FAST NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, PESHAWAR

DEPARTMENT OF COMPUTER SCIENCE

CL220 - OPERATING SYSTEMS LAB

LAB MANUAL # 07

INSTRUCTOR: MUHAMMAD ABDULLAH

SEMESTER SPRING 2021

Contents

Operating Systems Processes

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 execle(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 it's contents?

Question 2: What is the 2nd argument to it? What is it's 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.

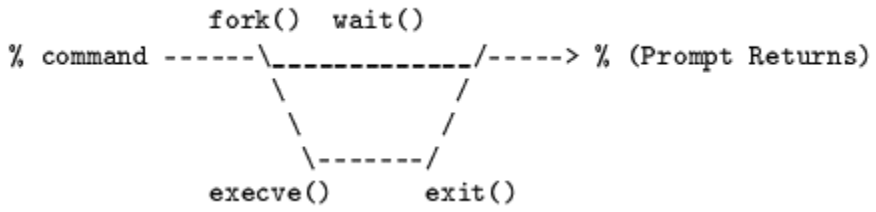


Figure 7.1: Fork() Exec() Representation

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.cby 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()
```

Process Waiting States

sleep() system call

The sleep() call can be used to cause delay in execution of a program. The delay can be provided as an integer number (representing number of seconds). Usage is simply sleep(int), which will delay the process execution for int seconds. To use sleep(), you will require the ***unistd.h*** C library.

Exercise

Write a C program that can display a count from 10 to 0 (reverse order) using a for or a while loop. Each number should be displayed after delay of 1 second.

sleepExercise.c

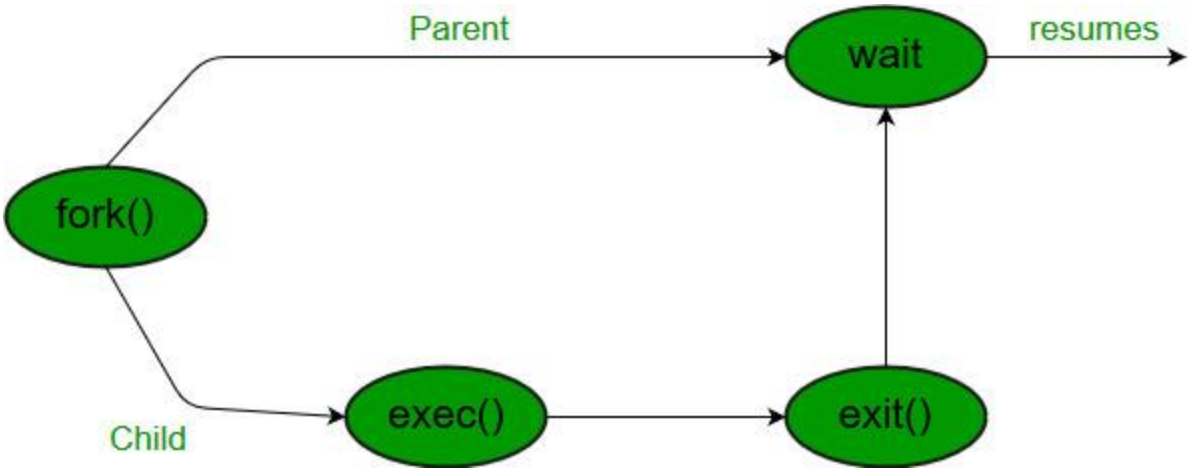
```
#include <unistd.h>
#include <stdio.h>
int main()
{
for(int i=10; i>=0; i--)
{
        printf("%d\n",i);
        sleep(2);
}
}
```

wait() system call

Suspend execution until a child process exits. Wait returns the exit status of that child.

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent ***continues*** its execution after wait system call instruction.
Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.



Syntax in c language:

```
#include
#include
// take one argument status and returns
// a process ID of dead children.
pid_t wait(int *stat_loc);
```

- If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.
- If only one child process is terminated, then return a wait() returns process ID of the terminated child process.
- If more than one child processes are terminated than wait() reap any ***arbitrarily child*** and return a process ID of that child process.
- When wait() returns they also define **exit status** (which tells our, a process why terminated) via pointer, If status are not **NULL**.
- If any process has no child process then wait() returns immediately "-1". ***NOTE: "This codes does not run in simple IDE because of environmental problem so use terminal for run the code"*** .

Examples:

wait1.c

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
   pid_t cpid;
   if (fork()== 0)
        exit(0);                    /* terminate child */
   else
        cpid = wait(NULL);       /* reaping parent */
   printf("Parent pid = %d\n", getppid());
   printf("Child pid = %d\n", cpid);
   return 0;
}
```

Output:

```
Parent pid = 5150
Child pid = 6046
```

wait2.c

```
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
        if (fork()== 0)
                printf("HC: hello from child\n");
        else
        {
                printf("HP: hello from parent\n");
                wait(NULL);
                printf("CT: child has terminated\n");
        }

        printf("Bye\n");
        return 0;
}
```

