# Signals and processes in linux

In Operating System, the fork() system call is used by a process to create another process. The process that used the fork() system call is the parent process and process consequently created is known as the child process. A process is an active program i.e a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.

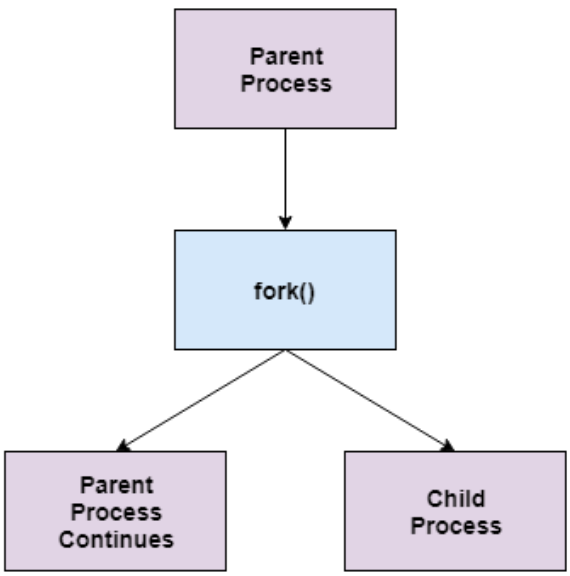
A process control block is associated with each of the processes. It contains important information about the process it is associated with such as process state, process number, program counter, list of files and registers, CPU information, memory information etc.

### Parent Process

A parent process may have multiple child processes but a child process only one parent process. On the success of a fork() system call, the PID of the child process is returned to the parent process and 0 is returned to the child process. On the failure of a fork() system call, -1 is returned to the parent process and a child process is not created.

### Child Process

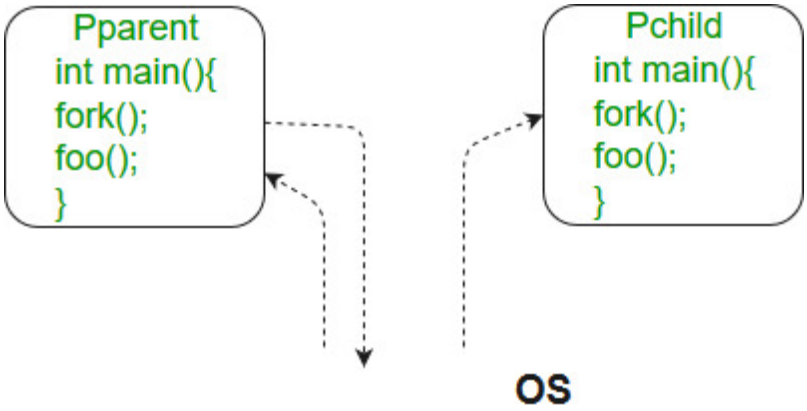
A child process may also be called a subprocess or a subtask. A child process is created as its parent process's copy and inherits most of its attributes. If a child process has no parent process, it was created directly by the kernel. If a child process exits or is interrupted, then a SIGCHLD signal is send to the parent process.



**fork() in C**

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which are used in the parent process.
It takes no parameters and returns an integer value.

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.

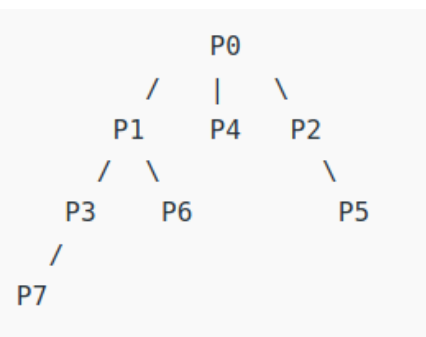



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

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

This example will print hello world twice.

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

This example will print hello world 8 times. 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. So here n = 3, 2^3 = 8



```c
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;
}
```

This example will output either:
Hello from child!
Hello from parent!

or

Hello from parent!
Hello from child!

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

Important: Parent process and child process are running the same program,
but it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different. See next example:

```c
void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

This example will output either:
Child has x = 2
Parent has x = 0

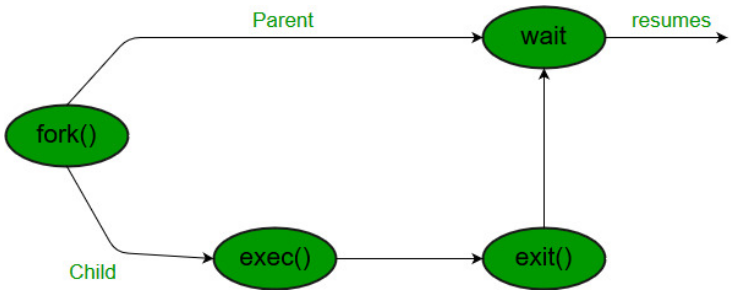
or

Parent has x = 0
Child has x = 2

Here, global variable change in one process does not affected two other processes because data/state of two processes are different. And also parent and child run simultaneously so two outputs are possible.

### Wait System Call in C

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction.
Child process may terminate due to any of these:
-It calls exit();
-It returns (an int) from main
-It receives a signal (from the OS or another process) whose default action is to terminate.



If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates. If only one child process is terminated, then return a wait() returns process ID of the terminated child process. If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process. When wait() returns they also define exit status (which tells our, a process why terminated) via pointer, If status are not NULL. If any process has no child process then wait() returns immediately “-1”.

