Linux OS—Working Modes

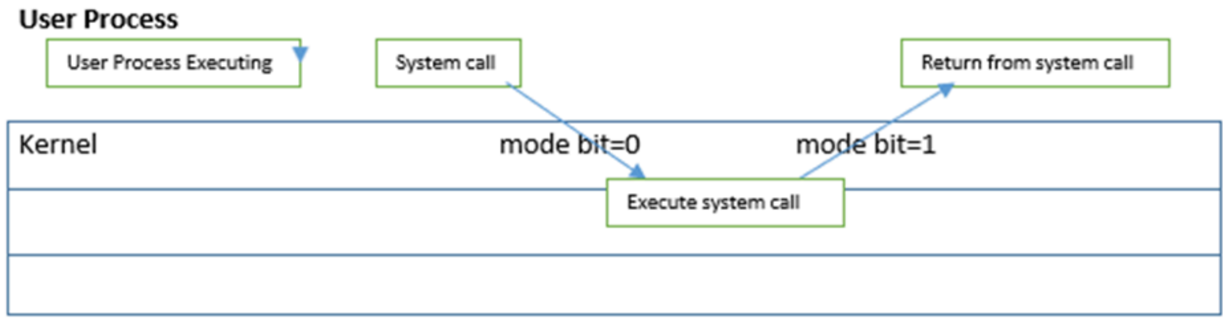
A system call is just what its name implies, a request for the operating system to do something on behalf of the user's program. In computing, a system call is how a program requests a service from an operating system's kernel. It provides an essential interface between a process and the operating system.

Dual-Mode Operation: Sharing system resources requires the operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly. One of the ways of protecting the programs in an operating system is by using two different modes.

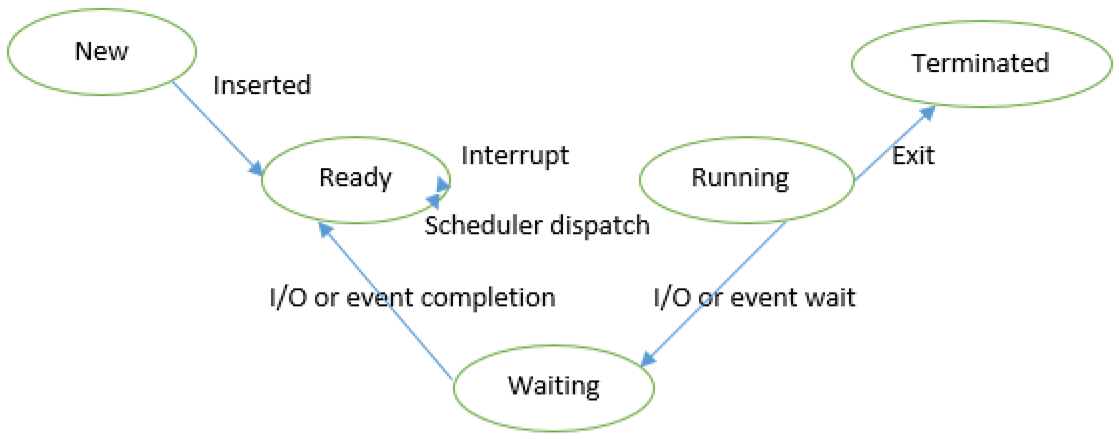
- User mode: execution is done on behalf of a user. It is a restricted mode that limits access to system resources.
- Kernel mode (monitor mode or system mode): execution done on behalf of the operating system. It is a privileged mode that allows access to system resources.

One of the ways to implement these two modes is by using a mode bit. When the value of the mode bit is 0, it is in monitor mode, but when the value is 1, it is in user mode.

When a user-level application needs to perform an operation that requires kernel mode access, such as accessing hardware devices or modifying system settings, it must make a system call to the operating system kernel. The operating system switches the processor from user mode to kernel mode to execute the system call and then switches back to user mode once the operation is complete.



