LAB # 3

BACKGROUND PROCESS AND SYSTEM CALL ‘Fork ( )’

OBJECTIVE

Study the features of Background Process.

THEORY

First compile the above program using command ‘gcc program name.cc’ , then execute it through the command ‘./a.out &’. The & will make it a background process, which is terminated only if the process ends, or is interrupted with a DEL key. While the said program is running in the background, open a file with any editor and type something, save it and execute it. For reference consider Program 1.

**Process Identification**

There is a function – *getpid( ) –* that enables us to get the identification number of a process. For reference consider Program 2 and Program 3.

The *pid* value will depend on the number of processes already running. It will always be unique. This *pid*  cannot be changed although it may be reused once the process no longer exists.

Run the program 3 twice. Both times as a background process, i.e suffix it with an ampersand (&). Once both are running as a background processes key the ps-a command and you will see that memory contains these two processes.

**Parent And Child**

A process in UNIX is not a standalone. In the same way as a human being or animal is born from another, a process to has to come from the womb of another process. This results in a parent-child relationship existing between processes. For reference consider Program 4.

Program 4 will give the ID number of its parents.

When we boot the system, a special process called the swapper or scheduler is created with a PID of 0. The swapper manages memory allocation for processes and influences CPU allocation. The swapper in turn creates three: the process dispatcher, vhand, and bdflush with ID numbers 1,2 and 3 respectively.

This is done by executing the file init which exists in the etc sub-directory. The process dispatcher now gives birth to the shell. From now on all processes initiated by us are children of the shell and in turn descendents of the process dispatcher. This gives rise to a tree-like structure, with ADAM as the swapper.

UNIX keeps track of all processes in an internal data structure called the process table. A listing of the process table can be got using the ps-el command.

**The 'fork( )'**

Processes initiated by us can also create children in the same manner as the swapper and the process dispatcher did. These children processes are created using the fork ( ) function. It is by forking processes that we can exploit the multitasking capability of UNIX. For reference consider Program 5.

The child process begins from the fork( ). All the statements after the call to fork( ) will be executed twice. Once by the parent process and once by the child process. But had there been any statements before the fork( ) they would have been only executed by the parent process. For reference consider Program 6.

# Example programs

## Program 1

main()

{

int pid;

pid = getpid( );

printf(“Process ID is %d \n”,pid);

}

## Program 2

main()

{

int ppid;

ppid = getppid( );

printf(“Parent Process ID is %d\n”,ppid);

}

## Program 3

main()

{

fork( );

printf(“Hello World\n”);

}

## Program 4

main()

{

int pid;

pid = fork( );

if(pid>0)

printf(“Parent process PID is %d\n”,pid);

}

## Program 5

main()

{

int pid

pid = fork( );

if (pid == 0)

printf(“Child Process\n”);

}

## Program 6

## main( )

{

int pid;

pid=fork( );

if(pid==0)

{

printf(“I am the child, my process ID is %d\n”,getpid( ));

printf(The child’s parent process ID is %d\n”,getppid( ));

}

else

{

printf(“I am the parent, my process ID is %d\n”,getpid());

printf(“The parents parent process ID is %d\n,getppid( ));

}

}

# Exercise

1. Write few lines about given Program and what is the output of same program?

#include<stdio.h>

main(int arc,char\*ar[])

{

int pid; char s[100];

pid=fork();

if(pid<0)

printf("error");

else if(pid>0)

{

wait(NULL);

printf("\n Parent Process:\n");

printf("\n\tParent Process id:%d\t\n",getpid());

execlp("cat","cat",ar[1],(char\*)0);

error("can’t execute cat %s,",ar[1]);

}

else

{

printf("\nChild process:");

printf("\n\tChildprocess parent id:\t %d",getppid());

sprintf(s,"\n\tChild process id :\t%d",getpid());

write(1,s,strlen(s));

printf(" ");

printf(" ");

printf(" ");

execvp(ar[2],&ar[2]);

error("can’t execute %s",ar[2]);

}

}