```c
int main()
{
    pid_t cpid;
    if (fork()== 0)
        exit(0);            /* terminate child */
    else
        cpid = wait(NULL); /* reaping parent */
    printf("Parent pid = %d\n", getpid());
    printf("Child pid = %d\n", cpid);

    return 0;
}
```

This example will output:

Parent pid = 21413
Child pid = 21414

```c
int main()
{
    if (fork()== 0)
        printf("HC: hello from child\n");
    else
    {
        printf("HP: hello from parent\n");
        wait(NULL);
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
```

```
HC: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye
```

Status information about the child reported by wait is more than just the exit status of the child, it also includes:
-normal/abnormal termination
-termination cause
-exit status

to find information about status, we use WIF….macros
1. WIFEXITED(status): child exited normally
• WEXITSTATUS(status): return code when child exits
2. WIFSIGNALED(status): child exited because a signal was not caught
• WTERMSIG(status): gives the number of the terminating signal
3. WIFSTOPPED(status): child is stopped
• WSTOPSIG(status): gives the number of the stop signal

/*if we want to prints information about a signal */
void psignal(unsigned sig, const char *s);

## Signals in C language

In this article we are going to show you how to use signal handlers in Linux using C language. But first we will discuss what is signal, how it will generate some common signals which you can use in your program and then we will look how various signals can be handled by a program while the program executes. So, let's start.

### Signal

A signal is an event which is generated to notify a process or thread that some important situation has arrived. When a process or thread has received a signal, the process or thread will stop what its doing and take some action. Signal may be useful for inter-process communication.

### Standard Signals

The signals are defined in the header file signal.h as a macro constant. Signal name has started with a "SIG" and followed by a short description of the signal. So, every signal has a unique numeric value. Your program should always use the name of the signals, not the signals number. The reason is signal number can differ according to system but meaning of names will be standard.

The macro NSIG is the total number of signal defined. The value of NSIG is one greater than the total number of signal defined (All signal numbers are allocated consecutively).

Following are the standard signals:

| Signal Name | Description |
|---|---|
| SIGHUP | Hang-up the process. The SIGHUP signal is used to report disconnection of the user's terminal, possibly because a remote connection is lost or hangs up. |
| SIGINT | Interrupt the process. When the user types the INTR character (normally Ctrl + C) the SIGINT signal is sent. |
| SIGQUIT | Quit the process. When the user types the QUIT character (normally Ctrl + \) the SIGQUIT signal is sent. |
| SIGILL | Illegal instruction. When an attempt is made to execute garbage or privileged instruction, the SIGILL signal is generated. Also, SIGILL can be generated when the stack overflows, or when the system has trouble running a signal handler. |
| SIGTRAP | Trace trap. A breakpoint instruction and other trap instruction will generate the SIGTRAP signal. The debugger uses this signal. |
| SIGABRT | Abort. The SIGABRT signal is generated when abort() function is called. This signal indicates an error that is detected by the program itself and reported by the abort() function call. |
| SIGFPE | Floating-point exception. When a fatal arithmetic error occurred the SIGFPE signal is generated. |
| SIGUSR1 and SIGUSR2 | The signals SIGUSR1 and SIGUSR2 may be used as you wish. It is useful to write a signal handler for them in the program that receives the signal for simple inter-process communication. |

## Default Action Of Signals

Each signal has a default action, one of the following:

Term: The process will terminate.
Core: The process will terminate and produce a core dump file.
Ign: The process will ignore the signal.
Stop: The process will stop.
Cont: The process will continue from being stopped.

Default action may be changed using handler function. Some signal's default action cannot be changed. SIGKILL and SIGABRT signal's default action cannot be changed or ignored.

## Signal Handling

If a process receives a signal, the process has a choice of action for that kind of signal. The process can ignore the signal, can specify a handler function, or accept the default action for that kind of signal.

-If the specified action for the signal is ignored, then the signal is discarded immediately.
-The program can register a handler function using function such as signal or sigaction. This is called a handler catches the signal.
-If the signal has not been neither handled nor ignored, its default action takes place.

We can handle the signal using signal or sigaction function. Here we see how the simplest signal() function is used for handling signals.

