National University of Computer and Emerging Sciences

Operating System Lab - 04

Lab Manual

Contents

Objective

The objective of this lab is to learn about Linux process management, commands to see running processes, system call to create processes, get their IDs, and wait and exec system call and to learn about zombie and orphan processes.

Process Management

In general, a process is an instance of a program written and compiled. There can be multiple instances of a same program. In other words, a program that is loaded into computer's memory and is in a state of execution is called a process. Processes are dynamic entity. Process essentially requires CPU and RAM resources but may also require I/O, Network or Printer depending on the program written.

Linux Processes

A Linux process or task is known as the instance of a program running under Linux environment. This means that if 10 users from servers are running gedit editor, then there are 10 gedit processes running on the server. Although they are sharing same executable code. The processes running in a Linux system can be view using 'ps' command which we covered in *lab manual 02.*

Process ID

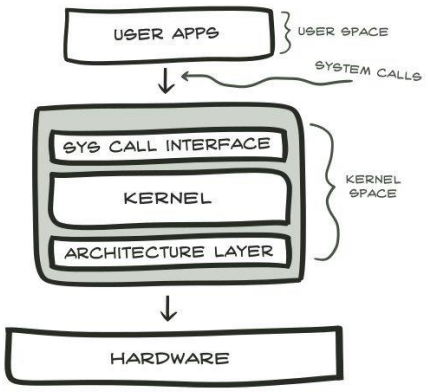
In Linux system, each process running has been assigned a unique ID which is known as PID (Process Identification Number). For example, Firefox is a running process if you are browsing the internet. Each time you start a new Firefox browser the system will automatically assign a PID to the new process of Firefox. A PID is automatically assigned when a new process is created on the system. If you wish to find out the PID of the running process like Firefox you may use the command 'pidof'.

```
$ pidof firefox
$ pidof bash
$ pidof bioset
```

To view the running process in a form of a tree we can use 'pstree' command. Which shows the running process in a form of a tree.

System Call Related to Process

There are a lot many Linux system calls available inbuilt for the users. A system call is as good as a function in C Programming Language. It can be compared with 'printf' function in C. The reason that it is often called system call rather than functions is that functions are limited to programming while system calls are specific for operating systems.



Fork System Call

In Linux, process is created by duplicating parent process. This is called forking. You invoke the fork system call with fork() function.

Definition

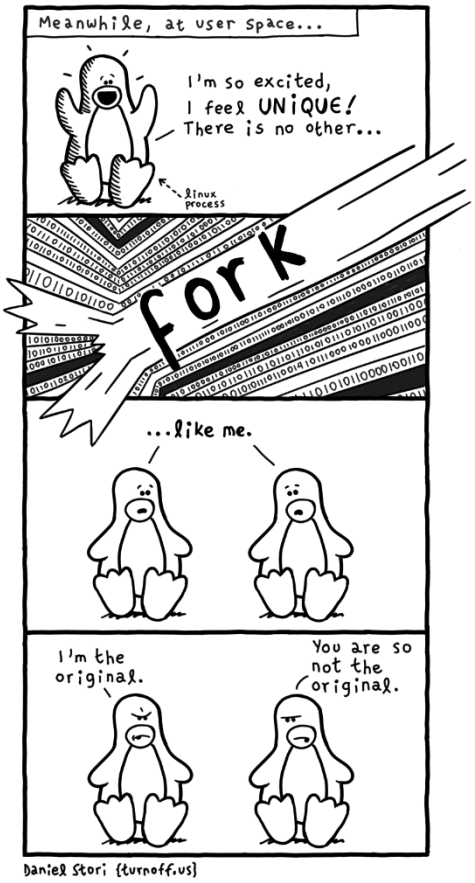
It is a system call that creates a new process under Linux operating system. It takes no argument. The purpose of fork() is to create a new process which becomes the child process to the caller. After the new child process is created, both processes will execute next instruction following the fork system call, therefore we have to distinguish the parent process from the child which can be done by evaluating the returned value of fork() function.

Returned Value from Fork

By examining the returned value of fork function, we can determine whether the child process is created and/or what is the pid of the child process. The following are the returned value and their meaning.