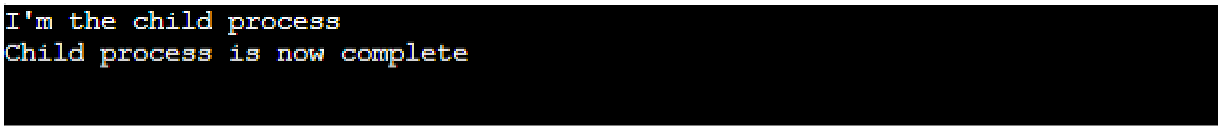
Process: A process is a program in execution. It consists of the data read from files, input from a user, program instructions, and many more. A process in Linux starts every time you start an application or run a program or command. The lifecycle of the process is shown as follows:



Initially, when the process is created, it enters the **new** state. Then, the operating system will allocate and initiate a process control block and some initial resources for this process. This is when it is admitted and is **ready** to start executing, but it is not actually executing on the CPU. Once the scheduler allocates the CPU to the ready process, it goes into the **running** state. The running process can be interrupted so that the context switch can be performed; this will move the running process back into the ready state. The running process might want to initiate some longer operations like reading data from the disk or waiting for some event; this is when it is moved to the **waiting** state. Once the I/O operation or event is complete, the process goes back to the ready state. When all the tasks of the process are complete or the process encounters any error, it needs to exit with an appropriate exit status and move to the **terminated** state.
fork(): A new process is created by the fork() system call. This function creates a new copy called the child out of the original process that is called the parent. When the parent process closes or crashes for some reason, it also kills the child process.
exit(): The exit() system call is used by a program to terminate its execution. The operating system reclaims resources that were used by the process after the exit() system call.
wait(): It blocks the calling process until one of its child processes exits or a signal is received. After the child process terminates, the parent continues its execution after the wait system call instruction.
waitpid(): It suspends execution of the current process until a child specified by the pid argument has changed state.
exec(): This family of functions replaces the current process image with a new process image. It loads the program into the current process space and runs it from the entry point. fork starts a new process, which is a copy of the one that calls it, whereas exec replaces the current process image with another one. Both parent and child processes are executed simultaneously in case of fork(), and control never returns to the original program unless there is an exec() error. Some of its variations are:
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);

Example: This program shows that by using wait, the parent process will wait for the child to complete.

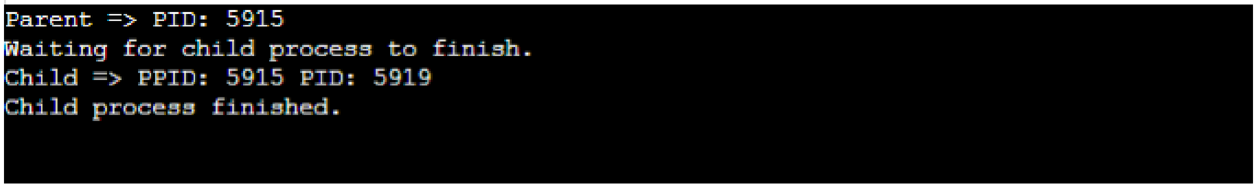
```
#include <sys/types.h>
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
 /* fork a child process */
 pid_t pid = fork();
 if (pid < 0) { /* error occurred */
        fprintf(stderr, "****Failure of fork system call****");
        return 1;
 }
 else if (pid == 0) { /* child process */
        printf("I'm the child process \n"); /* you can execute some commands here */
 }
 else { /* parent process */
        /* 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 process is now complete \n");
 }
}
```



```
I'm the child process
Child process is now complete
```

Example: Print the process IDs for parent and child.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main(void) {
 //using fork() function to create a child process from the main process
 pid_t pid = fork();
 if(pid == 0) {
   //printing the process ID and the parent ID i.e. PID and PPID respectively
   printf("Child => PPID: %d PID: %d\n", getppid(), getpid());
   exit(EXIT_SUCCESS); //exit() is used on the child process to finish the execution of the child
process
 }
 else if(pid > 0) {
   //printing the process ID i.e. PID
   printf("Parent => PID: %d\n", getpid());
   printf("Waiting for child process to finish.\n");
   wait(NULL); //wait (NULL) on the parent process is used to wait for the child process to get
finished.
   printf("Child process finished.\n");
 }
 else {
   printf("Unable to create child process.\n");
 }
 return EXIT_SUCCESS;
}
```

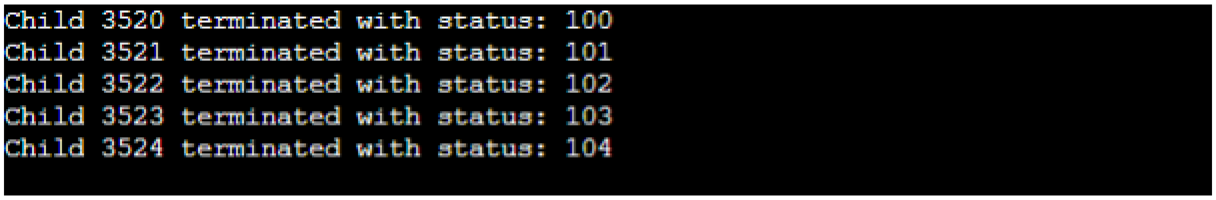


