Operating System

Lab No 4: Process Creation and Execution (fork() & exec())

CLO: 2

Rubrics for Lab:

Task	0	1	2	3
Understanding the Fork() and exec system call	**Students don't know how Fork() and exec works**	**Students have knowledge of Fork() and exec() but couldn't answer the question given in the tasks**	**Student partial answered the question given in task**	**Students have complete knowledge of Fork() Commands and answer the question properly**
As mentioned in task-2	**Student has no knowledge of Command line argument.**	**Student has little knowledge of Command line argument and is not able to write full C code.**	**Students perform the task partially**	**Student has complete knowledge of Command line argument and performed the full task**

Topic to be covered

Process Creation and execution

1. fork()
    a. What is fork ()?
    b. How fork () work?
    c. Sample Program (Dry Run as well)
    d. Graded Task
2. exec()
    a. What is exec ()?
    b. Sample Program (Dry Run as well)
3. Command Line Argument
    a. Parsing command line arguments
    b. Sample program
    c. Graded Task

Objectives

After this lab students will able of creating and execute its own process in operating system.
They will able to deal with command line arguments in operating system.
They will able to deal will Xtern terminal.

What is fork ()?

System call **fork ()** is used to create processes. It takes no arguments and returns a process ID.

How fork () works?

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.

Sample Program

Processes are created with the fork () system call (so the operation of creating a new process is sometimes called forking a process). The child process created by fork is a copy of the original parent process, except that it has its own process ID.

1. To display information about currently running processes us the ps command:

ps

2. pstree is a small, command line program that displays the processes (i.e., executing instances of programs) on the system in the form of a tree diagram.

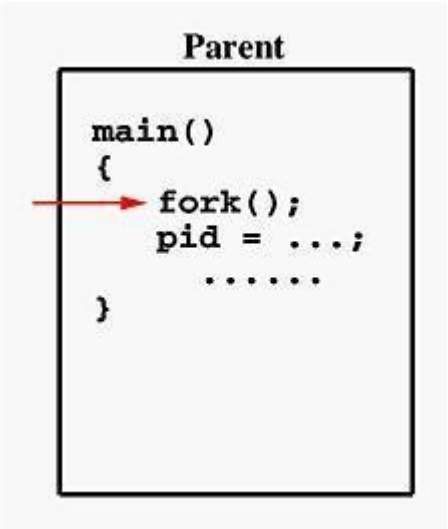
pstree

Example 1:

Let us take an example to make the above points clear.

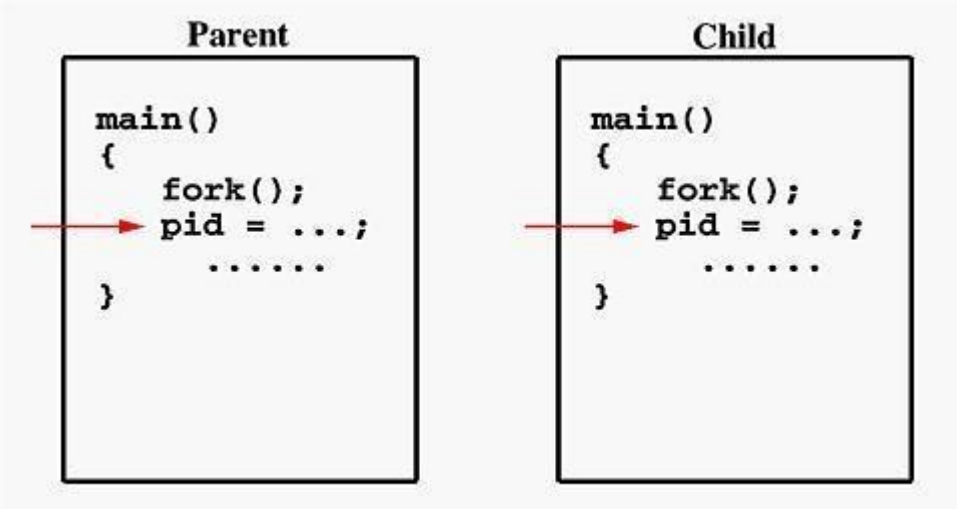
```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
     pid_t  pid;
     int    i;
     char   buf[BUF_SIZE];

     fork();
     pid = getpid();
     for (i = 1; i <= MAX_COUNT; i++) {
          sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
          write(1, buf, strlen(buf));
     }
}
```

Suppose the above program executes up to the point of the call to **fork ()**.



If the call to **fork ()** is executed successfully, UNIX will

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork ()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork ()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork ()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork ()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf ()** is "buffered," meaning **printf ()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.

If you run this program, you might see the following on the screen:

```
.....
This line is from PID 3456, value 13
This line is from PID 3456, value 14
.....
This line is from PID 3456, value 20
This line is from PID 4617, value 100
This line is from PID 4617, value 101
.....
```

This line is from PID 3456, value 21
This line is from PID 3456, value 22
.....

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result

Exercise 1 (Dry Run):

Task1:

```
#include <stdio.h> //standard c library for input output
#include <unistd.h> /* contains fork prototype */
int main()
{
int pid;
printf("Hello World!\n");
printf("I am the parent process and pid is : %d.\n",getpid());
printf("Here i am before use of forking\n");
pid = fork();   //new process is created
printf("Here I am just after forking\n");   //Child process will start execution from this line
if (pid == 0)
printf("I am the child process and pid is :%d.\n",getpid());
else
printf("I am the parent process and pid is: %d.\n",getpid());
}
```

Q1: Dry Run the above program and explain the output?

