AROR UNIVERSITY OF ART, ARCHITECTURE, DESIGN & HERITAGE SUKKUR

Operating System

Lab-07

Compiled by: Aurangzeb Magsi

Course Title:
OperatingSystem

Understanding Process in LINUX

1. Process

A computer program which is in execution called process.It is a dynamic entity.It needs resourcesCPU, I/O, allocated when created.

2. Process ID:

A Linux or Unix process is running instance of a program. For example, Firefox is a running process if you are browsing the Internet. Each time you start Firefox browser, the system is automatically assigned a unique process identification number (pid). A pid is automatically assigned to each process when it is created on the system. To find out PID of firefox or httpd process use the following command:

```
pidof httpd
pidof apache2
pidof firefox
```

Ps Command:

The ps command shows the processes we’re running, the process another user is running, or all the processes on the system. E.g. $ps

```
$ ps –ef
```

By default, the ps program shows only processes that maintain a connection with a terminal, a console, a serial line, or a pseudo terminal. Other processes run without needing to communicate with a user on a terminal. These are typically system processes that Linux uses to manage shared resources. We can use ps to see all such processes using the -e option and to get “full” information with –f.

3. Process Creation:

In Linux, process are created by first duplicating the parent process. This is called forking, after the fork system call. In C, you invoke the fork system call with the fork() function.

Fork():

It is a system call that creates a new process under the Linux operating system. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():
fork() returns a negative value, the creation of a child process was unsuccessful.
fork() returns a zero to the newly created child process.
fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer.
Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.

Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that

Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Example#01:

```
#include<stdio.h>
#include<unistd.h>
 int main() {
printf("Before Forking");
fork();
printf("After Forking");
return 0;
}
```

If the call to fork() is executed successfully, Linux will Make two identical copies of address spaces, one for the parent and the other for the child. Both processes will start their execution at the next statement following the fork() call. If we run this program, we might see the following on the screen:

Output: "Before Forking After Forking After Forking"

Here printf() statement after fork() system call executed by parent as well as child process. Both processes start their execution right after the system call fork(). 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#02:

```
#include<stdio.h>
#include<unistd.h>
 int main() {
fork();
printf("Before Forking");
printf("After Forking");
return 0;
}
```

Output: "Before Forking After Forking Before Forking After Forking"

4. Distinguish Parent and Child Process

Consider the following example, which distinguishes the parent from the child.

Example#3

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
void main() {
int p=fork(); //call fork();
if(p==0) {
printf("\n Hi I am child process\n");
}
else

printf("\nHi I am Parent Process\n");
return(0)
}
```

5. getpid() and getppid()

getpid() is use to get process identification which are executing while getppid() is used to get parent process identification. Let take some changes in the above program.

Example#4

```
#include <stdio.h>
#include <unistd.h>
#include < sys/types.h >
void main() {
int p=fork(); //call fork();
if(p==0) {
 printf("\n This is child process with id %d\n", getpid());
}
else
printf("\nThis is Parent Process with id %d\n", getpid());
return(0)
}
```

```
Output: This is Parent Process with id 2412 This is Child Process with id 2413
```

Example#5

```
#include <stdio.h>
#include <unistd.h>
#include < sys/types.h >
void main() {
int p=fork(); //call fork();
if(p==0) {
 printf("\n This is child process with id %d\n", getpid());
printf("\nThis is Parent Process with id %d\n", getppid());
}
else
printf("\nThis is Parent Process with id %d\n", getpid());
return(0)
}
```

```
Output: This is Parent Process with id 2412
        This is Child Process with id 2413
        The Parent id is 1
```

Here, prompt is returned to user after the completion of parent process. It is not waiting for the completion of child process. Because the parent process is already terminated, the getppid() return 1 to overcome this. To resolve this issue we need to use wait() system call.

6. wait():

In order to wait for a child process to terminate, a parent process will just execute a wait() system call. This call will suspend the parent process until any of its child processes terminates, at which time the wait() call returns and the parent process can continue. The prototype for the wait(call is: pid_t wait(int *status);

The return value from wait is the PID of the child process which terminated. The parameter to wait() is a pointer to a location which will receive the child's exit status value when it terminates.

Example#6

```
#include <stdio.h>
#include <unistd.h>
#include < sys/types.h >
void main() {
int p=fork(); //call fork();
if(p==0) {
printf("\n This is child process with id %d\n", getpid());
printf("\n The Parent id is %d\n", getppid());
}
else {
printf("\nHi I am Parent Process with id %d\n", getpid());
wait(0); //parent wait until child process completed }
}
```

```
Output: This is Parent Process with id 2534
        This is Child Process with id 2535
        The Parent id is 2534 exit()
```

When a process terminates it executes an exit() system call, either directly in its own code, or indirectly via library code. The prototype for the exit() call is:

```
#include <stdlib.h>
void exit(int status);
```

The exit() call has no return value as the process that calls it terminates and so couldn't receive a value anyway. Notice, however, that exit() does take a parameter value - status. As well as causing a waiting parent process to resume execution, exit() also returns the status parameter value to the parent process via the location pointed to by the wait()parameter.

Example#7

```
#include <stdio.h>
#include <unistd.h>
#include < sys/types.h >
void main() {
int p=fork(); //call fork();
if(p==0) {
printf("\n Hi I am child process with id %d\n", getpid());
exit(0);
printf("\n My Parent id is %d\n", getppid());
}
else { printf("\nHi I am Parent Process with id %d\n", getpid());

}}
Output: This is Parent Process with id 2751
        This is Child Process with id 2752
```

You can see the output, as the line My Parents id is not printed because the running process is terminated directly from program after reading the exit(0).

7. sleep():

sleep() makes the calling thread sleep until seconds have elapsed or a signal arrives which is not ignored. The prototype for the sleep() call is:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Example#8

```
#include <stdio.h>
#include <unistd.h>
#include < sys/types.h >
void main() {
int p=fork(); //call fork();
if(p==0) {
 printf("\n This is child process with id %d\n", getpid());
sleep(5); //sleep for 5 seconds
printf("\n The Parent id is %d\n", getppid());
exit(0);
}
else
{ wait(0);
printf("\nThis is Parent Processwith id %d\n", getpid());
}
printf(("\n The End");
}
Output: This is Parent Process with id 2751 This is Child Process with id 2752//after this line sleep for 4
The Parent id is 2751 sec then display next line
```