Output: depend on environment

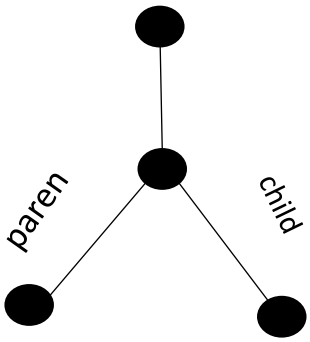
```
HC: hello from child
Bye
HP: hello from parent
CT: child has terminated
     (or)
HP: hello from parent
HC: hello from child
HC: Bye
CT: child has terminated    // this sentence does
Bye                         //not print before HC because of wait.
```

wait3.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<sys/wait.h>
int main()
{
int pid;
pid = fork();
if(pid==0)
{
        printf("Child is running...\n");
}
else
{
        printf("Parent is running...\n");
        wait(NULL);
        printf("Child terminated\n");
}
printf("Done\n");
return 0;
}
```

Output: depend on environment

```
Parent is running...
Child is running...
Done
Child terminated
Done
```



Parent running

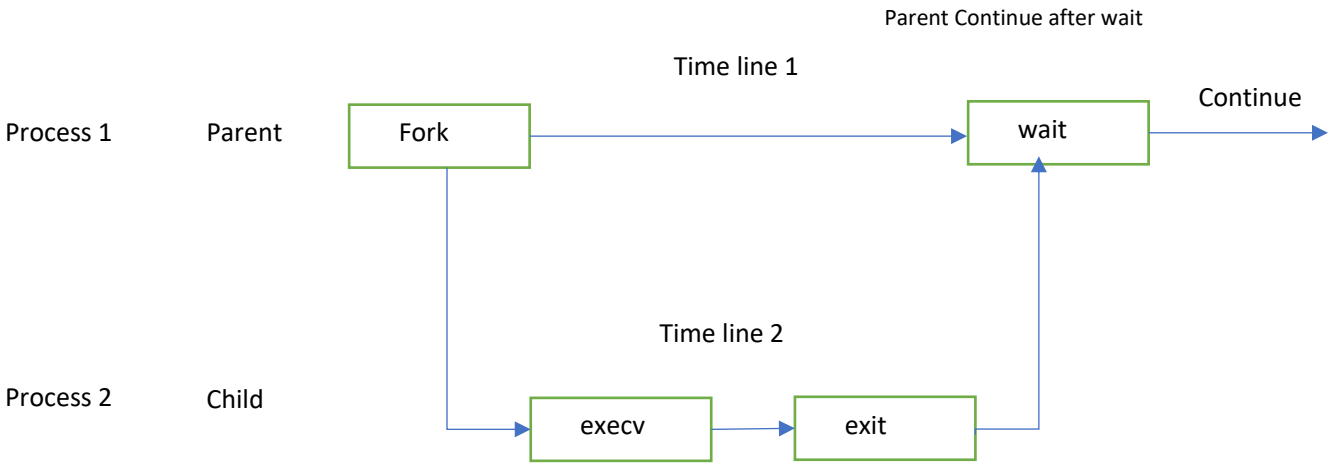
Child running
Done

Child terminated
Done

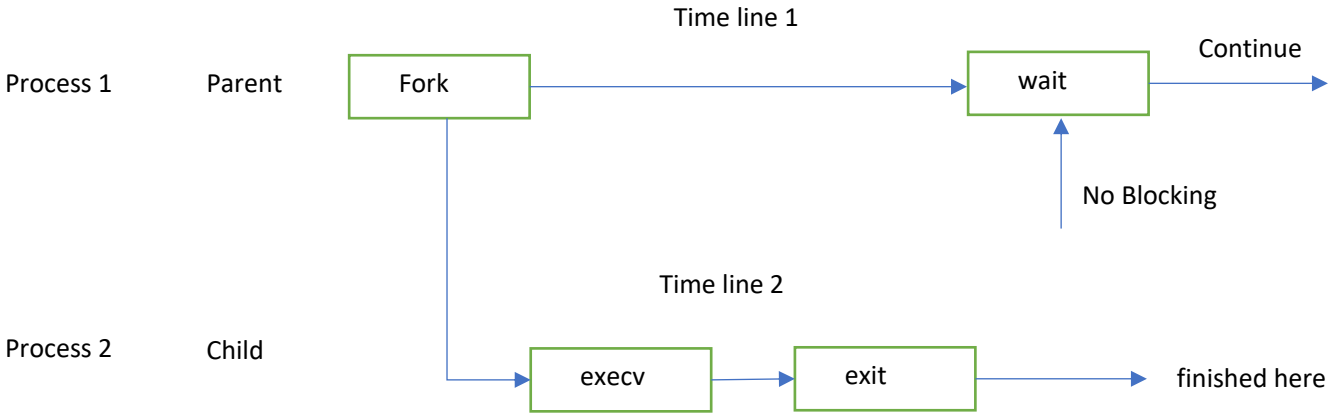
Terminal Inputs
fork
parent: -----> wait state
child ---->exec() --->exit
parent: ---

Parent and Child Processes

Case # 01: If parent process executes first



Case # 02: If child process executes first



References

https://www.geeksforgeeks.org/fork-system-call/?ref=leftbar-rightbar

https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html