LAB # 6

Create a child process “ls” command in linux

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

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
     /* parent will wait for the child to complete */
     wait(NULL);
     printf("Child Complete");
   }

   return 0;
}
```

Create a child process open MS Paint in Windows // Download codeblock for this

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

   /* allocate memory */
   ZeroMemory(&si, sizeof(si));
   si.cb = sizeof(si);
   ZeroMemory(&pi, sizeof(pi));

   /* create child process */
   if (!CreateProcess(NULL, /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
    NULL, /* don't inherit process handle */
    NULL, /* don't inherit thread handle */
    FALSE, /* disable handle inheritance */
    0, /* no creation flags */
    NULL, /* use parent's environment block */
    NULL, /* use parent's existing directory */
    &si,
    &pi))
   {
     fprintf(stderr, "Create Process Failed");
     return -1;
   }
   /* parent will wait for the child to complete */
   WaitForSingleObject(pi.hProcess, INFINITE);
   printf("Child Complete");

   /* close handles */
   CloseHandle(pi.hProcess);
   CloseHandle(pi.hThread);
}
```

Orphan Process:

To create a normal child (duplicate) process (no orphan process in this case)

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
     pid_t p;
     p=fork();
     if(p==0) //child
     {
         printf("I am child having PID %d\n",getpid());
         printf("My parent PID is %d\n",getppid());
     }
     else //parent
     {
         printf("I am parent having PID %d\n",getpid());
         printf("My child PID is %d\n",p);
     }
 }
```

Output:

```
I am parent having PID 130
I am child having PID 131
My child PID is 131
My parent PID is 130
```

Orphan Process:

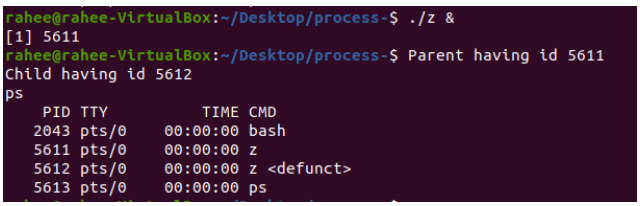
```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
p=fork();
if(p==0)
   {
sleep(5); //child goes to sleep and in the mean time parent terminates
      printf("I am child having PID %d\n",getpid());
      printf("My parent PID is %d\n",getppid());
   }
     else
   {
         printf("I am parent having PID %d\n",getpid());
         printf("My child PID is %d\n",p);
   }
}
```

```
rahee@rahee-VirtualBox:~/Desktop/process-$ ./proc
bash: ./proc: No such file or directory
rahee@rahee-VirtualBox:~/Desktop/process-$ ./a.out
I am parent having PID 5382
My child PID is 5383
rahee@rahee-VirtualBox:~/Desktop/process-$ I am child having PID 5383
My parent PID is 1330
```

Zombie Process

```
//zombie.c
#include<stdio.h>
#include<unistd.h>
int main()
{
   pid_t t;
   t=fork();
   if(t==0)
   {
       printf("Child having id %d\n",getpid());
    }
    else
    {
        printf("Parent having id %d\n",getpid());
        sleep(15); // Parent sleeps. Run the ps command during this time
    }
}
```

Output



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.

Let us take an example:
Example # 1

```
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:

```
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 # 2

```
int main()
{
fork();
printf(“Before Forking”);
printf(“After Forking”);
return 0;
}
Before Forking
After Forking
Before Forking
After Forking
```

Task 1:

Write a program to distinguish in between parent and child processes

Hints: getpid() is used to retrieve the identifier of a process, while getppid() is used to retrieve the identifier of parent process.

Task 2:

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);
```

Write a program where parent waits for child to terminate.

Task 3:

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 <std1ib.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.

Task 4:

Write a program to demonstrate the delay using **sleep()** system call. Sleep makes the calling thread sleep until *seconds* have elapsed or a signal arrives which is not ignored. Return Value: Zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler.

The prototype for the sleep() call is:

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