



**SET-224 /CET-225**  
**Operating Systems**

**LAB # 06**

**LAB Title**

Understanding of Process, Process creation using Fork (), Zombie Process and Orphan Process.

**Assessment of CLO: 04, PLO: 05**

Student Name:			
Roll No.			
Semester		Session	

S. No.	Perf. Level Criteria	Excellent (2.5)	Good (2)	Satisfactory (1.5)	Needs Improvement (0 ~ 1)	Marks Obtained
1	Project Execution & Implementation	Fully functional, optimized, and well-structured.	Minor errors, mostly functional.	Some errors, requires guidance.	Major errors, non-functional, or not Performed.	
2	Results & Debugging Or Troubleshooting	Accurate results with effective debugging Or Troubleshooting.	Mostly correct, some debugging Or Troubleshooting needed.	Partial results, minimal debugging Or Troubleshooting.	Incorrect results, no debugging Or Troubleshooting, or not attempted.	
3	Problem-Solving & Adaptability (VIVA)	Creative approach, efficiently solves challenges.	Adapts well, minor struggles.	Some adaptability, needs guidance.	Lacks innovation or no innovation, unable to solve problems.	
4	Report Quality & Documentation	Clear, structured, with detailed visuals.	Mostly clear, minor gaps.	Some clarity issues, missing details.	Poorly structured, lacks clarity, or not submitted.	
<b>Total Marks Obtained Out of 10</b>						

**Experiment evaluated by**

Instructor's Name	Engr.Bushra Aziz		
Date		Signature	

### Objective:

To be familiar with the Process. How processes are created. Understanding of Zombie process and Orphan process.

### THEORY

Everything that runs on a Linux system is a process—every user task, every system daemon— everything is a process. Knowing how to manage the processes running on your Linux system is an important (indeed even critical) aspect of system administration.

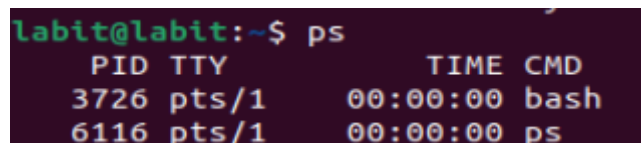
There are several definitions of the term process, including:

- A program in execution.
- An instance of Program running on a computer.
- The entity that can be assigned to and executed on a processor.
- A unit of activity with an associated set of system resources.

### Using ps command

The easiest method of finding out what processes are running on your system is to use the ps (process status) command. The ps command has a number of options and arguments, although most system administrators use only a couple of common command-line formats. We can start by looking at the basic usage of the ps command, and then examine some of the useful options. The ps command is available to all system users, as well as root, although the output changes a little depending on whether you are logged in as root when you issue the command.

For example, you might see the following output when you issue the command:



```
labit@labit:~$ ps
  PID TTY          TIME CMD
 3726 pts/1        00:00:00 bash
 6116 pts/1        00:00:00 ps
```

Result contains four columns of information. Where,

PID – the unique process ID

TTY – terminal type that the user is logged into

TIME – amount of CPU in minutes and seconds that the process has been running CMD – name of the command that launched the process.

To obtain much more complete information about the processes currently on the system.

### ps -aux

-To display a tree of processes. **pstree**

-To see currently running processes and other information like memory and CPU usage with real time updates.

### top

The -e option generates a list of information about every process currently running. The -f option generates a listing that contains fewer items of information for each process than the -l option.

- **ps -ef | less**

## Lab Experiment 6: Process creation using Fork (), Zombie Process and Orphan Process.

---

Note: For more understanding and to know about more options for ps command go to man page. “man ps”

### Process Creation

A “parent process” is a process that has created one or more child processes. In UNIX, every process except process 0 is created when another process executes the fork system call. The process that invoked fork is the parent process and the newly created process is the “child process”. Every process (except process 0) has one parent process, but can have many child processes. The kernel identifies each process by its process identifier (PID). Process 0 is a special process that is created when the system boots. Process 1, known as init, is the ancestor of every other process in the system.

When a child process terminates execution, either by calling the exit system call, causing a fatal execution error, or receiving a terminating signal, an exit status is returned to the operating system. A parent will typically retrieve its child's exit status by calling the wait system call. However, if a parent does not do so, the child process becomes a “zombie process.”

