Experiment No 2

By Harshali Bhuwad

fork() Syntax and Return Value:

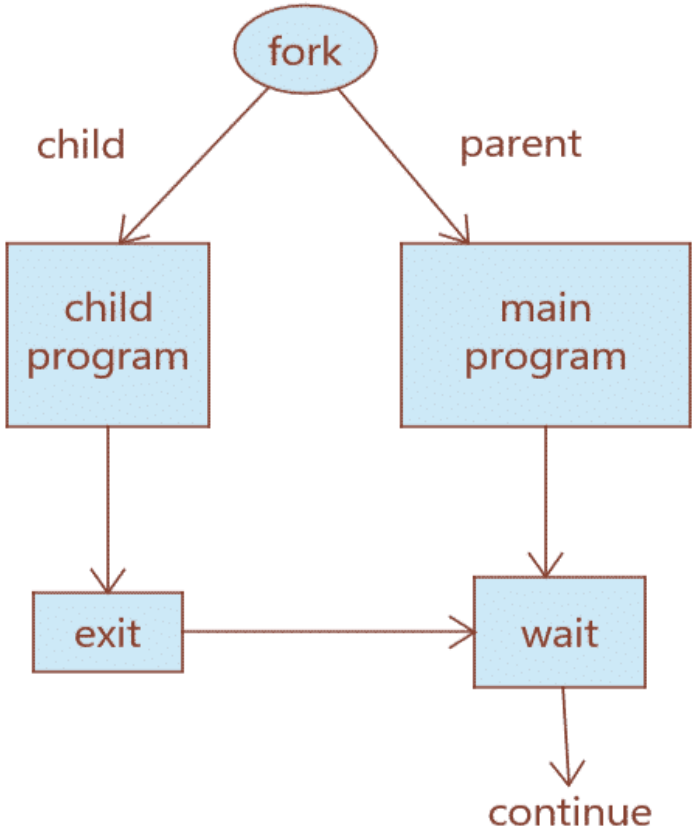
- fork() system call is used to create child processes in a C program.

```
pid_t fork(void);
```

- The fork() system function does not accept any argument. It returns an integer of the type **pid_t**.

- On success, fork() returns the PID of the child process which is greater than 0. Inside the child process, the return value is 0. If fork() fails, then it returns -1.

Example 1

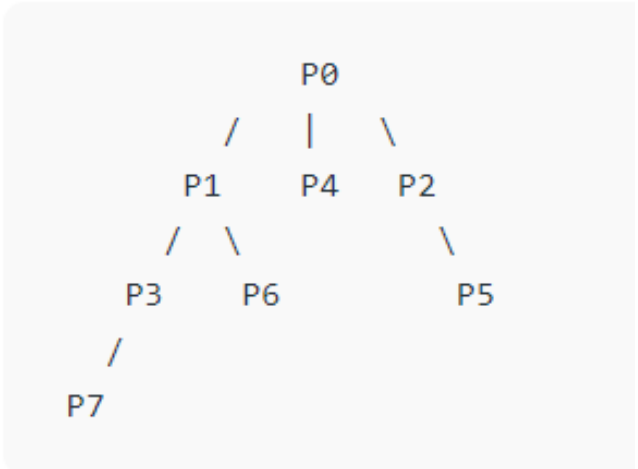



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

        // make two process which run same
        // program after this instruction

        fork();

        printf("Hello world!\n");
        return 0;
}
```

The number of times ‘hello’ is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls.

Example 2

```
fork ();   // Line 1
fork ();   // Line 2
fork ();   // Line 3

       L1       // There will be 1 child process
    /     \     // created by line 1.
  L2      L2    // There will be 2 child processes
 /  \    /  \   //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```



The main process: P0
Processes created by the 1st fork: P1
Processes created by the 2nd fork: P2, P3
Processes created by the 3rd fork: P4, P5, P6, P7

So there are total eight processes (new child processes and one original process).

Example 3

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
        // child process because return value zero
        if (fork() == 0)
                printf("Hello from Child!\n");

        // parent process because return value non-zero.
        else
                printf("Hello from Parent!\n");
}
int main()
{
        forkexample();
        return 0;
}
```

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.

- 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().
- 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.

Inner 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.

Example 1: Using exec system call in C program

example.c

CODE:

```c
#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:

```c
#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;
}
```