Lab No. 06
Using Fork, Exec, Wait & Exit System-Calls for Creating Chile Processes

Objective:

This lab describes how a program can create, terminate, and control children processes using system calls.

Activity Outcomes:

On completion of this lab students will be able to:

- Write programs that uses process creation system call fork ( )
- Use exec system call
- Use wait system call
- Use sleep system call

Instructor Notes

As pre-lab activity, read the content from the following (or some other) internet source: https://www.geeksforgeeks.org

1) Useful Concepts

Process Creation Concepts

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the programwithin its own address space and executes it independently of the other copies. Processes are organized hierarchically. Each process has a **parent process**, which explicitly arranged to create it. The processes created by a given parent are called its **child processes**. A child inherits many of its attributes from the parent process.

A **process ID** number names each process. A unique process ID is allocated to each process when it is created. The lifetime of a process ends when its termination is reported to its parent process; at thattime, all of the process resources, including its process ID, are freed. To monitor the state of your processes under Unix use the **ps** command:

```
ps [-option]
```

Process Identification

The **pid_t** data type represents process IDs which is basically a signed integer type (int). You can getthe process ID of a process by calling:

getpid() - returns the process ID of the parent of the current process (the parent process ID).

getppid() - returns the process ID of the parent of the current process (the parent process ID). Your program should include the header files ‘unistd.h’ and ‘sys/types.h’ to use these functions.

fork () | Process Creation

Processes are created with the fork () system call (so the operation of creating a new process is sometimes called forking a process). The child process created by fork is a copy of the original parent process, except that it has its own process ID. After forking a child process, both the parent and child processes continue to execute normally. If you want your program to wait for a child process to finish executing before continuing, you must do this explicitly after the fork operation, by calling wait () or waitpid (). These functions give you limited information about why the child terminated--for example, its exit status code.

A newly forked child process continues to execute the same program as its parent process, at the point where the fork call returns. You can use the return value from fork to tell whether the program is running in the parent process or the child process.

When a child process terminates, its death is communicated to its parent so that the parent may take some appropriate action. If the fork () operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process. If process creation failed, fork returns a value of -1 in the parent process and no child is created.

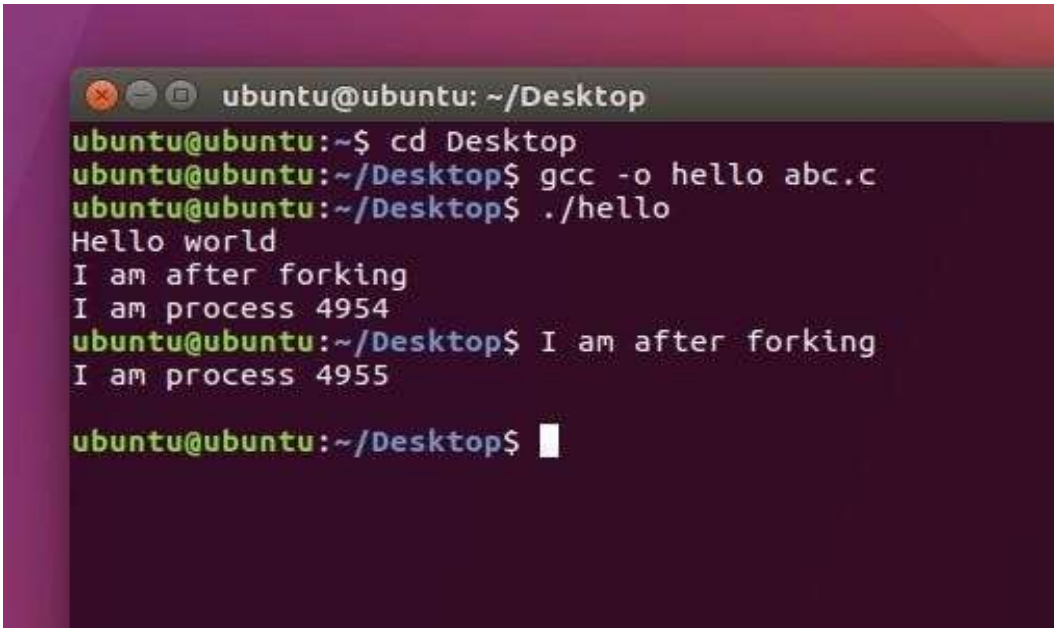
Example 1 – Single fork() :

```
#include <stdio.h>
#include <unistd.h> /* contains fork
prototype */ int main(void)
{
printf("Hello orld!\n"); fork( );
      printf("I am after forking\n");
      printf("\tI am process %d.\n",
getpid( ));
}
```