```
Parent => PID: 5915
Waiting for child process to finish.
Child => PPID: 5915 PID: 5919
Child process finished.
```

Example: Code to demonstrate the use of waitpid().

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
void waitdemo()
{
        int i, stat;
        pid_t pid[5];
        for (i=0; i<5; i++)
        {
                if ((pid[i] = fork()) == 0)
                {
                        sleep(1);
                        exit(100 + i);
                }
        }
        // Using waitpid() and printing exit status of children.
        for (i=0; i<5; i++)
        {
                pid_t cpid = waitpid(pid[i], &stat, 0);
                if (WIFEXITED(stat))// WIFEXITED(status): child exited normally
                        printf("Child %d terminated with status: %d\n",
                                cpid, WEXITSTATUS(stat));// WEXITSTATUS(status): return code
when child exits
        }
}
// main code
```

```
int main()
{
            waitdemo();
            return 0;
}
```



```
Child 3520 terminated with status: 100
Child 3521 terminated with status: 101
Child 3522 terminated with status: 102
Child 3523 terminated with status: 103
Child 3524 terminated with status: 104
```

Example: Demo of exec command.

demo.c

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

Hello.c

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

When we run demo.c we get the following output:

```
PID of demo.c = 4733
We are in Hello.c
PID of hello.c = 4733
```

Signals and Handling

signal(): Signals are a technique used to notify a process that some condition has occurred. A signal is a message or notification issued to a program by the operating system or another application.
Each signal may have a signal handler, which is a function that gets called when the process receives that signal. The function is called in “asynchronous mode,” meaning that nowhere in your program you have code that calls this function directly. Instead, when the signal is sent to the process, the operating system stops the execution of the process and “forces” it to call the signal handler function. When that signal handler function returns, the process continues execution from where it left.
A process can also explicitly send signals to itself or to another process. raise() and kill() functions can be used for sending signals. Both functions are declared in signal.h header file in C.

- **raise()**: sends a signal to the calling thread.
- **kill()**: sends a signal to a specified process, to all members of a specified process group, or to all processes on the system. It takes two arguments. The first, pid, is the process ID you want to send a signal to, and the second, sig, is the signal you want to send.

Signal Handler:

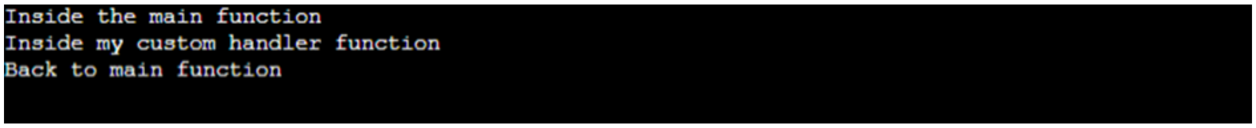
For our convenience, there are two pre-defined signal handler functions that we can use, instead of writing our own: SIG_IGN and SIG_DFL.
SIG_IGN: causes the process to ignore the specified signal.
SIG_DFL: causes the system to set the default signal handler for the given signal.
You can register your own signal handler by using signal() interface, which is the oldest one. It takes two arguments: a reference to a signal handler code and a signal number.
Example: User-defined signal handler.

```
// User-defined Signal Handler in C language
#include<stdio.h>
#include<signal.h>
// Handler for SIGINT, caused by action: “Ctrl-C at keyboard”
void handle_sig(int sig)
{
        printf("Caught a signal %d\n", sig);
}
int main()
{
        signal(SIGINT, handle_sig);//when user presses Clt-C signal will be caught
        while (1) ;//infinite loop
        return 0;
}
```

Example: Using raise() system calls.

