



Lab 1 - Simple Shell Commands and fork Process

objectives

- Getting familiar with Linux shell.
 - Exploring different Linux commands and their different classifications.
 - Writing and compiling C using Linux terminal.
 - Using C language to make system calls.
 - Understanding process creation and spawns.
 - Implementing a simple shell in C.
-

What is a shell?

- A shell process is a command line interpreter that provides the user with an interface to the operating system.
- It can be used to run commands and programs.

Basic Linux shell commands

- **cd** -- change directory
- **pwd** -- print working directory
- **ls** -- list all files and sub-directories in a directory
- **stat** -- display information about a file
- **rm** -- remove (i.e. delete) a file
- **cp** -- copy a file

- **cat**, **more**, **less** -- list the contents of a file
- **chmod** -- change file mode, i.e. file permissions
- **ln** -- create a link
- **wc** -- count words
- **mkdir** -- create a new directory
- **head** -- output only the first lines of a file
- **tail** -- output only the last lines of a file
- **grep** -- find a word in one or more files
- **ps** -- process status (lists running processes, often run as **ps aux** for the most information)
- **cut** -- extract a column from a file
- **sort** -- sort a file alphabetically
- **uniq** -- remove adjacent duplicate lines
- **find** -- find files and directories and perform subsequent operations on them
- More commands and with extra examples:
 - <https://gist.github.com/riipandi/3097780>

Compiling .c files

```
#include<stdio.h>

int main()
{
    printf("\nA sample C program\n\n");
    return 0;
}
```

```
gcc sampleProgram.c -o sampleProgram
```



-o flag is used to determine the output file name

What is a process?

Simply a process is a program in execution.

What is a fork ?

fork()

- It's a system call defined in the headers `sys/types.h` and `unistd.h`
- The purpose of `fork()` is to create a new process, which becomes the child process of the caller.
- `fork()` prototype:

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

what is a pid?

Each process is identified by a unique id given the type `pid_t` which is just an integer under the hood.



Any process can use the function `getpid()` to retrieve its process ID, and use `getppid()` to retrieve the process ID assigned to its parent.

Back to the prototype

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

The function takes no arguments.

- On fail, it returns a negative value

- On success, `fork()` returns:
 - a zero to the newly created child process.
 - a positive value, which is the process ID of the child process, to the parent.

Using this fact, after the system call to `fork()`, a simple test can tell which process is the child.

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

int main() {

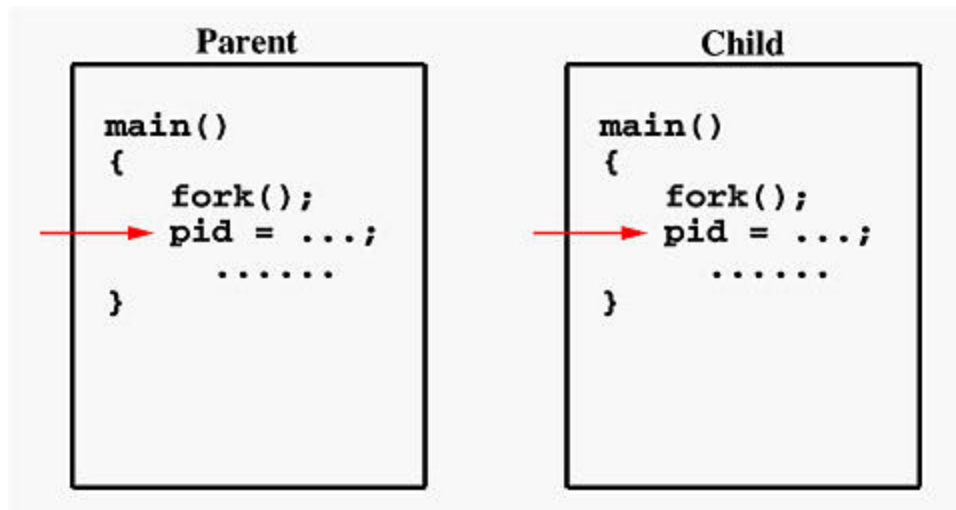
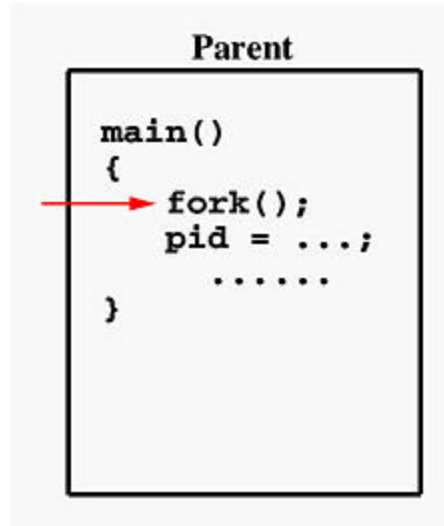
    /* fork a child process */
    pid_t pid = fork();

