I. EXAMPLE OF SINGLE FORK

(fork1.c)

SOURCE CODE

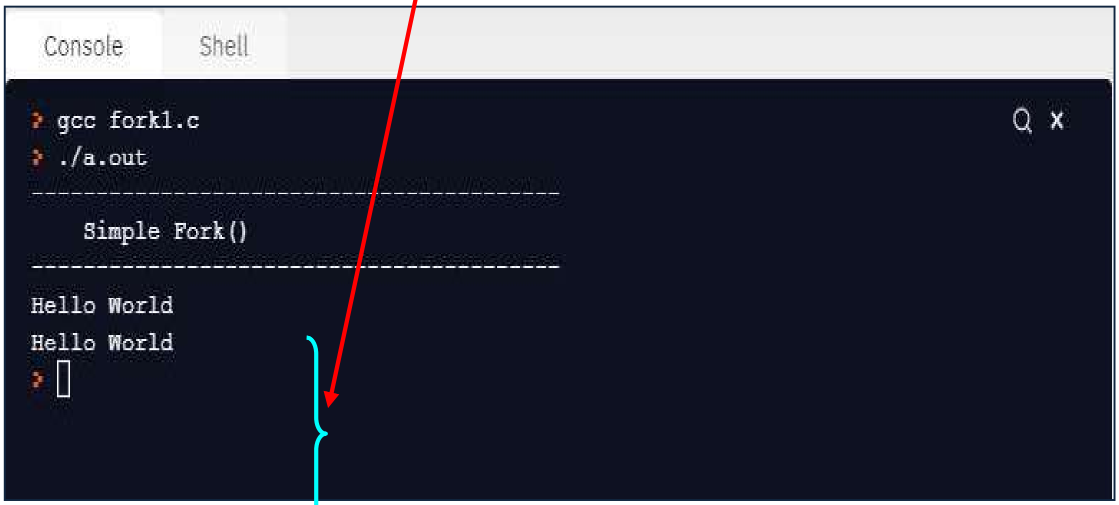
```
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("----------------------------------------------\n");
    printf("\tSimple Fork()\n");
    printf("----------------------------------------------\n");
// calling fork()
    fork();
    printf("Hello World\n");
    return 0;
}
```

Total Number of Processes are → 2^n

→ 2^1

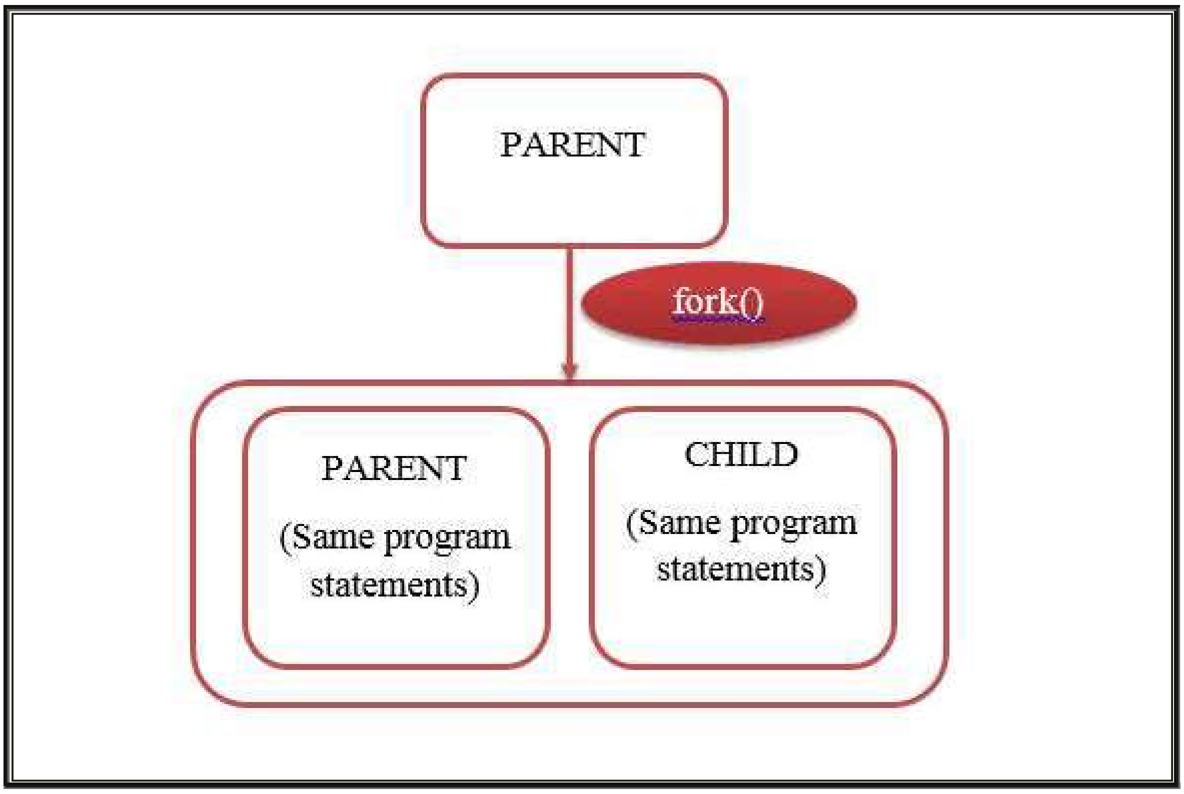
→ 2

OUTPUT



```
> gcc fork1.c
> ./a.out
---------------------------------------------
    Simple Fork()
---------------------------------------------
Hello World
Hello World
> 
```

