then executed. There are no destructor functions that terminate threads of current process.

PID of the process is not changed but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process. The new process is executed from the entry point.

Exec system call is a collection of functions and in C programming language, the standard names for these functions are as follows:

1. **execl**
2. **execle**
3. **execlp**
4. **execv**
5. **execve**
6. **execvp**

It should be noted here that these functions have the same base *exec*followed by one or more letters. These are explained below:

**e:** It is an array of pointers that points to environment variables and is passed explicitly to the newly loaded process.

**l:** l is for the command line arguments passed a list to the function

**p:** p is the path environment variable which helps to find the file passed as an argument to be loaded into process.

**v:** v is for the command line arguments. These are passed as an array of pointers to the function.

**Why exec is used?**

exec is used when the user wants to launch a new file or program in the same process.

**Inner Working of exec**

Consider the following points to understand the working of exec:

1. Current process image is overwritten with a new process image.
2. New process image is the one you passed as exec argument
3. The currently running process is ended
4. New process image has same process ID, same environment, and same file descriptor (because process is not replaced process image is replaced)
5. The CPU stat and virtual memory is affected. Virtual memory mapping of the current process image is replaced by virtual memory of new process image.

**Syntaxes of exec family functions:**

The following are the syntaxes for each function of exec:

int execl(const char\* path, const char\* arg, …)  
int execlp(const char\* file, const char\* arg, …)  
int execle(const char\* path, const char\* arg, …, char\* const envp[])  
int execv(const char\* path, const char\* argv[])  
int execvp(const char\* file, const char\* argv[])  
int execvpe(const char\* file, const char\* argv[], char \*const envp[])

**Description:**

The return type of these functions is Int. When the process image is successfully replaced nothing is returned to calling function because the process that called it is no longer running. But if there is any error -1 will be returned. If any error is occurred an *errno*is set.

In the syntax:

1. ***path***is used to specify the full path name of the file which is to be executes.
2. ***arg***is the argument passed. It is actually the name of the file which will be executed in the process. Most of the times the value of arg and path is same.
3. ***const char\* arg***in functions execl(), execlp() and execle() is considered as arg0, arg1, arg2, …, argn. It is basically a list of pointers to null terminated strings. Here the first argument points to the filename which will be executed as described in point 2.
4. ***envp***is an array which contains pointers that point to the environment variables.
5. ***file*** is used to specify the path name which will identify the path of new process image file.
6. The functions of exec call that end with ***e*** are used to change the environment for the new process image. These functions pass list of environment setting by using the argument ***envp***. This argument is an array of characters which points to null terminated String and defines environment variable.

To use the exec family functions, you need to include the following header file in your C program:

*#include <unistd.h>*

**Example 1: Using exec system call in C program**

Consider the following example in which we have used exec system call in C programming in Linux, Ubuntu: We have two c files here example.c and hello.c:

**example.c**

**CODE:**

*#include <stdio.h>*  
*#include <unistd.h>*  
*#include <stdlib.h>*  
int main()  
{  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("PID of example.c = %d\n", getpid());  
    char \*args[] = {"Hello", "C", "Programming", NULL};  
    execv("./hello", args);  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Back to example.c");  
    return 0;  
}

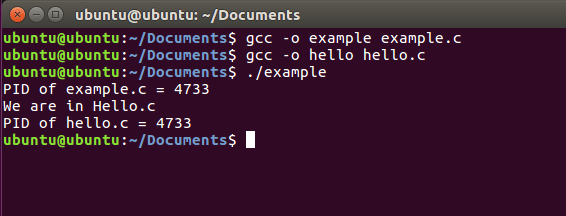
**hello.c**

**CODE:**

*#include <stdio.h>*  
*#include <unistd.h>*  
*#include <stdlib.h>*  
int main(int argc, char \*argv[])  
{  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("We are in Hello.c\n");  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("PID of hello.c = %d\n", getpid());  
    return 0;  
}

**OUTPUT:**

PID of example.c = 4733  
We are in Hello.c  
PID of hello.c = 4733

[](https://linuxhint.com/wp-content/uploads/2018/01/strong.png)

In the above example we have an example.c file and hello.c file. In the example .c file first of all we have printed the ID of the current process (file example.c is running in current process). Then in the next line we have created an array of character pointers. The last element of this array should be NULL as the terminating point.

Then we have used the function execv() which takes the file name and the character pointer array as its argument. It should be noted here that we have used ./ with the name of file, it specifies the path of the file. As the file is in the folder where example.c resides so there is no need to specify the full path.

When execv() function is called, our process image will be replaced now the file example.c is not in the process but the file hello.c is in the process. It can be seen that the process ID is same whether hello.c is process image or example.c is process image because process is same and process image is only replaced.

Then we have another thing to note here which is the printf() statement after execv() is not executed. This is because control is never returned back to old process image once new process image replaces it. The control only comes back to calling function when replacing process image is unsuccessful. (The return value is -1 in this case).

**Difference between fork() and exec() system calls:**

The fork() system call is used to create an exact copy of a running process and the created copy is the child process and the running process is the parent process. Whereas, exec() system call is used to replace a process image with a new process image. Hence there is no concept of parent and child processes in exec() system call.

In fork() system call the parent and child processes are executed at the same time. But in exec() system call, if the replacement of process image is successful, the control does not return to where the exec function was called rather it will execute the new process. The control will only be transferred back if there is any error.

**Example 2: Combining fork() and exec() system calls**

Consider the following example in which we have used both fork() and exec() system calls in the same program:

**example.c**

**CODE:**

*#include <stdio.h>*  
*#include <sys/types.h>*  
*#include <unistd.h>*  
int main()  
{  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("PID of example.c = %d\n", getpid());  
    pid\_t p;  
    p = fork();  
    if(p==-1)  
    {  
        [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("There is an error while calling fork()");  
    }  
    if(p==0)  
    {  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("We are in the child process\n");  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Calling hello.c from child process\n");  
    char \*args[] = {"Hello", "C", "Programming", NULL};  
    execv("./hello", args);  
    }  
    else  
    {  
        [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("We are in the parent process");  
    }  
    return 0;  
}

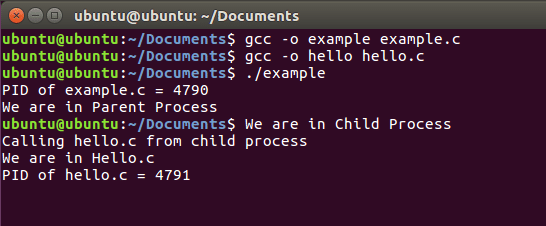
**hello.c:**

**CODE:**

*#include <stdio.h>*  
*#include <unistd.h>*  
*#include <stdlib.h>*  
int main(int argc, char \*argv[])  
{  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("We are in Hello.c\n");  
    [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("PID of hello.c = %d\n", getpid());  
    return 0;  
}

OUTPUT:

PID of example.c = 4790  
We are in Parent Process  
We are in Child Process  
Calling hello.c from child process  
We are in hello.c  
PID of hello.c = 4791

[](https://linuxhint.com/wp-content/uploads/2018/01/code-1.png)

In this example we have used fork() system call. When the child process is created 0 will be assigned to p and then we will move to the child process. Now the block of statements with if(p==0) will be executed. A message is displayed and we have used execv() system call and the current child process image which is example.c will be replaces with hello.c. Before execv() call child and parent processes were same.

It can be seen that the PID of example.c and hello.c is different now. This is because example.c is the parent process image and hello.c is the child process image. 