

# Computer Operating Systems

## Practice Session 1: fork and exec System Calls In Linux

Esin Ece AYDIN (aydinesi16@itu.edu.tr)  
Ömür Fatmanur Erzurumluğlu (erzurumluoglu18@itu.edu.tr)  
Uğur Ayvaz(ayvaz18@itu.edu.tr)

March 5, 2024

## ① Operating Systems, PS 1

- fork System Call
- exec System Call

```
$ man fork
```

FORK(2) Linux Programmer's Manual FORK(2)

## NAME

fork - create a child process

## SYNOPSIS

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

pid_t fork(void);
```

## DESCRIPTION

`fork()` creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

The child process and the parent process run in separate memory spaces. At the time of `fork()` both memory spaces have the same content. Memory writes, file mappings (`mmap(2)`), and unmappings (`munmap(2)`) performed by one of the processes do not affect the other.

The child process is an exact duplicate of the parent process except for the following points:

- \* The child has its own unique process ID, and this PID does not match the ID of any existing process group (`setpgid(2)`) or session.

- The `fork()` call creates an exact copy of the running process and returns two separate values:
  - Once for *parent process*
  - Once for *child process*
- How can we separate parent and child processes?
  - Return value in child process = 0
  - Return value in parent process = process ID of the child process
- After the call, the parent and child processes perform their specified functions.

# Example Program

```
1 #include <stdio.h>      // printf
2 #include <unistd.h>     // fork
3 #include <stdlib.h>     // exit
4 #include <sys/wait.h>   // wait
5
6 int main(void){
7     int f = fork();      // forking a child process
8     if(f == -1){         // fork is not successful
9         printf("Error\n");
10        exit(1);
11    }
12    else if (f == 0){     // child process
13        printf("    Child: Process ID: %d \n",getpid());
14        // waiting for 10 seconds
15        sleep(10);
16        printf("    Child: Parent Process ID: %d \n",getppid());
17    }
18    else{                // parent process
19        printf("Parent: Process ID: %d \n", getpid());
20        printf("Parent: Child Process ID: %d \n", f);
21        printf("Parent: Parent Process ID: %d \n", getppid());
22        // waiting until child process has exited
23        wait(NULL);
24        printf("Parent: Terminating... \n");
25        exit(0);
26    }
27    return 0;
28 }
```

# Output of the Example Program

```
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folders$ gcc forkExample.  
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folders$ ./a.out  
Parent: Process ID: 2266  
Parent: Child Process ID: 2267  
Parent: Parent Process ID: 1943  
Child: Process ID: 2267  
Child: Parent Process ID: 2266  
Parent: Terminating...
```

```
$ man exec
```

```
EXEC(3)                                Linux Programmer's Manual                                EXEC(3)
```

## NAME

execl, execlp, execl, execv, execvp, execvpe - execute a file

## SYNOPSIS

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

```
extern char **environ;
```

```
int execl(const char *pathname, const char *arg, ...
           /* (char *) NULL */);
int execlp(const char *file, const char *arg, ...
           /* (char *) NULL */);
int execl(const char *pathname, const char *arg, ...
           /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```

Feature Test Macro Requirements for glibc (see `feature_test_macros(7)`):

```
execvpe(): _GNU_SOURCE
```

The exec system call consists of several functions.

`execl`, `execv`, `execle`, `execve`, `execlp`, `execvp`

- `exec()` system calls execute a command or group of commands (file, program)
- only way to run a program within a process in Unix
- The process ID (PID) of the current process is not changed, this is because we are not creating a new process, we are simply replacing one process with another by calling `exec()`.
- `exec()` is used to start a new file or program in the same process.
- All `exec()` collection of functions provides the same functionality, but the functions prototypes vary according to how the parameters are sent.



There are 6 ways for calling exec function:

- `int execl(const char *path, const char *arg0, ...);`
- `int execlp(const char *path, const char *arg0, ..., char *const envp[]);`
- `int execlp(const char *file, const char *arg0, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execve(const char *file, char *const argv[], ..., char *const envp[]);`
- `int execvp(const char *file, char *const argv[]);`

- **l** specifies that the argument pointers ( $arg_0, arg_1, \dots, arg_n$ ) are passed as separate arguments. Typically, the **l** suffix is used when you know in advance the number of arguments to be passed.
- **v** specifies that the argument pointers ( $argv[0] \dots, argv[n]$ ) are passed as an array of pointers. Typically, the **v** suffix is used when a variable number of arguments is to be passed.
- **p** specifies that the function searches for the file in those directories specified by the PATH environment variable (without the **p**, the function searches only the current working directory). If the path parameter does not contain an explicit directory, the function searches first the current directory, then the directories set with the PATH environment variable.
- **e** allows the caller to specify the environment of the executed program via the argument **envp**. The **envp** argument is an array of pointers to null-terminated strings (*name = value*) and must be terminated by a null pointer. Without the **e** suffix, child processes inherit the environment of the parent process.

- The exec group calls are basically a way to replace the entire existing process with a new program. It loads the program into the available process space and runs it. exec replaces the current process with the executable file that the function points to.
- Processes' exit value is collected by parent process. By using this property control can return to original program if exec() receives an error.

- **errno** can be checked:

ENAMETOOLONG	: The length of path or file exceeds <i>PATH_MAX</i> , or a path name is longer than <i>NAME_MAX</i>
EACCES	: Permission denied
E2BIG	: Argument list and environment list is greater than <i>ARG_MAX</i>
ENOENT	: Path or file name not found
ENOEXEC	: Has the appropriate access permissions, but is not in the proper format
ENOMEM	: Not enough memory
ENOTDIR	: A component of the new process image file's path prefix is not a directory

## Example Program (Master)

```
1 #include <stdio.h>           // printf
2 #include <unistd.h>          // fork, execlp
3 #include <stdlib.h>          // exit
4 #include <errno.h>           // errno
5 #include <string.h>          // strerror
6
7 int main(void){
8     printf("\n Master is working: PID:%d \n",getpid());
9
10    int f = fork();           // forking a child process
11
12    if (f == 0) {              // child process
13        printf("\n This is child... PID: %d \n", getpid());
14        // execute the slave process
15        execlp("./execSlave", "./execSlave"
16              , "test1", "test2", (char*)NULL);
17        // exec returns only when there is an error
18        printf("\n %s", strerror(errno));
19    }
20    else{                       // parent process
21        exit(0);
22    }
23    return 0;
24 }
```

## Example Program (Slave)

```
1  #include <stdio.h>      // printf
2  #include <unistd.h>     // getpid
3
4  int main(int argc, char* argv[])
5  {
6      printf("\nSlave started working ... PID: %d \n",getpid());
7      printf("Name of the Program :%s \n",argv[0]);
8      printf("The first parameter of the program:%s \n",argv[1]);
9      printf("The second parameter of the program:%s \n",argv[2]);
10     return 0;
11 }
```

# Output of the Example Program

```
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folder$ gcc execSlave.c  
-o execSlave  
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folder$ gcc execMaster.c  
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folder$ ls  
a.out  execMaster.c  execSlave  execSlave.c  forkExample.c  
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folder$ ./a.out  
  
Master is working: PID:2314  
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folder$  
This is child... PID: 2315  
  
Slave started working ... PID: 2315  
Name of the Program :./execSlave  
The first parameter of the program:test1  
The second parameter of the program:test2
```

## Output of the Example Program (Failure)

```
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folders$ rm execSlave
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folders$ ./a.out

Master is working: PID:2324
musty@musty-VirtualBox:/media/sf_virtualbox_shared_folders$
This is child... PID: 2325

No such file or directory
```