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

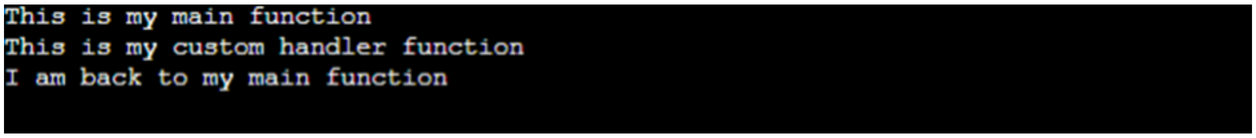
```
#include<signal.h>
void sig_handler(int signum){
 printf("Inside my custom handler function\n");
}
int main(){
 signal(SIGUSR1,sig_handler); // Register the signal handler
 printf("Inside the main function\n");
 raise(SIGUSR1);// process send SIGUSR1 signal to itself using raise() function
 printf("Back to main function\n");
 return 0;
}
```



```
Inside the main function
Inside my custom handler function
Back to main function
```

Example: Using kill to send signals.

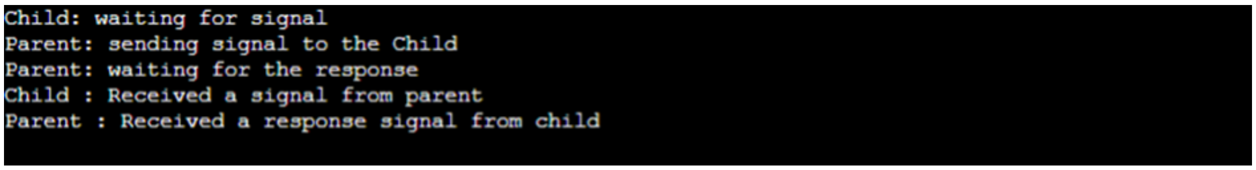
```
#include<stdio.h>
#include <unistd.h>
#include<signal.h>
void sig_handler(int signum){
 printf("This is my custom handler function\n");
}
int main(){
 pid_t pid;
 signal(SIGUSR1,sig_handler); // Register the signal handler
 printf("This is my main function\n");
 pid=getpid();      //get the Process ID of self
 kill(pid,SIGUSR1);// process sens SIGUSR1 signal to itself using kill() function
 printf("I am back to my main function\n");
 return 0;
}
```



```
This is my main function
This is my custom handler function
I am back to my main function
```

Example: Parent and child communication using kill and signal.

```
#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);
  //the child process sends another SIGUSR1 signal to parent and getppid()
 //function is used for getting parent process ID.
 kill(getppid(),SIGUSR1);
}
int main(){
 pid_t pid;
//fork() function creates child process and return zero to child process and child //process ID to
parent process.
 if((pid=fork())<0){ //create child and check for failure
   printf("Fork Failed\n");
   exit(1);
 }
 /* Child Process */
 else if(pid==0){
   //when signal is received from the parent, handler function is invoked
   signal(SIGUSR1,sig_handler_child); // Register signal handler
   printf("Child: waiting for signal\n");
   //waiting for signal from parent
   pause();
 }
 /* Parent Process */
 else{
   signal(SIGUSR1,sig_handler_parent); // Register signal handler
   //it is slept for 1 second so that child process can register signal handler
   //function and wait for the signal from parent.
   sleep(1);
   printf("Parent: sending signal to the Child\n");
   //After 1 second parent process send SIGUSR1 signal to child process and wait
  //for the response signal from child.
   kill(pid,SIGUSR1);
   printf("Parent: waiting for the response\n");
   pause();
 }
 return 0;
}
```



```
Child: waiting for signal
Parent: sending signal to the Child
Parent: waiting for the response
Child : Received a signal from parent
Parent : Received a response signal from child
```

How are system calls different from function calls?

- System call involves context switching (from user to kernel and back), whereas function call does not.
- System calls take much longer time than function calls.

- Most of the system calls return a value (error if failed), whereas it is not necessary for function calls.
- Avoiding excessive system calls might be a wise strategy for programs that need to be tightly optimized.