

Operating Systems Lab



Lab # 07

Exec System Call

Instructor: Engr. Muhammad Usman

Email: usman.rafiq@nu.edu.pk

Course Code: CL2006

Department of Computer Science,
National University of Computer and Emerging Sciences FAST
Peshawar Campus

Contents

Processes Running States	2
exec family of functions in C	4
Example: Using exec system call in C program	6
example.c	6
hello.c	7
Difference between fork and exec:	9
Example 2: Combining fork() and exec() system calls	9
example.c	9
hello.c:	11

Processes Running States

The job of the `exec()` call is to replace the current process with a new process/program. Once the `exec()` call is made, the current process is “gone” and a new process starts. The `exec()` call is actually a family of 6 system calls. Difference between all 6 can be seen from **man 3 exec**:

- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execlx(const char *path, const char *arg, char *const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execve(const char *path, const char *argv[], char *const envp[]);`
- `int execvp(const char *_le, char *const argv[]);`

Following is usage of one flavor of Exec, the `execv()` system call:

execvp.c

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    printf("X\n");
    char *av[] = {"ls", "-al", "/", 0};    //char *x ----> A character pointer is essentially a string
    execvp("ls", av);    // Here ls is the name of the program
    printf("X\n");
}
```

Here in this program “**ls**” is the name of the program and “**-al**” is the argument. These are called command line arguments.

```
char *av[] = {"ls", "-al", "/", 0};
```

Here “ls” is the program which you have to run.

Here “**-al**”, “**/**”, “**0**” all are command line arguments parameters or program parameters.

execv.c

```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int p;
    char *arg[] = {"/usr/bin/ls", 0};
    p=fork();
    if (p == 0)
```

```

{
printf("Child Process\n");
execv(arg[0], arg);
printf("Child Process\n");
}
else
{
printf("Parent Process\n");
}
}

```

A path that starts with / starts in the root directory. For example, /usr/bin/ls is a file called ls in the directory called bin in the directory called usr in the root directory. (It is the executable code for the ls command.)

Again, look at the code. We have referred to Exec() system call but in code we see execv(). Read man pages for this and find out. To help you understand, try finding answers to the following:

Question 1: What is the 1st argument to the execv() call? What is its contents?

Question 2: What is the 2nd argument to it? What is its contents?

Question 3: What is arg?

Question 4: Look at the code of the child process (p==0). How many times does the statement "Child Process" appear? Why?

Diagrammatically, the functioning of the exec() system calls would be represented in Figure 7.1, where the dotted line mark the execution and transfer of control whereas the straight lines mark the waiting time.

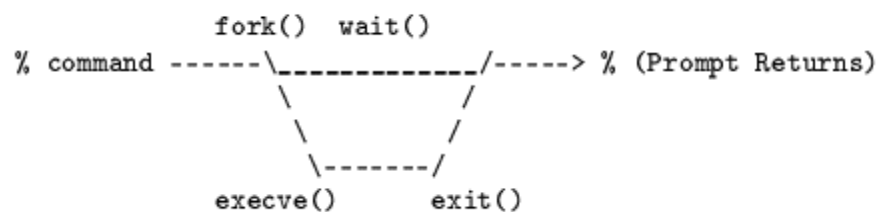


Figure 7.1: Fork() Exec() Representation

exec family of functions in C

The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program. It comes under the header file **unistd.h**. There are many members in the exec family which are shown below with examples.

1. **execvp** Using this command, the created child process does not have to run the same program as the parent process does. The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script. **Syntax:**

```
int execvp (const char *file, char *const argv[]);
```

file: points to the file name associated with the file being executed.

argv: is a null terminated array of character pointers.

Let us see a small example to show how to use execvp() function in C. We will have two .C files , **EXEC.c** and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling execvp() function in execDemo.c .

//EXEC1.c

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

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
    int i;
```

```
    printf("I am EXEC1.c called by execvp() ");
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

Now,create an executable file of EXEC1.c using command

```
gcc EXEC1.c -o EXEC1
```

//execDemo.c

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

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

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
    //A null terminated array of character pointers
```

```
    char *args[]={"/EXEC1",NULL};
```

```

    execvp(args[0],args);
    /*All statements are ignored after execvp() call as this whole
    process(execDemo.c) is replaced by another process (EXEC1.c)
    */
    printf("Ending-----");
    return 0;
}

```

Now, create an executable file of execDemo.c using command

```
gcc execDemo.c -o execDemo
```

After running the executable file of execDemo.c by using command ./execDemo, we get the following output:

```
I AM EXEC1.c called by execvp()
```

When the file execDemo.c is compiled, as soon as the statement execvp(args[0],args) is executed, this very program is replaced by the program EXEC1.c. “Ending——” is not printed because as soon as the execvp() function is called, this program is replaced by the program EXEC1.c.

2. **execv** This is very similar to execvp() function in terms of syntax as well. The syntax of **execv()** is as shown below:

Syntax:

```
int execv(const char *path, char *const argv[]);
```

path: should point to the path of the file being executed.

argv[]: is a null terminated array of character pointers.

Let us see a small example to show how to use execv() function in C. This example is similar to the example shown above for execvp() . We will have two .C files , **EXEC.c** and **execDemo.c** and we will replace the execDemo.c with EXEC.c by calling execv() function in execDemo.c .