- If the returned value is negative, it means that the child process creation was unsuccessful
- If the returned value is zero, the child process is created with pid = 0
- If the returned value is positive, the child process is created with the process with a process ID to the parent process (The returned process ID is of type pid_t defined in sys.type.h). Normally this process ID is an integer.

After the system call to fork() is issued, a simple test can tell which process is the child. Note that Linux will make an exact copy of the parent address space and give it to the child. Therefore, the parent and child processes have separate address space.

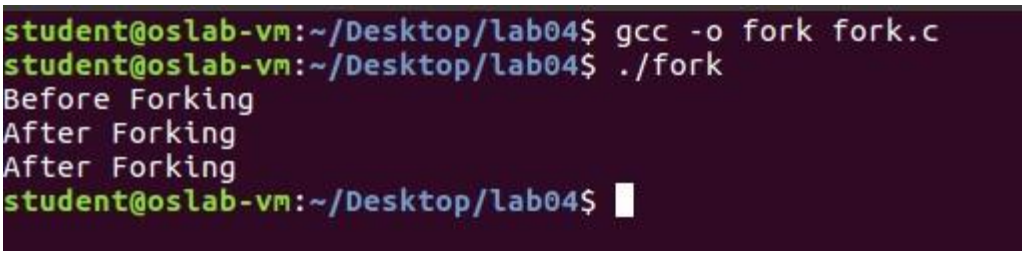


Example

Consider the following C language program:

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

The compilations and execution screenshot of the above code is shown below:



The fork() will make a copy of the process just as it executes it and then each process will execute next instructions after fork() execution. The above screenshot clarifies this statement that after fork the next statement ran twice, once by the parent process and the another by the child process. 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 for 'fork()' Distinguishing Parent and Child

The following codes distinguishes parent and child process:

```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>

int i = 5;

void parent_process();
void child_process();

int main() {
	pid_t pid;
	pid = fork();

	if(pid == 0) {
		i += 10;
		child_process();
	}
	else {
		parent_process();
	}
	return 0;
}
```

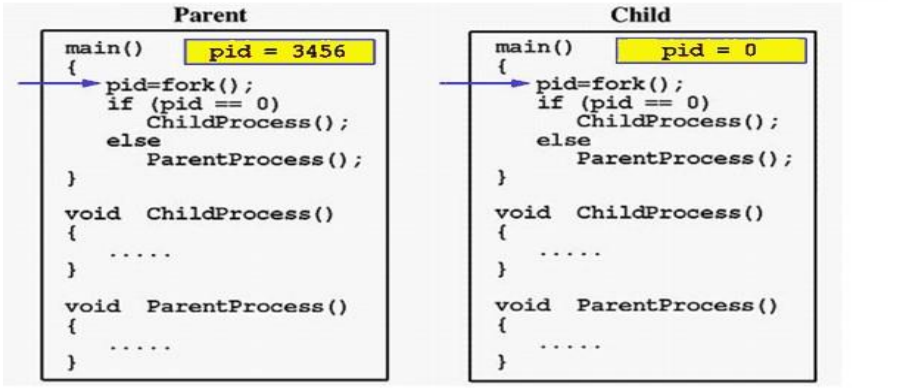
```
void parent_process() {
        printf("I am a parent process and my value of 'i' is %d \n",i);
}
void child_process() {
        printf("I am a child process and my value of 'i' is %d \n",i);
}
```

The screen shot below shows the execution of the above code:

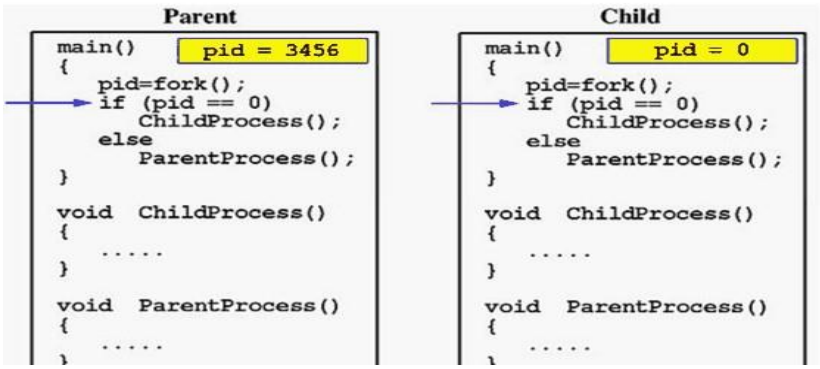
```
student@oslab-vm:~/Desktop/lab04$ gcc -o fork_distinguish fork_distinguish.c
student@oslab-vm:~/Desktop/lab04$ ./fork_distinguish
I am a parent process and my value of 'i' is 5
I am a child process and my value of 'i' is 15
student@oslab-vm:~/Desktop/lab04$
```

In the program above, both process prints the line, whether the line is printed by parent or child followed by the value of the variable 'I'. When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process.

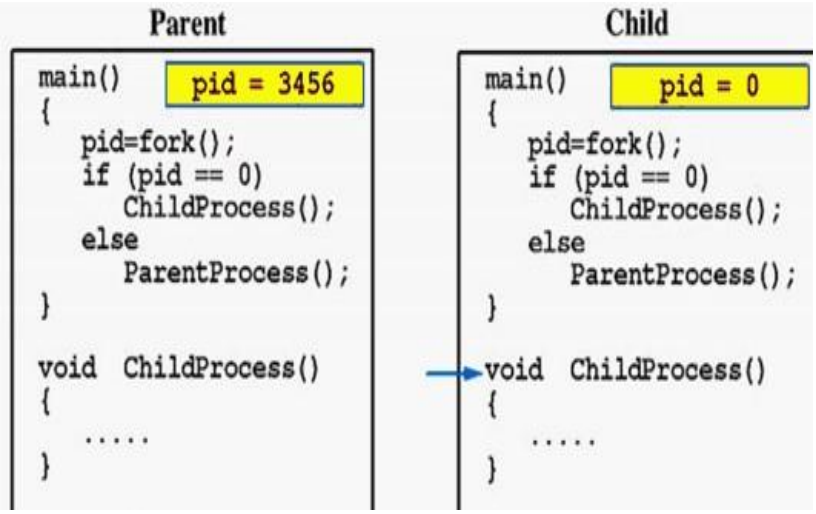
The images below will diagrammatically show the above code execution of forking:



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the above image, the fork function is called, which creates a new process and assign value of pid in parent process and assign 0 in child process. During if condition, it checked if the pid is zero or not to distinguish between parent and child process and invoke the functions of each process respectively.



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.

System Call to Get Process IDs

There are two system calls (functions) which can get the Process ID one is to get the process ID of the current process and another one to get the ID of its parent process. These are:

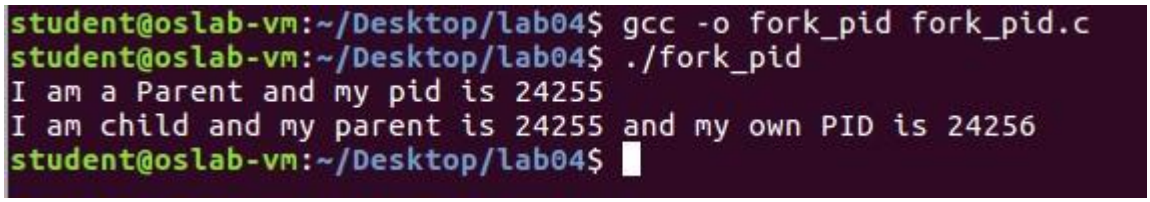
- 'getpid()' returns the PID of current process
- 'getppid()' returns the PID of parent's process

Consider the following example that prints PID, if a child then it prints its parent's PID and if it's a parent then it prints its own PID.

```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>

int main() {
        pid_t pid;
        pid = fork();
        if(pid == 0) {
                printf("I am child and my parent is %d and my own PID is %d\n", getppid(), getpid());
        }
        else if(pid > 0) {
                printf("I am a Parent and my pid is %d\n", getpid());
        }
        return 0;
```

```
}
```

The screenshot bellows shows the running instance of the above code.