Write this program and run using the following commands:

gedit hello.c //write the above C code and close the editor gcc -o hello hello.c /./compile using built-in GNU C compiler ./hello//run the code

Output:



When this program is executed, it first prints Hello World! . When the fork is executed, an identical process called the child is created. Then both the parent and the child process begin execution at thenext statement.

Note that:

- There is no guarantee which process will print I am a process first.
- The child process begins execution at the statement immediately after the fork, not at the beginning of the program.
- A parent process can be distinguished from the child process by examining the return value of the fork call. Fork returns a zero to the child process and the process id of the child process to the parent.

Example 2:

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
int pid;
printf("Hello World!\n");
printf("I am the parent process and pid is : %d
.\n",getpid()); printf("Here i am before use of
forking\n");
pid = fork();
printf("Here I am just after forking\n"); if (pid
```

```
== 0)
printf("I am the child process and pid is
:%d.\n",getpid()); else
printf("I am the parent process and pid is: %d
.\n",getpid());
}
```

Executing the above code will give the following output:



This programs give Ids of both parent and child process.The above output shows that parent process is executed first and then child process is executed.

wait ( ) | Process Completion

A process wait ( ) for a child process to terminate or stop, and determine its status. These functions are declared in the header file "sys/wait.h"

wait ( ): A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction.

Child process may terminate due to any of these:

It calls exit( );

- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

Wait ( ) will force a parent process to wait for a child process to stop or terminate. Wait ( ) return the pid of the child or -1 for an error.

exit ( ): Exit () terminates the process which calls this function and returns the exit status value. Both UNIXand C (forked) programs can read the exit status value.By convention, **a status of 0** means normal termination. Any other value indicates an error orunusual occurrence.

Example:

// C program to demonstrate working of wait() and exit()#include<stdio.h>

```
#include<stdlib.h>

#include<sys/wait.h> #include<unistd.h> int main()
```

```
{
pid_t cpid;

if (fork()== 0)

exit(0);              /* terminate child – exit (0)
means normal termination */ else
cpid = wait(NULL); /* parent will wait until child
terminates */ printf("Parent pid = %d\n",
getpid());
printf("Child pid = %d\n", cpid); return 0;
}
```

Output:



Sleep ( )

A process may suspend for a period of time using the sleep () command:

Orphan Process: When a parent dies before its child, the child is automatically adopted by the original "init" process whose PID is 1.

Zombie Process: A process that terminates cannot leave the system until its parent accepts its return code. If its parent process is already dead, it'll already have been adopted by the "init" process, which always accepts its children's return codes (orphan). However, if a process's parent is alive but never executes a wait ( ), the process's return code will never be accepted and the process will remain a zombie.

Ecec ( )

The exec family of functions replaces the current running process with a new process. It can beused to run a C program by using another C program. It comes under the header file **unistd.h.** The program that the process is executing is called its process image. Starting execution of a new program causes the process to forget all about its previous process image; when the new program exits, the process exits too, instead of returning to the previous process image.

There are a lot of exec functions included in exec family of functions, for executing a file as a process image e.g. execv(), execvp() etc. You can use these functions to make a child process execute a new program after it has been forked. The functions in this family differ in how youspecify the arguments, but otherwise they all do the same thing.

Execv () : Using this command, the created child process does not have to run the same program as the parent process does. The exec type system calls allow a process to run any program files, which include a binary executable or a shell script .

Syntax: int execv (const char *file, char *const argv[]);

file: points to the file name associated with the file being executed.
argv: is a null terminated array of character pointers.

Example: Let us see a small example to show how to use execv () function in C. We will have two .C files: example.c and hello.c and we will replace the example.c with hello.c by calling execv() function in example.c

example.c

CODE:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of example.c = %d\n", getpid());
    char *args[] = {"Hello", "C", "Programming", NULL};
    execv("./hello", args);
    printf("Back to example.c");
    return 0;
}
```

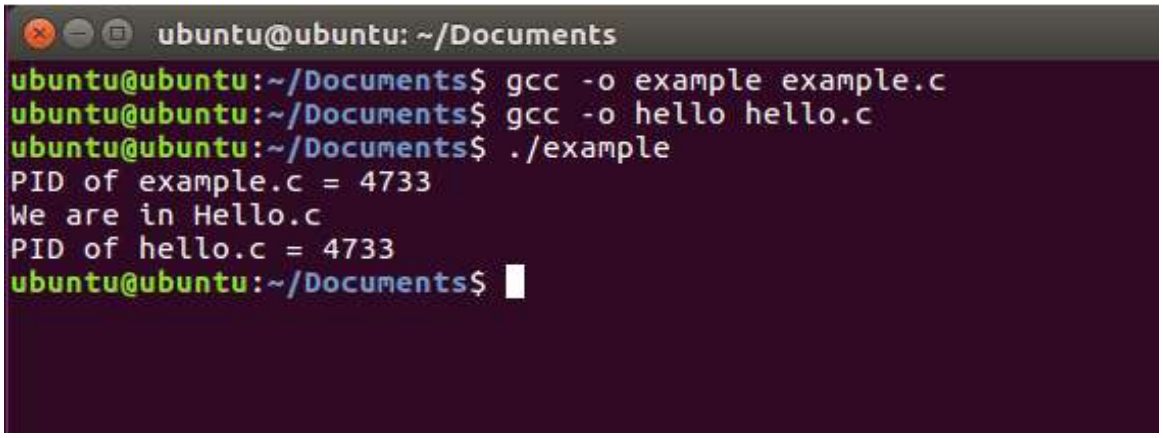
hello.c

CODE:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in Hello.c\n");
    printf("PID of hello.c = %d\n", getpid());
    return 0;
}
```

Output:

```
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
```



2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*25*	*Medium*	*CLO-7*
2	*25*	*Medium*	*CLO-7*

Activity 1:

In this activity, you are required to perform tasks given below:

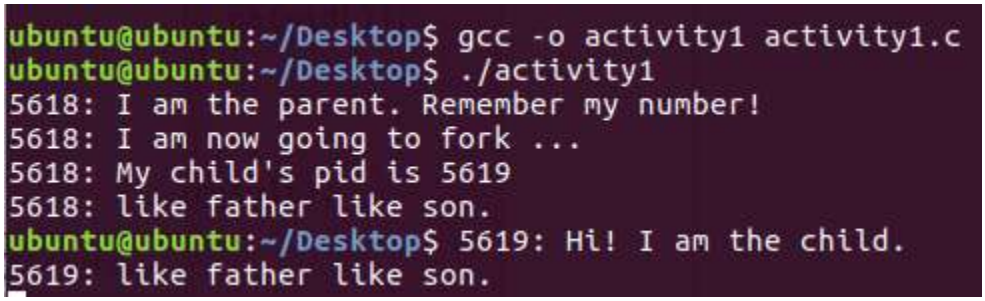
- ***Print something and Check id of the parent process***
- ***Create a child process and print child process id in parent process***
- ***Create a child process and print child process id in child process***

Solution:

```
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
int forkresult;
printf("%d: I am the parent. Remember my number!\n", getpid());
printf("%d: I am now going to fork ... \n", getpid());
forkresult = fork();
if (forkresult != 0)
```

```
{
/* the parent will execute this code */
printf("%d: My child's pid is %d\n", getpid(), forkresult);
}
else /* forkresult == 0 */
{
 /* the child will execute this code */ printf("%d: Hi! I am the
child.\n", getpid());
}
printf("%d: like father like son. \n", getpid()); return 0;
}
```

Out-put



Activity 2:

Create a process and make it an orphan.

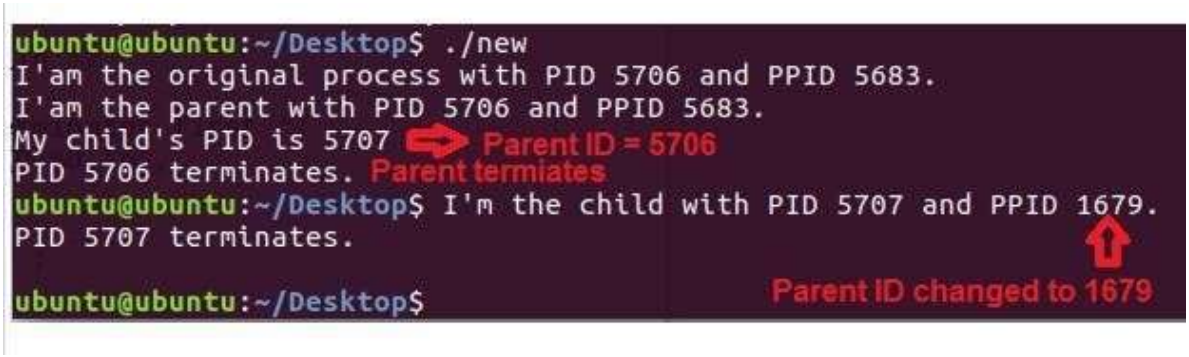
Hint: To, illustrate this insert a sleep statement into the child's code. This ensured that the parent process terminated before its child.

Solution:

```
#include <stdio.h>
intmain()
{
int pid ;
printf("I'am the original process with PID %d and PPID %d.\n",
getpid(), getppid()) ;
pid = fork ( ) ; /* Duplicate. Child and parent continue from here
*/
if ( pid != 0 ) /* pid is non-zero,so I must be the parent*/
{
printf("I'am the parent with PID %d and PPID %d.\n", getpid(),
getppid()) ;
printf("My child's PID is %d\n", pid ) ;
```

```
}
else /* pid is zero, so I must be the child */
{
sleep(4); /* make sure that the parent terminates first */
printf("I'm the child with PID %d and PPID %d.\n", getpid(),
getppid()) ;
}
printf ("PID %d terminates.\n", getpid()) ;
return 0;
}
```

Output:



Activity 3:

Write a C/C++ program in which a parent process creates a child process using a fork() system call. The child process takes your age as input and parent process prints the age.

Solution:

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main()
{
int i;
pid_t p=fork();
if (p==0)
        {
        printf("%d: I am the child\n", getpid());
        printf("enter your age" );
        scanf("%d",&i);
        exit(i);
        }
else
{
wait(&i);
printf("%d: Hello I am parent \n", getpid());
printf("%d is your age", i/256);
}
return 0;
}
```

Out-put:

```
ubuntu@ubuntu:~/Desktop$ ./act
5941: I am the child
enter your age4
5940: Hello I am parent
4 is your ageubuntu@ubuntu:~/Desktop$
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write a C++ program that creates an array of size 1000 and populates it with random integers between 1 and 100. Now, it creates two child processes. The first child process finds how many prime numbers are there among first 500 number while the second child process finds the number of prime numbers among the remaining 500 numbers.

Task 2:

Write a C/C++ program that:

- Creates an array of 20 integers
- Uses fork() to create a child process
- Child process prints all even numbers
- Parent process prints all odd numbers

Task 3:

Write a program where:

- A parent creates one child process
- Child prints numbers from 1 to 5 with a delay using sleep()
- Parent uses wait() and prints a message only after the child finishes execution

Lab Exercise and Summary

Summary should cover Introduction, Procedure, Data Analysis and Evaluation

LABORATORY SKILLS ASSESSMENT (Psychomotor)
Total Marks: 100

(Max Marks)	Level 1 0% ≤ S < 50%	Level 2 50% ≤ S< 70%	Level 3 70% ≤ S< G0%	Level 4 G0%≤ S ≤100%	Score (S)
Procedural Awareness (20)	Selects inappropriate Linux commands, shell scripting techniques, or process management methods.	Selects and applies partially appropriate Linux commands and techniques	Selects and applies considerably appropriate Linux commands and techniques.	Selects and applies completely appropriate Linux commands and techniques	
Practical Implementation (20)	Makes major critical errors in executing Linux commands, scripting, and system processes.	Makes numerous critical errors in executing commands and process management.	Makes minor non-critical errors in executing Linux commands and system operations.	Executes Linux commands and manages processes correctly with no errors.	
Process Management and Shell Scripting (20)	Program logic contains major errors with incorrect or contradictory script flow.	Program logic has some errors with occasional contradictions in process execution.	Program logic is mostly correct but may contain occasional redundancy or minor errors.	Program logic is completely correct with no contradictions or redundant processes.	
Syntax Correctness and Results (20)	Program does not follow proper syntax for Linux commands and shell scripting, leading to incorrect outputs	Program partially follows proper syntax, producing correct results for few inputs.	Program adequately follows proper syntax, producing correct results for most inputs.	Program fully follows proper syntax, producing accurate results for all inputs.	
Use of OS Tools (10)	Uses OS tools (like terminal, process manager) with limited competence.	Uses OS tools with some competence.	Uses OS tools with considerable competence.	Uses OS tools proficiently with a high degree of competence.	
Safety (10)	Requires constant reminders to follow system safety procedures (e.g., file permissions, process handling).	Requires some reminders to follow system safety procedures.	Follows system safety procedures with minimal reminders.	Routinely follows system safety procedures.	
				Marks Obtained	

LABORATORY SKILLS ASSESSMENT (Affective)
Total Marks: 40

Marks)	Level 1 0% ≤ S < 50%	Level 2 50% ≤ S < 70%	Level 3 70% ≤ S < G0%	Level 4 G0% ≤ S ≤ 100%	Score (S)
Attitude s Engagement (5)	Shows little interest in lab activities; does not participate actively.	Participates occasionally but lacks enthusiasm and consistency.	Engages actively in most lab activities with interest.	Highly motivated, participates enthusiastically, and shows a proactive approach	
Responsibility s Punctuality (5)	Frequently misses deadlines and is often late to lab sessions.	Occasionally late or misses deadlines but tries to catch up.	Submits work on time and attends lab sessions regularly.	Always punctual, meets deadlines, and takes full responsibility for assigned tasks.	
Collaboration s Teamwork (10)	Rarely collaborates, struggles to work in a team, and does not contribute effectively.	Works with team members occasionally but struggles with communication.	Cooperates well, contributes effectively, and maintains professional interactions.	Actively engages in teamwork, supports peers, and demonstrates excellent collaboration.	
Communicatio n s Presentation Skills (10)	Struggles to explain concepts, unclear verbal/written communication.	Communicates ideas with some clarity but lacks confidence or coherence.	Presents ideas effectively with minor issues in clarity or structure.	Communicates clearly, confidently, and effectively in all aspects of lab work.	
Report Quality (10)	Report contains many errors.	Report is somewhat organized with some spelling or grammatical errors.	Report is well organized and cohesive but contains some grammatical errors.	Report is well organized and cohesive and contains no grammatical errors. Presentation seems polished.	
				Marks Obtained	