    if (pid < 0) { /* error occurred */
        printf("Fork failed Unable to create child process.\n");
        return 1;
    }

    else if (pid == 0) { /* child process */
        printf("I'm the Child => PPID: %d PID: %d\n", getppid(), getpid());
        exit(EXIT_SUCCESS);
    }

    else { /* parent process */
        printf("I'm the Parent => PID: %d\n", getpid());
        printf("Waiting for child process to finish.\n");

        /* parent will wait for the child to complete */
        wait(NULL);
        /* When the child is ended, then the parent will continue to execute its code */
        printf("Child Complete \n");
    }
}
```



Both processes will start their execution at the next statement following the `fork()` call.

What is the purpose of `wait()` ?

- A call to `wait()` blocks the calling process until one of its child processes exits or a signal is received.
- used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
- `wait()` prototype:

```
pid_t wait(int *wstatus);
```

It puts the process to sleep and waits for a child process to end. It then fills in the argument `wstatus` with the exit code of the child process.

Since we are not interested in why the child terminated and only need to block the parent until the child finishes execution, we shall pass NULL. 😊

what about `exit()` ?

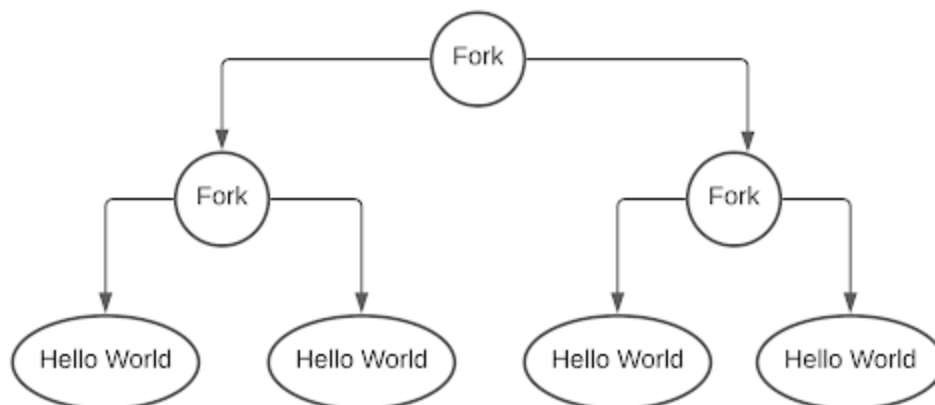
it terminates the calling process immediately.

Using multiple forks

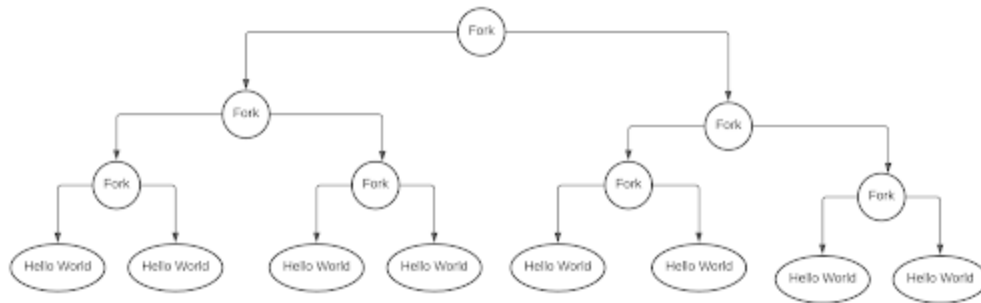
so far, we used only one `fork()` statement. Let's experiment with 2 statements and expect what the results should be

How many times will the statement "Hello World" be printed?