In the above example, after the fork, if the process is a child it prints its own PID along with its parent PID else if it is a parent then it prints its own PID and exits.

Execute System Call 'exec'

Fork allow you to create a process by duplicating the process of the current program, that is from whom it is called. However, limitation arises when you wish to execute a different program and this is where execute system call come in handy! The exec system call causes a process to invoke another executable. There are many functions that act as front ends to the exec system call (see the exec man page). These functions replace the image of the current process' with the executable image specified as an argument to exec. For example, the following code tries to launch gedit:

```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>

int main() {
        execl("/bin/ls", "ls", (char*)0);
        /* We can only reach this code if execl returned with
        * an error
        */
        perror("execl");
        return 0;
}
```

If the above code is compiled and ran successfully, you should get a window of 'gedit' editor of 'untitled document' which would be blank.

Alternate Ways

There are alternate ways to call exec system call, above is one example/way and some of other ways is listed below:

execlp("ls","ls","-lrt",(char*)0**);** this is similar to execl(..) the only difference is that the full path of the command have to be issued in execl(). And another usual method is shown below:

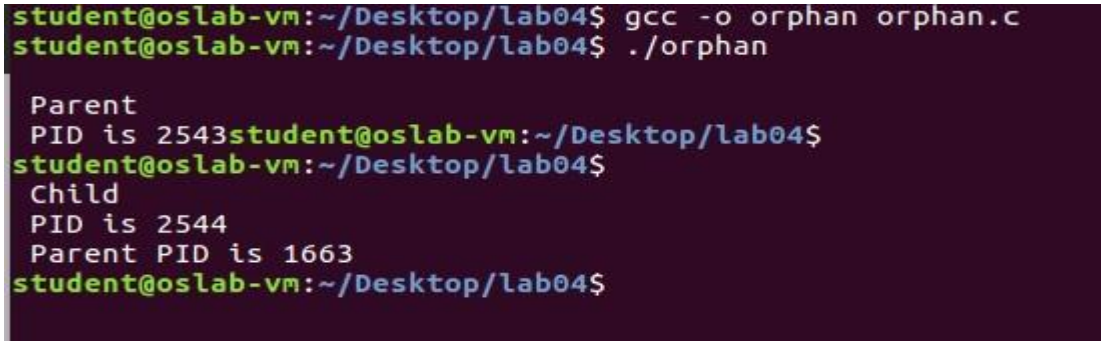
```
char * execarg[]= {“echo”,”Hello world”, NULL}
execvp (“echo”, execarg);
```

Orphan Process

In general English terms, orphan is someone who lost parents. Same is the story here. If the parent process has finished execution and exited, but at the same time if the child process remains unexecuted, the child is then termed to be an orphan. This is done by making the child process sleep for sometimes. By that time of child’s sleep, parent process will complete its execution and will exit. Since, parent process is no more there, child is referred to be an orphan now.

