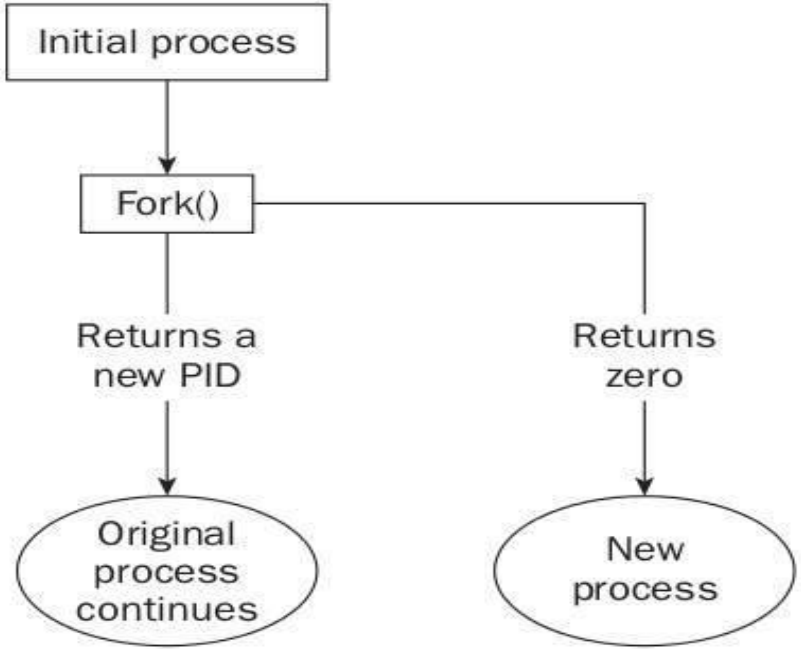
Tutorial-2

Operating System Lab

Fork

- 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()**


Initial process
Fork()
Returns a new PID
Returns zero
Original process continues
New process

- A child process use same pc(program counter), same CPU registers, same open files which use in parent process.
- It take no parameters and return integer value.
- Below are different values returned by fork():
  - ***Negative Value***: creation of a child process was unsuccessful.
  - ***Zero***: Returned to the newly created child process.
  - ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process. The returned process ID is of type **pid_t** defined in **sys/types.h**.
- A process can use function **getpid()** to retrieve the process ID assigned to this process.

Example 1:

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {
        fork();
        printf("Hello world!\n");
        return 0;
}
```

Output:

Hello world!
Hello world!

- 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.
- 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.

Example 2:

```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>
int main()
{
    // 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");
return 0;
}
```

Output:

1.
Hello from Child!
Hello from Parent!

(or)

2.
Hello from Parent!
Hello from Child!

- In the previous code, a child process is created, fork() returns 0 in the child process and positive integer to the parent process.
- Here, two outputs are possible because parent process and child process are running concurrently.
- So we don’t know if OS first give control to which process a parent process or a child process.

Pipe

- A *pipe* is a form of *redirection* that is used in Linux and other Unix-like operating systems to send the output of one program to another program for further processing.
- Redirection is the transferring of *standard output* to some other destination, such as another program, a file or a printer, instead of the display monitor (which is its default destination).
- Standard output, sometimes abbreviated *stdout*, is the destination of the output from *command line* (i.e., all-text mode) programs in Unix-like operating systems.

- Pipes are used to create what can be visualized as *a pipeline of commands*, which is a temporary direct connection between two or more simple programs.
- This connection makes possible the performance of some highly specialized task that none of the constituent programs could perform by themselves.
- A *command* is merely an instruction provided by a user telling a computer to do something, such as launch a program.
- The command line programs that do the further processing are referred to as *filters*.

Examples

- A pipe is designated in commands by the vertical bar character, which is located on the same key as the backslash on U.S. keyboards. The general syntax for pipes is:
  - command_1 | command_2 [| command_3 . . .]
- This chain can continue for any number of commands or programs.

Example:

- A very simple example of the benefits of piping is provided by the *dmesg* command, which repeats the startup messages that scroll through the *console* (i.e., the all-text, full-screen display) while Linux is *booting* (i.e., starting up).
- dmesg by itself produces far too many lines of output to fit into a single screen; thus, its output scrolls down the screen at high speed and only the final screenful of messages is easily readable.
    - dmesg | less
    - dmesg | sort -f | less

Example:

- The following uses three pipes to search the contents of all of the files in current directory and display the total number of lines in them that contain the string *Linux* but not the string *UNIX*:
  - cat * | grep "Linux" | grep -v "UNIX" | wc -l
- the *cat* command, which is used to read and *concatenate* (i.e., string together) the contents of files, concatenates the contents of all of the files in the current directory.
- The asterisk is a *wildcard* that represents *all* items in a specified directory, and in this case it serves as an argument to cat to represent all objects in the current directory.

Example(continued):

- cat * | grep "Linux" | grep -v "UNIX" | wc -l

- The first pipe sends the output of cat to the *grep* command, which is used to search text.
- The *Linux* argument tells grep to return only those lines that contain the string *Linux*.
- The second pipe sends these lines to another instance of grep, which, in turn, with its *-v* option, eliminates those lines that contain the string *UNIX*.
- Finally, the third pipe sends this output to *wc -l*, which counts the number of lines and writes the result to the display screen.