//EXEC2.c

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

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
int i;
```

```
    printf("I am EXEC2.c called by execv() ");
```

```
    printf("\n");
```

```
    return 0;
```

```
}
```

Now,create an executable file of EXEC2.c using command

```
gcc EXEC2.c -o EXEC2
```

//execDemo1.c

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

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

```
#include<unistd.h>
```

```
int main()
```

```
{
```

```
    //A null terminated array of character pointers
```

```
    char *args[]={"/EXEC2",NULL};
```

```
    execv(args[0],args);
```

```
    /*All statements are ignored after execv() call as this whole  
    process(execDemo1.c) is replaced by another process (EXEC2.c)
```

```
    */
```

```
    printf("Ending-----");
```

```
    return 0;
```

```
}
```

Now, create an executable file of execDemo1.c using command

```
gcc execDemo1.c -o execDemo1
```

After running the executable file of execDemo1.c by using command ./execDemo, we get the following output:

```
I AM EXEC.c called by execv()
```

Example: 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(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

hello.c

CODE:

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

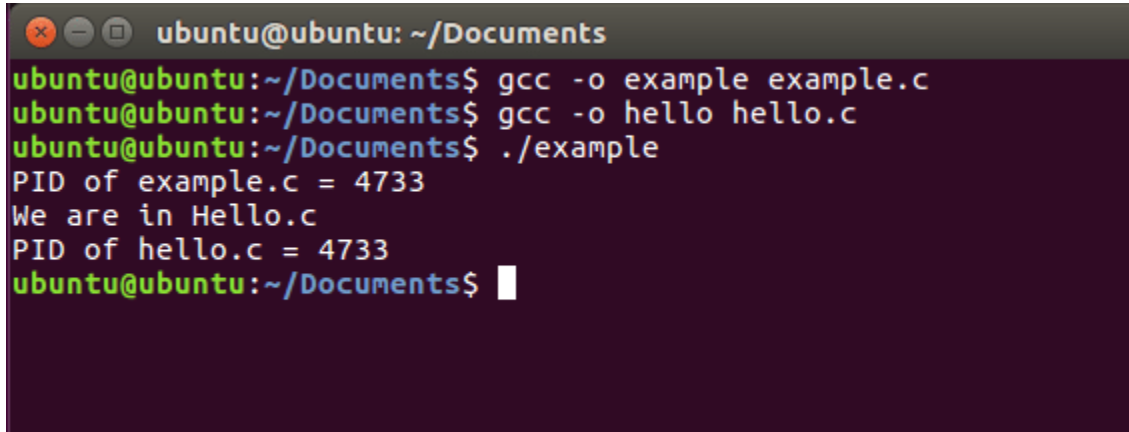
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

OUTPUT:

PID of example.c = 4733

We are in Hello.c

PID of hello.c = 4733

A terminal window titled 'ubuntu@ubuntu: ~/Documents' with a dark purple background. It shows the following commands and output:

```
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```

One thing that we should 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:

Here we will see the effect of `fork()` and `exec()` system call in C. The fork is used to create a new process by duplicating the calling process. The new process is the child process. See the following property.

- The child process has its own unique process id.
- The parent process id of the child process is same as the process id of the calling process.
- The child process does not inherit the parent's memory lock and semaphores.

The `fork()` returns the PID of the child process. If the value is non-zero, then it is parent process's id, and if this is 0, then this is child process's id.

The `exec()` system call is used to replace the current process image with the new process image. It loads the program into the current space, and runs it from the entry point.

So the main difference between `fork()` and `exec()` is that `fork` starts new process which is a copy of the main process. the `exec()` replaces the current process image with new one, Both parent and child processes are executed simultaneously.

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(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    pid_t p;
    p = fork();
    if(p== -1)
```

```
{  
    printf("There is an error while calling fork()");  
}  
if(p==0)  
{  
    printf("We are in the child process\n");  
    printf("Calling hello.c from child process\n");  
    char *args[] = {"Hello", "C", "Programming", NULL};  
    execv("./hello", args);  
}  
else  
{  
    printf("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("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

OUTPUT:

PID of example.c = 4790

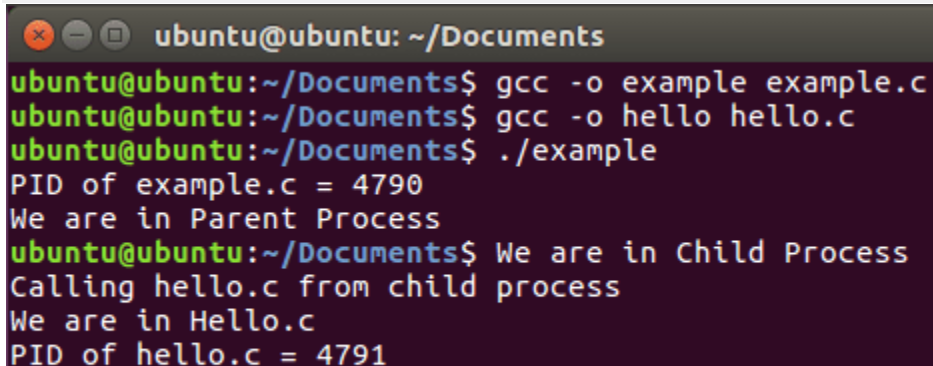
We are in Parent Process

We are in Child Process

Calling hello.c from child process

We are in hello.c

PID of hello.c = 4791

A terminal window with a dark background and light-colored text. The window title is 'ubuntu@ubuntu: ~/Documents'. The terminal shows the following commands and output:

```
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4790
We are in Parent Process
ubuntu@ubuntu:~/Documents$ We are in Child Process
Calling hello.c from child process
We are in Hello.c
PID of hello.c = 4791
```

References

<https://linuxhint.com/linux-exec-system-call/>

<https://www.geeksforgeeks.org/exec-family-of-functions-in-c/>