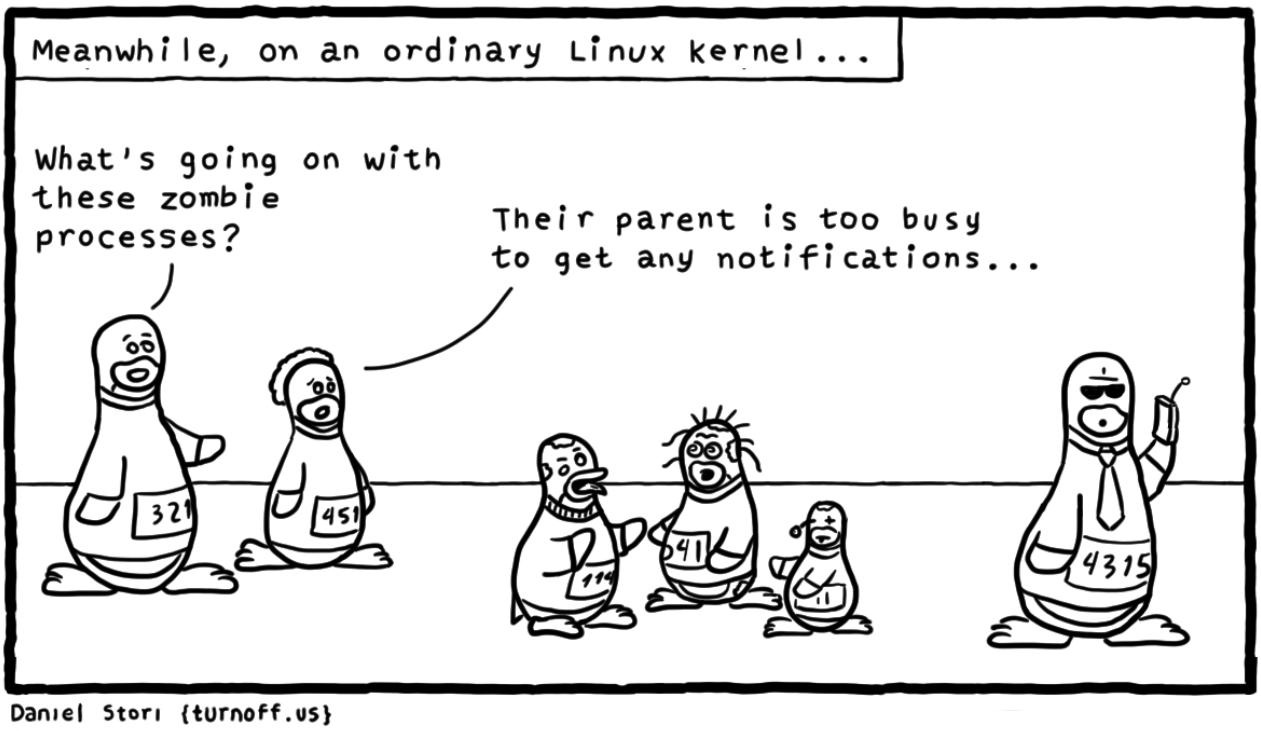
```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>

int main() {
        int pid = fork();
        if (pid > 0) {
        //parent process
                sleep(1);
                printf("\n Parent ");
                printf("\n PID is %d", getpid());
        }
        if(pid == 0) {
                sleep(5);
                printf("\n Child ");
        printf("\n PID is %d",getpid());
                printf("\n Parent PID is %d",getppid());
        }
        return 0;
}
```

When the execution is on, one can get the following information where process ids for both parent and child are clearly displayed. And parent process would be executed first and process id will be obtained for parent first. Then after the span of 5 seconds, child gets executed and its process id will be obtained. The child referred to be an orphan. Below screenshot shows the execution of the above program.

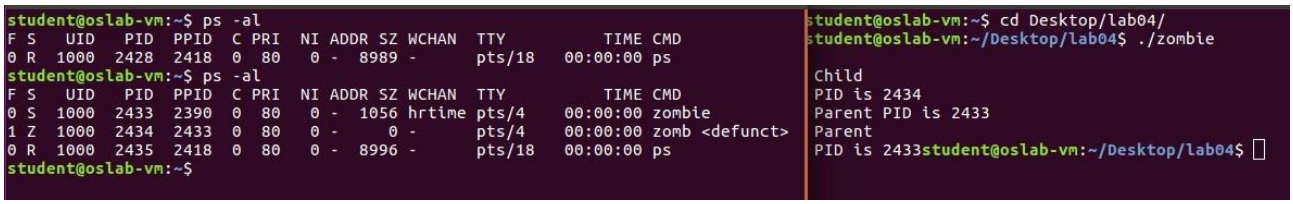


Zombie Process

A child when ready to inform its exit status to parent, but the parent process is busy in doing something (maybe sleeping), the child is called to be a Zombie or Defunct. A small piece of code is presented below which can be serve as an example for zombie process.




```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>

int main() {
        int pid = fork();
        if(pid > 0) {
                sleep(10);
                printf("\n Parent");
                printf("\n PID is %d",getpid());
        }
        if(pid == 0) {                  printf("\n Child");
                printf("\n PID is %d", getpid());
        printf("\n Parent PID is %d", getppid());
        }
        return 0;
}
```



Execution is slightly different here and usage of 2 terminals is required here. To find the presence of Zombie, 'ps -al' command has to be issued which will display the presence Zombie process as shown in the image above.

