**Task#1:**

* fork() function command creates another process, i.e. child, which is the duplicate of the parent process. Parent process is the ne which called this fork() function. fork() function returns a PID (process ID) of the child in the parent process.
* exec() function command replaces the current program space with some other one as provided by the user. Once it is called, the code lines after the exec() are totally replaced by a set of other instructions as indicated by the user.

**Task#2:**

**Program#1:**

int main()

{

int pid;

pid = getpid();

printf(“Process ID is %d\n”, pid);

return 0;

}

pid variable will have a value that is the process id of the current process.

printf will print the message and then the variable pid on the terminal/console.

%d id to identify the type of variable that is to be printed! Here it is double.

**Program#2:**

int main()

{

long i;

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

for(i=0; i<=400;i++)

{

printf(“i is %d\n”, i);

}

return 0;

}

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

It will display ‘Process ID is’ in one line and the pid obtained from getpid() in the next line.

The for-loop will display the counter ‘i’ value 400 times, each in a next line, like, ‘i is 0’ then in next line, ‘i is 1’ upto 399.

**Program#3:**

int main()

{

int ppid;

ppid = getppid();

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

return 0;

}

Here the variable ppid will have a value (process id) of the process which created the current process where getppid() is called. Simply, getppid() returns the value of the parent process; the process which has created the current process.

**Program#4:**

int main()

{

fork();

printf(Hello World\n”);

return 0;

}

fork() commands creates a child which is a duplicate of the current process. Here, all the code after fork() function is duplicated to a child process. This program prints “Hello World” two times. One of the parent process, second of the child process.

When I run this program, it printed parent process’s “Hello World” first and then the child’s one.

**Prgoram#5:**

int main()

{

printf(“This is to demonstrate the fork() \n”);

fork();

return 0;

}

First the line saying “This is to demonstrate the fork()” is printed on the terminal.

Then fork() call will create another child, with the duplicated code after the fork() call.

The parent is executed first (I judged it when I run the next programs) and then the child and afterwards program terminates. As there is not print command after the fork() call, we cannot judge in which way these executed.

**Program#6:**

int main()

{

fork();

fork();

printf(“Hello World\n”);

return 0;

}

Two consecutive fork() calls will create a hierarchy of childs. “Hello World” will be printed in a tree form. First the parent is executed. After that the child of that parent is executed and so on.

So in this case, “Hello World” is printed 4 times, each time in a next line.

The number of times the “Hello World” is printed can also be calculated with a formula, which is 2n where n is the number of consecutive fork() function calls.

**Program#7:**

int main()

{

fork();

printf(“The PID is %d\n”, getpid());

return 0;

}

fork() creates anther copy of the parent process’s code after the fork() call. It prints the following lines twice:

The PID is

5690(let say the process id)

Firstly it will be the process ID of the parent and then the process ID of the child. The PID of child is 1. This shows that the difference between the parent’s and child’s ID is 1.

**Program#8:**

int main()

{

int pid;

pid = fork();

if(pid > 0)

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

return 0;

}

Once a fork() call is made, it returns some value. This value is 0 in case the pid (variable) is in the child’s process. While in the parent’s process, the value of pid is the ID of child which is not 0.

So, the print message will only be printed once it is being executed in parent’s code. Because only then, the condition is satisfied!

**Program#9:**

int main()

{

int pid;

pid = fork();

if(pid == 0)

printf(“Child Process\n”);

return 0;

}

As in above program, in this program, the print line will only get printed once it is being executed in the child’s code. Because only then it satisfies the condition!

**Program#10:**

int 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 parent process ID is %d\n”, getppid());

}

return 0;

}

In this program, the else part executes first. This shows that whenever fork() call is made, first the parent program is given the CPU resource.

First the pid of the parent is printed and then ppid (parent pid).

After that, pid and ppid of child is printed.

**Task#3:**

**Program#11:**

#include <unistd.h>

int main()

{

printf(“Before exec my id is %d \n”, getpid());

printf(“My parent process id is %d \n”, getppid());

printf(“exec strats\n”);

execl(“Program#12”, “ex2”, (char\*)0); //set path

printf(“This will not print\n”);

return 0;

}

**Program#12:**

int main()

{

printf(“After the exec my process id is %d\n”, getpid());

printf(“My parent process id is %d\n”, getppid());

printf(“exec ends\n”);

return 0;

}

First both the programs are compiled and object files are made. Then object file of program#11 is run. Program#11 is run upto line printf(“exec strats\n”); and then the rest of the program code is replaced by the code of Program#12. Then the following lines execute and program terminates:

printf(“After the exec my process id is %d\n”, getpid());

printf(“My parent process id is %d\n”, getppid());

printf(“exec ends\n”);

The pid and ppid values are the same each time.