Following table illustrates various process system calls.

General Class	Specific Class	System Call
Process Related Calls	Process Creation and Termination	exec(), fork(), wait(), exit()
	Process Owner and Group	getuid(), geteuid(), getgid(), getegid()
	Process Identity	getpid(), getppid()
	Process Control	signal(), kill(), alarm()

### The fork() System Call

Fork system call use for creates a new process, which is called child process, which runs concurrently with process (which process called system call fork) and this process is called parent process. After a new child process 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 use in the parent process.

It takes no parameters and returns an 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.

fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:

- The child has its own unique process ID, and this PID does not match the ID of any existing process group.
- The child's parent process ID is the same as the parent's process ID.

Predict the Output of the following programs to understand fork()

### Program1:

```
# importing os module import os
os.fork()
print("I am process:")
```

### Program2:

```
# importing os module import os
os.fork()
os.fork()
os.fork()
print("I am process:")
```

### Wait() Method Of Python Os Module

The wait() method of os module in Python enables a parent process to synchronize with the child process. i.e, to wait till the child process exits and then proceed.

### Example Program:

```
# import the os module of Python import os
# Create a child process

import os
import time

#os.fork()
#print ("i am parent process")
#os.fork()
#print ("using fork command")
retval=os.fork()
if retval == 0:
    child_id=os.getpid()
    print (f'i am child process '{child_id}''')
else:
    parent_id=os.getpid()
```

```
childexit=os.wait()
print (f" child exit '{childexit[0]}'" )
print (f'hello i am parent '{parent_id}')
```

### Orphan and Zombie Process State

#### Zombie Processes

On Unix and Unix-like computer operating systems, a "zombie process" or defunct process is a process that has completed execution but still has an entry in the process table. This entry is still needed to allow the parent process to read its child's exit status. The term zombie process derives from the common definition of zombie as an undead person. When a process ends, all of the memory and resources associated with it are deallocated so they can be used by other processes. However, the process's entry in the process table remains. The parent can read the child's exit status by executing the wait system call, whereupon the zombie is removed.

To observe the creation of Zombie Process, Python sleep() call will be used to simulate a delay in the parent process.

Example Program:

```
# Create a child process import os
import time
retval=os.fork()
if retval == 0 :

print("Child process Runnin")
print("Child process Running")
print("Child process Running")
print("Child process exiting")
else:
# Parent process did not use wait()
time.sleep(60)
print("Now Parent process Running")
```

Zombie or defunct process can be seen in another instance of the terminal by using ps -ef, just after running the above program.

Note: The Process with status as Z or marked as <defunct> i.e., de-functioning is ZOMBIE Process.

#### Orphan Processes

An "orphan process" is a computer process whose parent process has finished or terminated, though it remains running itself. In a Unix-like operating system any orphaned process will be immediately adopted by the

## Lab Experiment 6: Process creation using Fork (), Zombie Process and Orphan Process.

---

special init system process. This operation is called reparenting and occurs automatically. Even though technically the process has the \init" process as its parent, it is still called an orphan process since the process that originally created it no longer exists. A process can be orphaned unintentionally, such as when the parent process terminates or crashes.

Example Program:

```
import os
import time

retval=os.fork()
if retval == 0 :
print("Child process Runnin")
print("Child process Running")
print("Child process Running")
print("Child process Going To Sleep for 60 seconds")
time.sleep(60)
print("Child process Running after a nap ")
print("Child process exiting") else:
# Parent process Running
print("Parent process Running and exiting.")
```

To observe the creation of Orphan Process, Python sleep() call will be used to simulate a delay in the child process. Parent id of this orphan process i.e 1 , can be seen in another instance of the terminal by using ps -ef, just after running the above program.

Compile and Run Python program

```
$ python lab6.py
```

### EXERCISES

1. Using a Linux system, write a program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds.
2. Write a program that creates a child process which further creates its two child processes. Display the process id of the terminated child to understand the hierarchy of termination of each child process.

3. Write a program that create child process then parent call the wait() function to know status of child also display process id of both processes.