

## LAB 3

### Objectives:

- Understand a process
- How to create a process
- How to identify a process
- How to create a child process
- Understanding process communication
- Understanding the system calls used for all above objectives
  - fork() system call
  - pipe() system call
  - execlp() / execvp() / execve() system call

### Requirements:

- Linux
- GCC/G++
  - To install C++ run the following command
  - `sudo apt-get install gcc`
  - `sudo apt-get install g++`

## 1. Fork() System Call

<http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html>

<https://man7.org/linux/man-pages/man2/fork.2.html>

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

- If fork() returns a negative value, the creation of a child process was unsuccessful.
- fork() returns a zero to the newly created child process.
- fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid\_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

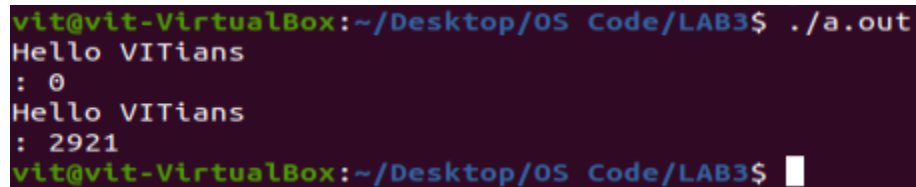
Therefore, after the system call to fork(), a simple test can tell which process is the child. Please note that Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Example of fork system call in C.

**Example CODE 1:**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(void)
{
    pid_t pid = fork();
    int exit;
    if (pid != 0)
    {
        wait(&exit);
    }
    printf("Hello VITians \n: %d\n", pid);
}
```

**Output:**



```
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ ./a.out
Hello VITians
: 0
Hello VITians
: 2921
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$
```

0--> refers to the child process

2921 (random PID)--> refers to the parent process

**Example code2:**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t forkStatus;

    forkStatus = fork();
```

```

/* Child... */
if (forkStatus == 0) {
    printf("Child is running, processing.\n");
    sleep(5);
    printf("Child is done, exiting.\n");
}

/* Parent... */
else if (forkStatus != -1) {
    printf("Parent is waiting...\n");
    wait(NULL);
    printf("Parent is exiting...\n");
}

/* Unable to create child */
else {
    perror("Error while calling the fork function");
}
return 0;
}

```

### Output:

```

Parent is exiting...
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ ./a.out
Parent is waiting...
Child is running, processing.

```

```

vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ ./a.out
Parent is waiting...
Child is running, processing.
Child is done, exiting.
Parent is exiting...
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$

```

## 2. Pipe() System Call

Check manual

Man pipe

```

PIPE(7)                                Linux Programmer's Manual                                PIPE(7)

NAME
    pipe - overview of pipes and FIFOs

DESCRIPTION
    Pipes and FIFOs (also known as named pipes) provide a
    unidirectional interprocess communication channel. A
    pipe has a read end and a write end. Data written to
    the write end of a pipe can be read from the read end
    of the pipe.

```

It creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe.

- **`pipefd[0]` refers to the read end** of the pipe.
- **`pipefd[1]` refers to the write end** of the pipe.

Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.



```
/*-----Pipe Example-----*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h> //header file to provide access to POSIX OS API

int main()
{
    int status, pid, pip[2];
    char instr[20];
    char outstring[20];
    strcpy(outstring, "CSE2005\n");

    //creating pipe
    status=pipe(pip);
    if(status == -1)
    {
        perror("Unable to create pipe");
        exit(1);
    }

    //Write part
    printf("Sending Message . . . \n");
    write(pip[1], outstring, strlen(outstring)+1);
    close(pip[1]); // close the write part

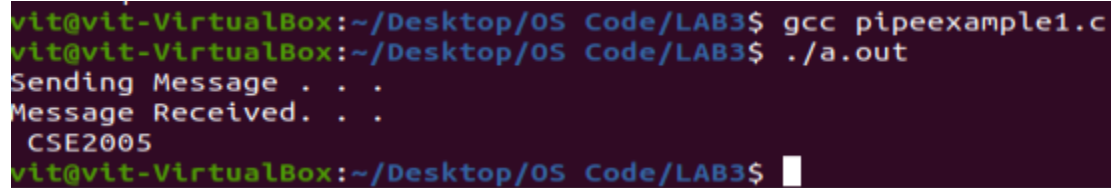
    //Read part
```

```

        read(pip[0], instring, strlen(outstring)+1);
        printf("Message Received. . .\n %s", instring);
        close(pip[0]); // close the read part
    }

```

#### Output:



```

vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ gcc pipeexample1.c
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ ./a.out
Sending Message . . .
Message Received. . .
CSE2005
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$

```

### 3. Communication of two processes using pipe() and fork()

/\*Here we demonstrate an example of process communication in which child communicates to parent using pipe\*/

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
using namespace std;
#include<unistd.h> //header file to provide access to POSIX OS API

int main()
{
    int status, pid, pip[2];
    char instring[20];
    char outstring[20];
    strcpy(outstring, "Hi I am child");

    //creating pipe
    status=pipe(pip);

    if(status == -1){
        perror("Unable to create pipe");
        exit(1);
    }

    pid = fork(); //create child process
    if(pid == -1) {
        perror("Unable to create process");
        exit(2);
    }
}

```

```

    }
    else if(pid == 0) {    // child process: send message to parents
        close(pip[0]);    //close the read part of pipe
        printf("Sending Message . . . \n");
        write(pip[1],outstring, strlen(outstring)+1);
        close(pip[1]);    // writing done, close it
        exit(0);
    }
    else {                // parent process: read message from child
        close(pip[1]);    //close the write part of pip
        read(pip[0], instring, strlen(outstring)+1);
        printf("Message Received. . .\n %s\n",instring);
        close(pip[0]);    // Read done, close it
        exit(0);
    }
}

```

Output:

```

vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ gcc pipe-process-communication-exampl
e1.c
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ ./a.out
Sending Message . . .
Message Received. . .
Hi I am child
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$

```

Questions:

- 1) Create four identical processes and print their order of execution. While printing make sure that the processes are uniquely identified.
- 2) Create 3 processes A, B, and C.
  - a) Process A will generate a random number X (>0 and <100) and will forward X to process B through pipes.

Example: Say a random number generated is 5. So 5 is send

- b) Process B will read X through pipes and generate fibonacci series upto X term. Now Process B will forward the number in term  $X/2$  to Process C.

Example: Fib = 1 1 2 3 5

Here  $X/2$  term is  $5/2$  i.e. 2 will be forwarded to Process C.

- c) Process C will read the number forwarded by process B and will determine if the number is odd or even.

Example: 2 is even.

## 4. exec family of functions in C

### 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 execlx(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.

1. **path** is used to specify the full path name of the file which is to be executed.
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()`, `execvp()` 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.

#### **first.c**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of first process = %d\n", getpid());
    char *args[] = {"Hello", "Hi", "System calls", NULL};
    execv("./second", args);
    printf("Back to example.c");
    return 0;
}
```

#### **second.c**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in second.c\n");
    printf("PID of second.c = %d\n", getpid());
    return 0;
}
```

**Output:**



```
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ gcc -o second second.c
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ gcc -o first first.c
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$ ./first
PID of first process = 4110
We are in second.c
PID of second.c = 4110
vit@vit-VirtualBox:~/Desktop/OS Code/LAB3$
```

**Question:**

- 3) Write a program in C/C++ under the Linux environment that would perform the following:
  - a) In a loop, read a character string containing the name of an executable program with command-line arguments, if any.
  - b) Fork a child process, and execute the program.
  - c) The loop will terminate if the command "quit" is entered.