```
int signal () (int signum, void (*func)(int))
```

The signal() will call the func function if the process receives a signal signum. The signal() returns a pointer to function func if successful or it returns an error to errno and -1 otherwise.

The func pointer can have three values:

1.SIG_DFL: It is a pointer to system default function SIG_DFL(), declared in h header file. It is used for taking default action of the signal.
2.SIG_IGN: It is a pointer to system ignore function SIG_IGN(),declared in h header file.
3.User defined handler function pointer: The user defined handler function type is void(*)(int), means return type is void and one argument of type int.

## SIGCHLD

SIGCHLD is normally sent to a process to notify that one of its child processes ended, so the parent process can collect its exit code.
When you fork a child process from a parent process, the SIGCHLD is set but the handler function is NOT executed. After the child exits, it trips the SIGCHLD and thus causes the code in the handler function to execute...
The SIGCHLD signal is sent to the parent of a child process when it exits, is interrupted, or resumes after being interrupted. By default the signal is simply ignored.

## Basic Signal Handler Example

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){

  //Return type of the handler function should be void
  printf("\nInside handler function\n");
}

int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example1.c -o Example1
tump@tump:~/Desktop/c_prog/signal$ ./Example1
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
4 : Inside main function
^\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

in the screenshot of the output of Example1.c, we can see that in main function infinite loop is executing. When user typed Ctrl+C, main function execution stop and the handler function of the signal is invoked. After completion of the handler function, main function execution resumed. When user type typed Ctrl+\, the process is quit.

## Ignore Signals Example

```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
int main(){
  signal(SIGINT,SIG_IGN); // Register signal handler for ignoring the signal

  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$
tump@tump:~/Desktop/c_prog/signal$
1 : Inside main function
2 : Inside main function
^C3 : Inside main function
4 : Inside main function
^C5 : Inside main function
^\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

Here handler function is register to SIG_IGN() function for ignoring the signal action. So, when user typed Ctrl+C,  SIGINT signal is generating but the action is ignored.

## Reregister Signal Handler Example

```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void sig_handler(int signum){
  printf("\nInside handler function\n");
  signal(SIGINT,SIG_DFL);   // Re Register signal handler for default action
}

int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$
tump@tump:~/Desktop/c_prog/signal$
1 : Inside main function
2 : Inside main function
3 : Inside main function
^C
Inside handler function
4 : Inside main function
^C
tump@tump:~/Desktop/c_prog/signal$
```

In the screenshot of the output of Example3.c, we can see that when user first time typed Ctrl+C, the handler function invoked. In the handler function, the signal handler re register to SIG_DFL for default action of the signal. When user typed Ctrl+C for second time, the process is terminated which is the default action of SIGINT signal.

## Sending Signals:

A process also can explicitly send signals to itself or to another process. raise() and kill() function can be used for sending signals. Both functions are declared in signal.h header file.

```c
int raise(int signum)
```

The raise() function used for sending signal signum to the calling process (itself).
It returns zero if successful and a nonzero value if it fails.

```
int kill(pid_t pid, int signum)
```

The kill function used for send a signal signum to a process or process group specified by pid.

## SIGUSR1 Signal Handler Example

```
#include<stdio.h>
#include<signal.h>

void sig_handler(int signum){
  printf("Inside handler function\n");
}

int main(){
  signal(SIGUSR1,sig_handler); // Register signal handler
  printf("Inside main function\n");
  raise(SIGUSR1);
  printf("Inside main function\n");
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$
tump@tump:~/Desktop/c_prog/signal$
Inside main function
Inside handler function
Inside main function
tump@tump:~/Desktop/c_prog/signal$
```

Here, the process send SIGUSR1 signal to itself using raise() function.

## Raise with Kill Example Program

```
#include<stdio.h>
#include <unistd.h>
#include<signal.h>
void sig_handler(int signum){
  printf("Inside handler function\n");
}

int main(){
  pid_t pid;
  signal(SIGUSR1,sig_handler); // Register signal handler
  printf("Inside main function\n");
  pid=getpid();      //Process ID of itself
  kill(pid,SIGUSR1);        // Send SIGUSR1 to itself
  printf("Inside main function\n");
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$
tump@tump:~/Desktop/c_prog/signal$
Inside main function
Inside handler function
Inside main function
tump@tump:~/Desktop/c_prog/signal$
```

Here, the process send SIGUSR1 signal to itself using kill() function. getpid() is used to get the process ID of itself.

In the next example we will see how parent and child processes communicates (Inter Process Communication) using kill() and signal function.

## Parent Child Communication with Signals

```
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
  printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
  printf("Child : Received a signal from parent \n");
  sleep(1);
  kill(getppid(),SIGUSR1);
}

int main(){
  pid_t pid;
  if((pid=fork())<0){
    printf("Fork Failed\n");
    exit(1);
  }
  /* Child Process */
  else if(pid==0){
    signal(SIGUSR1,sig_handler_child); // Register signal handler
    printf("Child: waiting for signal\n");
    pause();
  }
  /* Parent Process */
  else{
    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1);
    printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example6
tump@tump:~/Desktop/c_prog/signal$ ./Example6
Child: waiting for signal
Parent: sending signal to Child
Parent: waiting for response
Child : Received a signal from parent
Parent : Received a response signal from child
tump@tump:~/Desktop/c_prog/signal$
```

Here, fork() function creates child process and return zero to child process and child process ID to parent process. So, pid has been checked to decide parent and child process. In parent process, it is slept for 1 second so that child process can register signal handler function and wait for the signal from parent. After 1 second parent process send SIGUSR1 signal to child process and wait for the response signal from child. In child process, first it is waiting for signal from parent and when signal is received, handler function is invoked. From the handler function, the child process sends another SIGUSR1 signal to parent. Here getppid() function is used for getting parent process ID.