```
int main()
{
    fork();
    fork();
    printf("Hello world!\n");
    return 0;
}
```



```
int main()
{
    fork();
    fork();
    fork();
    printf("Hello world!\n");
    return 0;
}
```



Multiple forks result in 2^n processes

execv() command

The `execv` system call is used to execute a file which is residing in an active process. When `execv()` is called the previous executable file is replaced and new file is executed. we can say that using `execv` system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.

example for using exec system call in c program

example.c

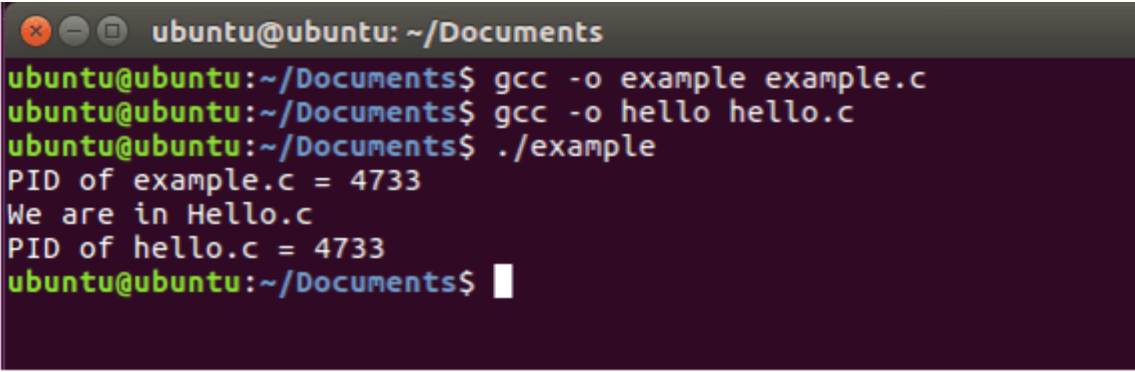
```
int main(int argc, char *argv[]){
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
}
```

```
    return 0;
}
```

hello.c

```
int main(int argc, char *argv[]){
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

output:



```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```

Difference between `fork()` and `execv()` system calls:

The `fork()` system call is used to create an exact copy of a running process and the created copy is the child process and the running process is the parent process.

Whereas, `execv()` system call is used to replace a process image with a new process image. Hence there is no concept of parent and child processes in `execv()` system call.

In `fork()` system call the parent and child processes are executed at the same time.

But in `execv()` system call, if the replacement of process image is successful, the control does not return to where the `exec` function was called rather it will execute the new process. The control will only be transferred back if there is any error.

Example for Combining `fork()` and `exec()` system calls

example.c


```

int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    pid_t p;
    p = fork();
    if(p==-1) {
        printf("There is an error while calling fork()");
    }
    if(p==0) {
        printf("We are in the child process\n");
        printf("Calling hello.c from child process\n");
        char *args[] = {"Hello", "C", "Programming", NULL};
        execv("./hello", args);
    }
    else {
        printf("We are in the parent process");
    }
    return 0;
}

```

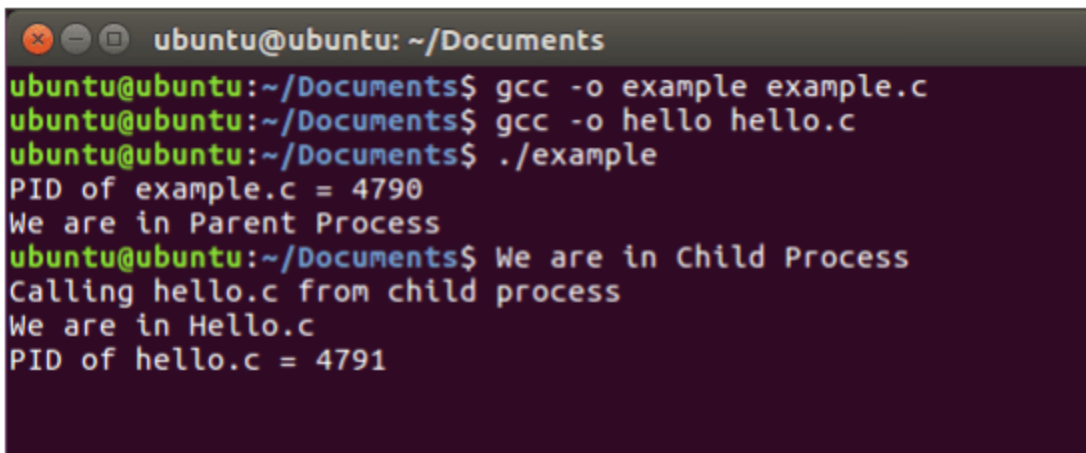
hello.c

```

int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}

```

output:



```

ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4790
We are in Parent Process
ubuntu@ubuntu:~/Documents$ We are in Child Process
Calling hello.c from child process
We are in Hello.c
PID of hello.c = 4791

```