BITS Pilani - Hyderabad Campus
Operating Systems (CS F372)

Tutorial - 3

The objective of this tutorial is to introduce system calls and commands related to process creation and tracing in the Linux OS.

1. System calls - what are they? How are they different from function calls.

What is *fork* system call used for?
Let's try some programs to understand these system calls.

```
#include<stdio.h>
#include<unistd.h>
int main()
{
fork();
printf("hello \n"); /* NOTE: The code following the fork() runs in two
                        processes – the original process(parent), and the
                        child process. The child gets a copy of all the
                        local variables from parent */
return 0;
}
```

The fork() call returns different values in parent(pid of child) and child(0). This is used to differentiate a child from a parent process. Normally, a different program is run in the child process (using exec() family of system calls). Commands that we enter in command line are run in this way by the Shell.

Parent normally waits for the child process. Only after the child finishes its execution, the parent resumes. This is done by the wait() or waitpid() system call.
getpid() returns PID of the current process, getppid() returns PID of its parent.

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

int main()
{
        pid_t p;
        int status;
        p = fork();
        if(p < 0) {
                perror("fork");
        } else if (p == 0) { /* child process */
                printf("This is child process and the PID is %d\n", getpid());
```

```
                sleep(5);
                printf("child done\n");
        } else {   /* parent process */
                printf("This is parent process and PID of child is %d\n", p);
                wait(&status);
                printf("parent done\n");
        }
        return 0;
}
```

2. How can you trace system calls in Linux?

Use *strace* to trace all the system calls called by a program as follows:

Assuming you have compiled executable file a.out:

```
$ strace ./a.out
```

It will display all the system calls that are called by the process *a.out*. Remember that the shell creates a child process to run any command, so you see execve as the first system call in the output of strace. The shell is executing the program a.out in the child process (that it created before the strace started tracing…). execve is a system call to execute a program. In the strace output, look for clone system call.

To follow the child process in the trace, give –f option to strace:

```
$ strace –f ./a.out
```

It will follow the child as well (i.e. it will print system calls executed in the child process also).

3. Write a program to create a zombie process. (What is a zombie process?)

Zombie/defunct process : it has finished execution but the parent has not yet reaped it.. (what is reaping?)

Use sleep() system call to help create the above processes.

The program to create a zombie process will contain the following fragment of code:

```
if ((p = fork()) < 0) {
    perror("fork");
    exit(1);
} else if (p == 0) {
    printf("child\n");
} else {
    printf("parent\n");
    sleep(60); /* sleep for enough time to let the child
                  terminate and allow us to check out the status
                  of child process */
}
```

You can see that the child process is defunct by running the following command:

```
$ ps –ef | grep a.out
```

Or, by finding out the pid of the child, and grep-ing the *State* of the process in *status* file corresponding to its entry in */proc* . For example, if the PID is 12273:

```
$ grep State /proc/12273/status
```

shows the status as:

```
State:    Z (zombie)
```

4. How can we run a new program in a process: exec() family of calls.

Fork system call is normally followed by one of the exec family of functions in the child. For example:

```
int s;
if(fork() == 0)
    execlp("ls", "ls", "-la", NULL);
else
    wait(&s);
```

This will create a child process and run "ls –la" command in the child, while the parent waits. The execlp() is one of the simpler functions of exec family that takes the file name as first argument (it searches for the given file in all the directories in the PATH environment variable, see below). Following the file name are the command (ignored by exec) and comma separated list of options. The list must end with a NULL (sometimes 0 is permitted as well).

Note that Linux has a system call execve() that will be called internally by all the functions belonging to the exec-family. You can view the output of strace to confirm this. The output of strace will contain lines similar to the following (It found the ls command under /bin/ls after searching a few directories listed in PATH):

```
[pid  4661] execve("/usr/local/bin/ls", ["ls", "-l"], [/* 62 vars */]) = -1
ENOENT (No such file or directory)
[pid  4661] execve("/usr/sbin/ls", ["ls", "-l"], [/* 62 vars */]) = -1 ENOENT
(No such file or directory)
[pid  4661] execve("/usr/bin/ls", ["ls", "-l"], [/* 62 vars */]) = -1 ENOENT
(No such file or directory)
[pid  4661] execve("/sbin/ls", ["ls", "-l"], [/* 62 vars */]) = -1 ENOENT (No
such file or directory)
[pid  4661] execve("/bin/ls", ["ls", "-l"], [/* 62 vars */]) = 0
```

Homework exercise: Create your own shell that can run: cat, ls, pwd commands.
The following are the steps you/shell should perform:
Compile your program and save as myshell.
When you run myshell it displays a prompt $ and wait for one of the commands that it can run.
You can type one of the following commands: cat, ls, pwd
The shell forks and uses exec to run the command in child process. After the command output is displayed, the shell prints $ in a new line and waits for your next command.
If you enter a command that is different from the above, it should print error: "command not found", display prompt $ and waits for your command. The shell terminates only when you type *exit* or press Ctrl-C.