**1.Process and Process Management**

An instance of a running program is called a process. Every time you run a shell command, a program is run and a process is created for it. Each process in Linux has a process id (PID).

Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.

getpid()- to get the process ID.

getppid()-to gte the parent process ID.

A program can obtain the process ID of the process it’s running in with the **getpid(**) system call, and it can obtain the process ID of its parent process with the **getppid()** system call.

Ex:Printing the Process ID

**Process1.c**

#include <stdio.h>

#include <unistd.h>

int main ()

{

printf ("The process ID is %d\n", (int) getpid ());

printf ("The parent process ID is %d\n", (int) getppid ());

return 0;

}

**Compile:** gcc Process1.c

**execute:**./a.out

**Output:**

The process ID is 30263

The parent process ID is 30211

**Note:**Observe that if you invoke this program several times, a different process ID is reported because each invocation is in a new process. However, if you invoke it every time from the same shell, the parent process ID (that is, the process ID of the shell process) is the same

**When you start a process (run a command), there are two ways you can run it −**

* Foreground Processes
* Background Processes

### **Foreground Processes**

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the **ls** command. If you wish to list all the files in your current directory, you can use the following command −

$ls

lists all the contents of the present directory.

The process runs in the foreground, the output is directed to my screen, and if the ls command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in the foreground and is time-consuming, no other commands can be run (start any other processes) because the prompt would not be available until the program finishes processing and comes out.

### **Background Processes**

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

## **How to find out what processes are running in the background**

1. You can use the ps command to list all background process in Linux. Other Linux commands to obtain what processes are running in the background on Linux.
2. top command – Display your Linux server’s resource usage and see the processes that are eating up most system resources such as memory, CPU, disk and more.
3. htop command – Just like a top command but with an improved user interface.

Ex: Running sleep command in the background

--Enter the command followed by &

syntax: command &

$sleep 10000 &

--To list background processes

$jobs

--One can bring a background process to the foreground

syntax:$fg %n

$fg %1

--To stop the foreground process press CTRL+z.run the bg command to restart a stopped background process.

Sysntax:$fg %n

$fg %1

--Finally, kill a running process named “sleep 10000” using the

syntax: $kill %n

$kill %1

## **Linux background processes list command**

## Open the terminal application and issue the following ps command command to show all running process on the system including those running in the background:

## $ sudo ps -aux | less

## OR

## $ ps aux | more

**Description of Process STATE code**

|  |  |
| --- | --- |
| Process STATE code | Description |
| D | uninterruptible sleep (usually IO) |
| I | Idle kernel thread |
| R | running or runnable (on run queue) |
| S | interruptible sleep (waiting for an event to complete) |
| T | stopped by job control signal |
| t | stopped by debugger during the tracing |
| W | paging (not valid since the 2.6.xx kernel) |
| X | dead (should never be seen) |
| Z | defunct (“zombie”) process, terminated but not reaped by its parent |

**1.1 Viewing Active Processes**

The ps command displays the processes that are running on your system

$ps

o/p:

PID TTY TIME CMD

21693 pts/8 00:00:00 bash

21694 pts/8 00:00:00 ps

**1.2 Creating Processes**

Two common techniques are used for creating a new process.

1>Using system call

2>Using fork and exec

**1.2.1 Using system call**

The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell.

Ex:Using the system Call

**system.c**

#include <stdlib.h>

int main ()

{

int return\_value;

return\_value = system ("pwd /");

return return\_value;

}

**Compile:** gcc system.c

**execute:**./a.out

**Output:**/home/sheetaln/

**1.2.2 Using fork and exec**

fork-The fork is used to create a new process by duplicating the calling process. The new process is the child process. See the following property.

-The child process has its own unique process id.

-The parent process id of the child process is same as the process id of the calling process.

-The fork() returns the PID of the child process. If the value is non-zero, then it is parent process’s id, and if this is 0, then this is child process’s id.

exec-The exec() system call is used to replace the current process image with the new process image. It loads the program into the current space, and runs it from the entry point.

So the main difference between fork() and exec() is that fork starts new process which is a copy of the main process. the exec() replaces the current process image with new one, Both parent and child processes are executed simultaneously.

**Example program of fork and exec**

**Program1:forkexce.c**

#include<stdio.h>

#include<unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include<stdio.h>

int main()

{

printf("before creating child process\n");

int x;

x=fork();

if(x==0)

{

char\* arg[]={"Hello",NULL};

execv("cp",arg);

printf("C-After creating c\_p\n");

}

else

{

printf("P-After creating c\_p\n");

}

}

**Program2:child\_process.c**

#include<stdio.h>

main()

{

printf("in child process\n");

}

**Compile**: gcc -o cp child\_process.c

gcc -o FC forkexce.c

**execute:** ./FC

**Output:**

before creating child process

P-After creating c\_p

in child process

**1.3 Zombie Process**

When a process is created in UNIX using fork() system call, the address space of the Parent process is replicated. If the parent process calls wait() system call, then the execution of parent is suspended until the child is terminated. At the termination of the child, a ‘SIGCHLD’ signal is generated which is delivered to the parent by the kernel. Parent, on receipt of ‘SIGCHLD’ reaps the status of the child from the process table. Even though, the child is terminated, there is an entry in the process table corresponding to the child where the status is stored. When parent collects the status, this entry is deleted. Thus, all the traces of the child process are removed from the system. If the parent decides not to wait for the child’s termination and it executes its subsequent task, then at the termination of the child, the exit status is not read. Hence, there remains an entry in the process table even after the termination of the child. This state of the child process is known as the Zombie state.

Example that creats zombie process.

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int main()

{

int i;

int pid = fork();

if (pid == 0)

{

for (i=0; i<20; i++)

printf("I am Child\n");

}

else

{

printf("I am Parent\n");

while(1);

}

}

### **Different ways in which the creation of Zombie can be Prevented**

### **1>Using wait() system call :**

### Ex:

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int main()

{

int i;

int pid = fork();

if (pid == 0)

{

for (i=0; i<20; i++)

printf("I am Child\n");

}

else

{

wait(NULL);

printf("I am Parent\n");

while(1);

}

}

### **2>By ignoring the SIGCHLD signal :**

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

#include<sys/types.h>

int main()

{

int i;

int pid = fork();

if (pid == 0)

{

for (i=0; i<20; i++)

printf("I am Child\n");

}

else

{

signal(SIGCHLD,SIG\_IGN);

printf("I am Parent\n");

while(1);

}

}

**1.4 Orphan Processes**

Normally, when a child process is killed, the parent process is updated via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," the init process, becomes the new PPID (parent process ID). In some cases, these processes are called orphan processes.

**1.5 Daemon Processes**

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

**NOTE:**The **top** command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Ex: $top