Solution to Avoid Creation of Zombie Processes

Using wait() system call we can avoid the creation of zombie process. The below changes to the above code will force the parent to wait

```
#include<stdio.h>
#include <sys/types.h>
#include<unistd.h>
#include<sys/wait.h>

int main() {
        int pid = fork();
        if(pid > 0) {
                wait(NULL);
                sleep(10);
                printf("\n Parent");
                printf("\n PID is %d",getpid());
        }
        if(pid == 0) {

                printf("\n Child");
                printf("\n PID is %d", getpid());
                printf("\n Parent PID is %d", getppid());
        }
        return 0;
 }
```

What happens when the main make-zombie program ends when the parent process exits, without ever calling wait? Does the zombie process stay around? No—try running ps again, and note that both of the make-zombie processes are gone. When a program exits, its children are inherited by a special process, the init program, which always runs with process ID of 1 (it's the first process started when Linux boots).The init process automatically cleans up any zombie child processes that it inherits.

Lab Activity

Activity 1)

Write a program that accomplish the following purpose:

a) Call the system call to create the child process and store the value returned from the call.
   a. If the returned value is less than zero,
      i. Print 'Unsuccessful Child Process Creation". ii. Terminate using exit system call
   b. If the return value is greater than zero
      i. Add a wait system call so that the parent would wait for child process to complete.
      ii. Make a loop that prints even numbers from 1 - 10
      iii. Print "Parent Ends"
   c. If the return value is equal to zero
      i. Print the parent ID
      ii. Make a loop that prints odd numbers from 1 - 10 iii. Print "Child Ends"

b) Stop

Activity 2)

Write a program to declare a counter variable initialized by zero. After fork() system call two processes will run in parallel both incrementing their own version of counter and print numbers 1 -5 . After printing numbers child process will sleep for three second, then print process id of its grandparent and terminates by invoking a gedit editor. Meanwhile, its parent waits for its termination.

Activity 3)

Write a program which creates processes 4 processes for parallel programming. Each parent will wait for the termination of its child.

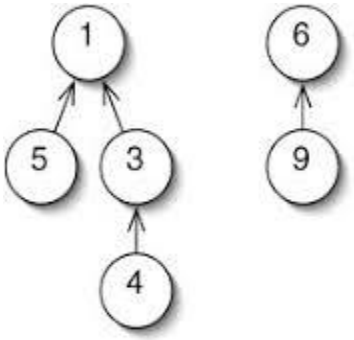
Main process	Process P1 waiting for p2	Process p2 waiting for p3	Process p3 runs a bash command

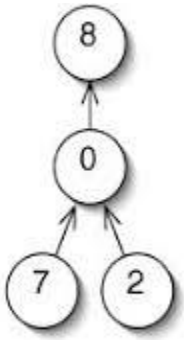
Activity 4)

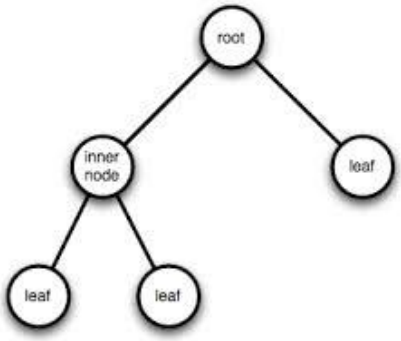
Google execl(), execlp() system call and study more about it than what is written in the manual.
https://www.mkssoftware.com/docs/man3/execl.3.asp

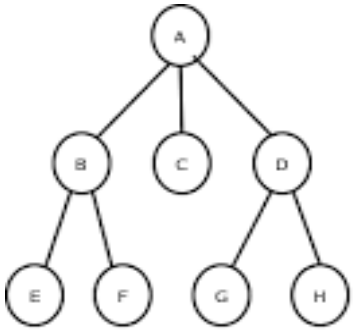
Activity 5)

Implement the following 9 tree structure. Each node must print its name and PID.
e.g. I am Process A and my PID is 2453









Logical View