East West University Bangladesh
Computing Science and Engineering Department

## CSE-325: Operating System, Process Management- Lab 4

## Objectives:

- Work with PS commands at the Linux command line

Process is a running instance of a program. Linux is a multitasking operating system, which means that more than one process can be active at once. Use **ps** command to find out what processes are running on your system. To monitor and control the processes, Linux provides lot of commands such as **ps, kill, killall, nice, renice** and **top** commands.

**1. List Currently Running Processes (ps -ef, ps –aux)**

**ps** command lists down all the process which are currently running in a machine.

Where:

- -e to display all the processes.
- -f to display full format listing.
- -x to display system processes.
- -a to display all users information.

**2. ps auxf/ pstree commands**

There are few ways to view process tree in Linux. The most common one is to use "ps", and add "f" to the option, which stands for forest. There's a program in Linux that's created just for this purpose, which is "pstree".

**Run the two commands and write their difference.**

**3. nice/renice/top commands**

**How can I see priority of a process running in Linux?**

a. You can see the process priority by running "ps" command with "-al" option as shown below. Process priority values range from -20 to 19.

b. The column that starts with "NI" shows the nice value (priority of the process). You can clearly see that most of them has got a '0' priority. A process with the nice value of -20 is considered to be on top of the priority. And a process with nice value of 19 is considered to be low on the priority list.

**nice -10 <command name>** Will set a process with the priority of "10". Running nice command with no options will give you the priority with which the current shell is running.

**Write the difference between two commands**

- nice -10 <command name>
- nice --10 <command name>

c. In order to change the priority of an already running process you can use "renice" command. **renice -4 -p 3423** (this will set the priority of process id no 3423 to -4, which will in turn increase its priority over others)

d. **Normal users can only decrease their process priority**. However root user can increase/decrease all process's priority.

e. You can also see the process priority (nice value) and process status using **"top"** command.

**Process Status:**

```
D: uninterruptible sleep
R: running
S: sleeping
T: traced or stopped
Z: zombie
```

**4. kill/killall**

**What Linux or Unix permissions do when I need to kill a process?**

The kill command sends the specified signal such as kill process to the specified process or process groups. If no signal is specified, the TERM signal is sent. kill command can be internal as part of modern shells built-in function or external located at /bin/kill. Usage and syntax remain similar regardless internal or external kill command.

**Rules are simple:**

a. You can kill all your own process.
b. Only root user can kill system level process.
c. Only root user can kill process started by other users.

kill [singnal] PID
kill -15 PID
kill -9 PID
kill -1 PID

**Search in man kill and write your understanding about signal -15, -9 and -1.**

killall Process-Name-Here
killall -9 firefox-bin [To kill the Firefox web-browser process]

**Write the difference between kill and killall commands**

**5. Process Programming in C**

a. There is an example c program at the end of the **wait manual page** (man wait). Read it, compile it, and run it. The program has also been supplied with lab works (prog0.c).
The program (prog0) demonstrates the use of **fork()** and **waitpid**(). It creates a child process. If no command-line argument is supplied, then the child suspends its execution using **pause**(), to allow the user to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately, using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid**(), and uses the W*() macros described above to analyze the wait status value.

**The following shell session demonstrates the use of the program:**

```
$ ./a.out &
Child PID is 32360
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
```

```
$ kill -TERM 32360
killed by signal 15
```

**Answer the following questions:**

i. Both **exit()** and **_exit()** are used in the program. What's the difference?
ii. Write your understanding about the following line of code:
**`w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);`**

**b. system(), fork(), exec()**

i. Compile and run the following 4 programs (prog1,prog2, prog3, and prog4). Write in your report- what they do? and their differences?

**prog1.c:**

```
1. #include <stdlib.h>
2. #include <stdio.h>
3. int main()
4. {
5.     printf("Running ps with system\n");
6.     system("ps -ax &");
7.     printf("Done.\n");
8.     exit(0);
9. }
```

**prog2.c**

```
1. #include <unistd.h>
2. #include <stdio.h>
3. int main()
4. {
5.     printf("Running ps with execlp\n");
6.     execlp("ps", "ps", "-ax", 0);
7.     printf("Done.\n");
8.     exit(0);
9. }
```

**prog3.c**

```
1. #include <stdio.h>
2. #include <unistd.h>
3. int main(){
4.
5.    pid_t child_p;
6.    printf("Running ps with fork\n");
7.
8.    child_p = fork();
9.
10. execlp("ps", "ps", "-ax", 0);
11. return 0;
12.}
```

**prog4.c**

```
1. #include <unistd.h>
2. #include <stdio.h>
3. int main()
4. {
5.     pid_t pid;
6.     printf("Running ps again with fork\n");
7.     pid = fork();
8.     if ( pid == 0 ) { // in the child, do exec
9.         execlp("ps", "ps", "-ax", 0);
10.    }
11.    else if (pid < 0) // failed to fork
12.    {
13.        printf("fork failed.\n");
14.        exit(1);
15.    }
16.    else // parent
17.    {
18.        wait(NULL);
19.    }
20.    exit(0);
21.}
```

### c. more on fork() and wait()

Compile and run the following program.

**Answer why is the output weird (mixed with the $ prompt)? And fix it with the wait() system call.**

**prog5.c**

```
1. #include <sys/types.h>
2. #include <unistd.h>
3. #include <stdio.h>
4. int main()
5. {
6.     pid_t pid;
7.     char *message;
8.     int n;
9.     printf("fork program starting\n");
10.    pid = fork();
11.    switch(pid)
12.    {
13.    case -1:
14.        perror("fork failed");
15.        exit(1);
16.    case 0:
17.        message = "This is the child";
18.        n = 7;
19.        break;
20.    default:
21.        message = "This is the parent";
22.        n = 3;
23.        break;
24.    }
25.    for(; n > 0; n--) {
26.        puts(message);
27.        sleep(1);
28.    }
29.    exit(0);
30.}
```

## d. zombies and waitpid()

i. Run prog0.c. Play with it, and explain about **WUNTRACED, WCONTINUED, WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED, pause().**

ii. Read the **NOTES** section in the `wait` manual page (`man 2 wait`) to get a clear idea about zombie processes. **And explain- why zombie is not welcomed.**

iii. Compile and run the following small program. This program can leave a zombie process in the system. You can trace the zombie process with command **ps u**

### prog6.c

```
1. /* zombie test. */
2. #include <sys/types.h>
3. #include <unistd.h>
4. #include <stdio.h>
5. int main()
6. {
7.   pid_t pid;
8.   switch(pid = fork())
9.     {
10.          case -1:
11.            perror("fork failed");
12.            exit(1);
13.          case 0:
14.            printf(" CHILD: My PID is %d, My parent's PID is %d\n",
                                                 getpid(), getppid());
15.            exit(0);
16.          default:
17.            printf("PARENT: My PID is %d, My child's PID is %d\n",
                                                       getpid(), pid);
18.            printf("PARENT: I'm now looping...\n");
19.            while(1);
20.      }
21.        exit(0);
   }
```

**Your task:**

a. Write a similar program that leaves 5 zombies.
b. Write the differences between a **zombie process** and an **orphan process**?
c. Correct the above program to avoid zombies with waitpid() system call.
d. Write the difference between exit() and return.
e. Compile and run "my_shell.c" program. Explain your understanding about the program.

**Reference**

a. `man fork, man exec`
b. `man wait, man exit`
c. `man 2 kill`