Linux Systems Call

Objective

To write a program using following system calls of LINUX operating system: fork, getpid, wait.

Description

When a computer is turned on, the program that gets executed first is called the *operating system*. It controls pretty much all the activity in the computer. This includes who logs in, how disks are used, how memory is used, how the CPU is used, and how you talk with other computers. The operating system we usually use is "Unix".

The way programs talk to the operating system is via *system calls*. A system call looks like a procedure call (see below), but it's different – **it is a request to the operating system to perform some activity.**

getpid()

Each process is defined by a unique process id called as pid. The init process (which is the supreme parent to all processes) possesses id 1. All other processes have some other (possibly arbitrary) process id. The getpid system call returns the current process's id as an integer.

```
//…

int pid = getpid();

printf ("The process's id is %d\n", pid); …//
```

fork()

the fork system call creates a new child process. Actually, it's more accurate to say that it *forks* a currently running process. That is, it creates a *copy* of the current process as a new child process, and then both processes resume execution form the fork() call. Since, it creates two processes, fork also returns two values; one to each process. To the parent process, fork returns the *process id of the newly created child process.* To the child process, fork returns 0. The reason it returns 0 is precisely because this is an invalid process id. You would have no way of differentiating between the parent and child processes if fork returned an arbitrary positive integer to each.

Therefore, a typical call of fork looks something like this:

```
int pid;

if ((pid = fork()) == 0) {

   /*child process executes inside here */

}

else {

  /*parent process executes inside here */

}
```

wait()

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.

kill()

The kill() system call attempts to kill a running process pid passed to it.

```
// kill(pid == 2);                    it will kill the process with pid 2
```

LAB TASK

1. Write a program using fork(). It will create two processes: a parent process with some random *pid* say 4059 and a child process with *pid* 0. Assign child process with some id e.g 123. The child process should print “Hello from child 123. The parent process should print “Goodbye from parent *pid*”. You should ensure that the child process always prints first.
2. Extend this program and create children of process 123. Assign *pid* 124 to the child. Call fork() command to child 124 and the parent process say 4060 (fork command for each

child of 123). Assign the child processes with *pids* 125 and 126. Kill the parent process created by *pid* 4060 as shown below with red process as the killed process: