

## Statement of Purpose:

This lab describes how a program can create, terminate, and control processes using system calls. We will start with the some basic process creation commands and later will execute complex process termination, and wait system calls. Moreover, you will learn how to create orphans and zombie processes and how to replace one process with another during its execution using `exec()`.

## Activity Outcomes:

This lab teaches you the following topics:

- Process creation system call `fork()`
- `Exec` system call
- `Wait` system call
- `Sleep` system call

## Instructor Note:

As pre-lab activity, introduce students to basic process creation using `fork` and `exec`.

## 1) Stage J (Journey)

### 1. 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 program within 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 that time, 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]**

Used without options this produces a list of all the processes owned by you and associated with your terminal.

These are some of the column headings displayed by the different versions of this command.

PID	SZ(size in Kb)	TTY(controlling terminal)	TIME(used by CPU)	COMMAND
-----	----------------	---------------------------	-------------------	---------

This PID is the process ID which you will be using to identify parent and child processes.

#### **Process Identification:**

The **pid\_t** data type represents process IDs which is basically a signed integer type (int). You can get the 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.

Function: `pid_t getpid (void)`

The `getpid()` function returns the **process ID** of the current process.

Function: `pid_t getppid (void)`

The `getppid()` function returns the **process ID of the parent** of the current process.

## 1.1 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 World!\n");
    fork( );
    printf("I am after forking\n");
    printf("\tI am process %d.\n", getpid( ));
}
```

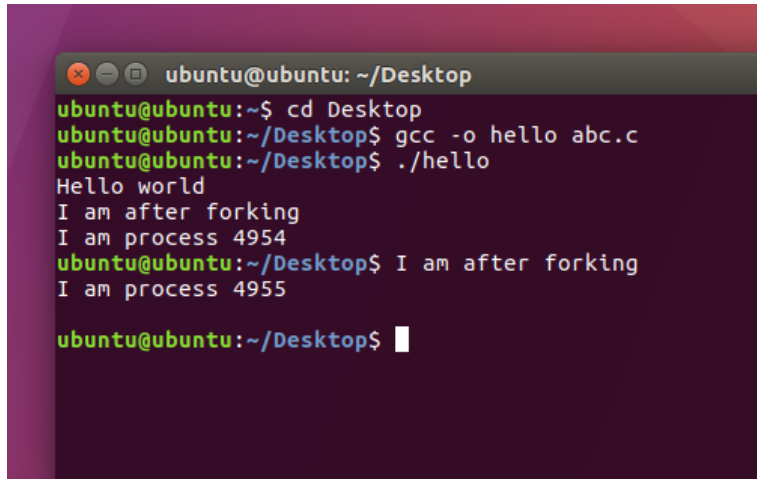
Write this program and run using the following commands:

`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:**



```
ubuntu@ubuntu: ~/Desktop
ubuntu@ubuntu:~$ cd Desktop
ubuntu@ubuntu:~/Desktop$ gcc -o hello abc.c
ubuntu@ubuntu:~/Desktop$ ./hello
Hello world
I am after forking
I am process 4954
ubuntu@ubuntu:~/Desktop$ I am after forking
I am process 4955
ubuntu@ubuntu:~/Desktop$
```

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 the next statement.

### **Note that:**

1. There is no guarantee which process will print I am a process first.
2. The child process begins execution at the statement immediately after the fork, not at the beginning of the program.
3. 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:



```

ubuntu@ubuntu:~/Desktop$ gedit fork1.c
ubuntu@ubuntu:~/Desktop$ gcc -o fork1 fork1.c
ubuntu@ubuntu:~/Desktop$ ./fork1
Hello World!
I am the parent process and pid is : 5292 .
Here i am before use of forking
Here I am just after forking
I am the parent process and pid is: 5292 .
ubuntu@ubuntu:~/Desktop$ Here I am just after forking
I am the child process and pid is :5293.

```

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.

### Example 3 – Multiple Fork() :

```

#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
printf("Here I am just before first forking statement\n");
fork();
printf("Here I am just after first forking statement\n");
fork();
printf("Here I am just after second forking statement\n");
printf("\t\tHello World from process %d!\n", getpid());
}

```

### Output:

```

ubuntu@ubuntu:~/Desktop$ gedit fork2.c
ubuntu@ubuntu:~/Desktop$ gcc -o fork2 fork2.c
ubuntu@ubuntu:~/Desktop$ ./fork2
Here I am just before first forking statement
Here I am just after first forking statement
Here I am just after second forking statement
Hello World from process 5217!
ubuntu@ubuntu:~/Desktop$ Here I am just after second forking statement
Hello World from process 5219!
Here I am just after first forking statement
Here I am just after second forking statement
Hello World from process 5218!
Here I am just after second forking statement
Hello World from process 5220!

```

#### Example 4:

##### Using fork ( ) inside a function

Predict the output and verify after executing the following code

```


#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}

```

##### Output:

```
8 #include <stdio.h>
9 #include <sys/types.h>
10 #include <unistd.h>
11 void forkexample()
12 {
13     // child process because return value zero
14     if (fork() == 0)
15         printf("Hello from Child!\n");
16
17     // parent process because return value non-zero.
18     else
19         printf("Hello from Parent!\n");
20 }
21 int main()
22 {
23     forkexample();
24     return 0;
25 }
26
```



## 1.2 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**"

### 1. 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.

## **2. Exit ():**

Exit () terminates the process which calls this function and returns the exit status value. Both UNIX and C (forked) programs can read the exit status value.

By convention, a **status of 0** means normal termination. Any other value indicates an error or unusual occurrence.

### **Example 1:**

```
// 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 */

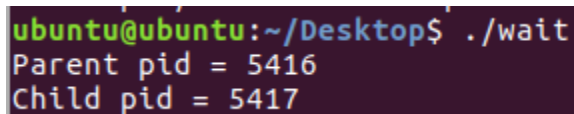
    printf("Parent pid = %d\n", getpid());

    printf("Child pid = %d\n", cpid);

    return 0;

}
```

### **Output:**

A terminal window with a dark background. The prompt is 'ubuntu@ubuntu:~/Desktop\$'. The command './wait' has been executed. The output shows 'Parent pid = 5416' on the first line and 'Child pid = 5417' on the second line.

```
ubuntu@ubuntu:~/Desktop$ ./wait
Parent pid = 5416
Child pid = 5417
```

### **Example 2:**

```
// C program to demonstrate working of wait()
```

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



```

#include<sys/wait.h>

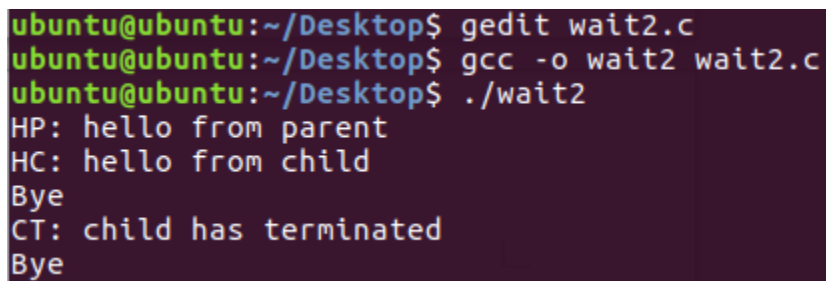
#include<unistd.h>

int main()
{
    if (fork() == 0)
        printf("HC: hello from child\n");
    else {
        printf("HP: hello from parent\n");
        wait(NULL); /*waiting till child terminates
        printf("CT: child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}

```

#### **Output:**



```

ubuntu@ubuntu:~/Desktop$ gedit wait2.c
ubuntu@ubuntu:~/Desktop$ gcc -o wait2 wait2.c
ubuntu@ubuntu:~/Desktop$ ./wait2
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye

```

### **3. 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*.

**Creation of orphan and zombie is left as an exercise for students in activities at the end of lab.**

#### **4. Exec ( ):**

The exec family of functions replaces the current running process with a new process. It can be used 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 you specify 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 1:**

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:

```
#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:

```
#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
```

```
ubuntu@ubuntu: ~/Documents
ubuntu@ubuntu:~/Documents$ gcc -o example example.c
ubuntu@ubuntu:~/Documents$ gcc -o hello hello.c
ubuntu@ubuntu:~/Documents$ ./example
PID of example.c = 4733
We are in Hello.c
PID of hello.c = 4733
ubuntu@ubuntu:~/Documents$
```

## 2) Stage a1 (apply)

### Lab Activities:

#### Activity 1:

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

1. Print something and Check id of the parent process
2. Create a child process and print child process id in parent process
3. Create a child process and print child process id in child process

#### Activity 2:

##### 1. 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.

##### Steps to create an orphan process:

1. print something and get its pid and ppid
2. create a child process
3. Now as a parent process print parent id and id of child process
4. Make child sleep for 5 seconds
5. Now while child is sleeping parent will terminate. Print parent id of child to make sure it is orphaned (PPID has been changed)

#### Activity 3:

##### Create a process and make it a Zombie.

1. Execute fork to create a child

2. In parent process (using if statement) Create an infinite loop so that it never terminates and never executes wait ()
  3. Make parent sleep for 100 sec
  4. Terminate child process exit using exit.
- Now this child is a zombie because no parent is waiting for him

To view status of processes, use the following commands on command line:

1. Execute the c code in background using ./abc &
2. View process status using 'ps -lf'
3. You will see zombie process with state 'Z' in STAT column
4. Get parent id and kill parent process using "kill (parent id)" command
5. again execute 'ps -lf'
6. Note that Zombie is gone now

### Activity 4:

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.

### Activity 5:

1. Write a C/C++ program that asks user to enter his name and his university name. Within the same program, execute another program that asks the user to enter his degree name and department name.
2. Hint: write 2 separate programs and execute using execv()

## 3) Stage V (verify)

### Home Activities:

Practice fork, exec, sleep and wait at home.