Task1 Graded (On Paper):

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main()
{
        printf("Here I am just before first forking statement\n");
        fork();
        printf("Here I am just after first forking statement\n");
        fork();
        printf("Here I am just after second forking statement\n");
        printf("\t\tHello World from process %d!\n", getpid());
        return 0;
}
```

Q2: Tell how many processes are created in above source code and draw a process tree for given source code and dry run it.

Exec:

What is exec?

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell.This is where the exec system call comes into play exec will replace the contents of the currently running process with the information from a program binary. The following code replaces the child process with the binary created for hello.c

Sample Program

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
}
```

Step 1: Create a file “hello.c” and type following source code

Step 2: Compile and make a executable file named hello.out
Step3: Create another file “parent.c”. Type following code.

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
    int pid;
    printf("I am the parent process and pid is : %d.\n",getpid());
    printf("Here i am before use of forking\n");
    pid = fork();   //new process is created
    printf("Here I am just after forking\n");   //Child process will start execution from this line
    if (pid == 0)
    printf("I am the child process and pid is :%d.\n",getpid());
    printf("I am loading „hello“ process\n");
    execv("hello.out","hello.out",NULL);
    else
    printf("I am the parent process and pid is: %d.\n",getpid());
}
```

Exercise 2:

Q1: Compile the above program? Do you see any error? If Yes! Explain the error in code?
Q2: Resolve the error in the code and handover the new file.

Parsing command line arguments

Introduction

Command-line parameters are passed to a program at runt-time by the operating system when the program is requested by another program, such as a command interpreter ("shell") like cmd.exe on Windows or bash on Linux and OS X. The user types a command and the shell calls the operating system to run the program. Exactly how this is done is beyond the scope of this article (on Windows, look up Create Process; on UNIX

and UNIX-like

systems look up fork(3) and exec(3) in the manual).

The uses for command-line parameters are various, but the main two are:

1. Modifying program behavior - command-line parameters can be used to tell a program how you expect it to behave; for example, some programs have a -q (quiet) option to tell them not to output as much text.

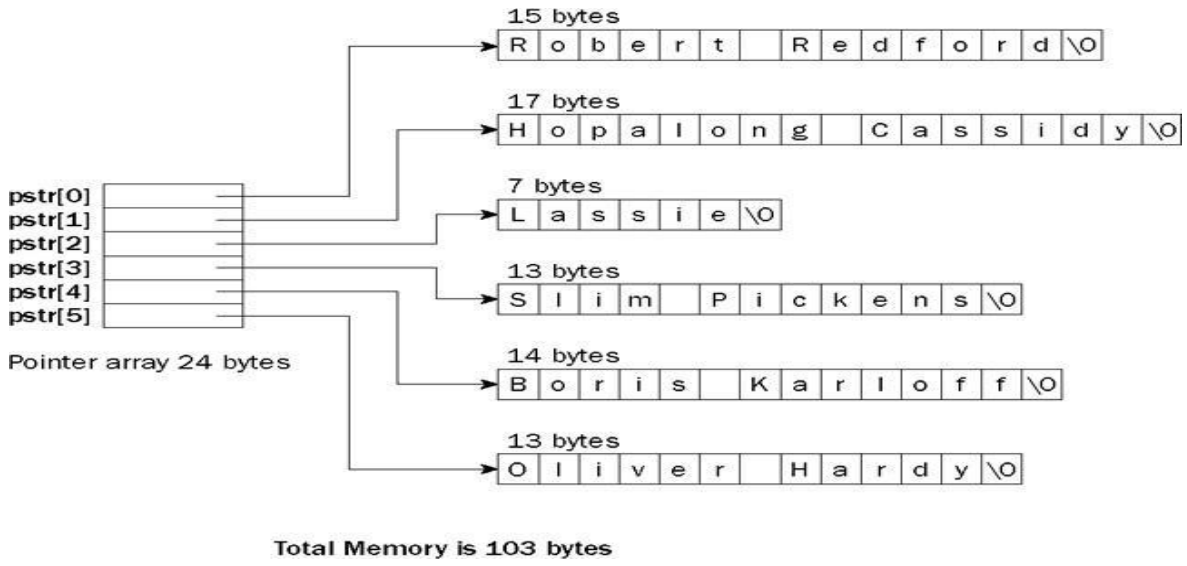
2. Having a program run without user interaction - this is especially useful for programs that are called from scripts or other programs.

The command-line

Adding the ability to parse command-line parameters to a program is very easy. Every C and C++ program has a main function. In a program without the capability to parse its command- line, main is usually defined like this:

int main()

To see the command-line we must add two parameters to main which are, by convention, named argc (argument count) and argv (argument vector [here, vector refers to an array, not a C++ or Euclidean vector]). argc has the type int and argv usually has the type char** or char* [](see below). main now looks like this:



Int main (int argc , char * argv[]) // char ** argv

argc tells you how many command-line arguments there were. It is always at least 1,

because the first string in argv (argv[0]) is the command used to invoke the program. argv contains the actual command-line arguments as an array of strings, the first of which (as we have already discovered) is the program's name.

Earlier it was mentioned that argc contains the number of arguments passed to the program. This is useful as it can tell us when the user hasn't passed the correct number of arguments, and we can then inform the user of how to run our program (Error check).

Exercise 3:

Q1: Write a program that prints out all the parameters that are passed to it via command line (Hint: argc)

For instance *./program.out hello Pakistan*

Should print out put:

Number of parameters 2
Hello
Pakistan

Graded task 2

Write another program that takes two integers as command line parameters and prints the sum of the two parameters. (Hint: The parameters are treated as strings and not integers.)