Pictorial Representation



- When the child process is created, both the parent process and the child process will point to the next instruction (same Program Counter) after the fork().
- In this way the remaining instructions or C statements will be executed the total number of process times, that is 2^n times, where n is the number of fork() system calls

II. CALLING DOUBLE FORK

SOURCE CODE

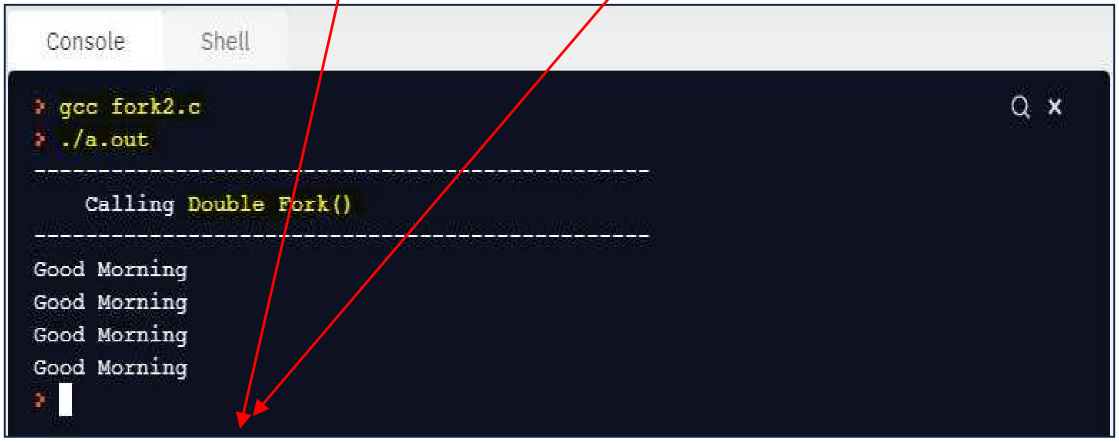
```
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("-----------------------------------------------\n");
    printf("\tCalling Double Fork()\n");
    printf("-----------------------------------------------\n");
// calling double fork()
    fork();
    fork();
    printf("Good Morning\n");
    return 0;
}
```

Total Number of Processes are $\rightarrow 2^n$

$\rightarrow 2^2$

$\rightarrow 4$

OUTPUT



```
> gcc fork2.c
> ./a.out
------------------------------------------------
    Calling Double Fork()
------------------------------------------------
Good Morning
Good Morning
Good Morning
Good Morning
>
```

III. CALLING TRIPLE FORK

Total Number of Processes are $\rightarrow 2^n$

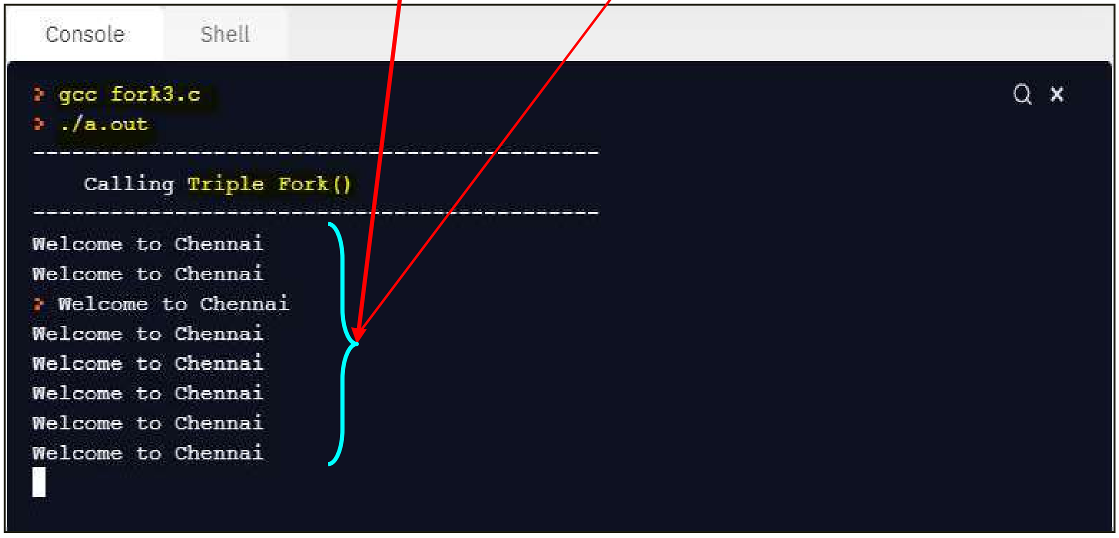
$\rightarrow 2^3$

$\rightarrow 8$

SOURCE CODE

```c
#include<unistd.h>
#include<stdio.h>
int main()
{
    printf("-----------------------------------------------\n");
    printf("\tCalling Triple Fork()\n");
    printf("-----------------------------------------------\n");
// calling triple fork()
    fork();
    fork();
    fork();
    printf("Welcome to Chennai\n");
    return 0;
}
```

OUTPUT



```
> gcc fork3.c
> ./a.out
--------------------------------------------
    Calling Triple Fork()
--------------------------------------------
Welcome to Chennai
Welcome to Chennai
> Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
Welcome to Chennai
```

IV. PRINT THE RETURNED VALUE OF CHILD AND PARENT PROCESSES USING FORK

SOURCE CODE

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

This header supports the pid_t

```c
int main()
{
    pid_t id;
    printf("----------------------------------------\n");
    printf("\tReturn Code of Parent & Child\n");
    printf("----------------------------------------\n");
// calling fork()
    id=fork();
    if(id==0)
```

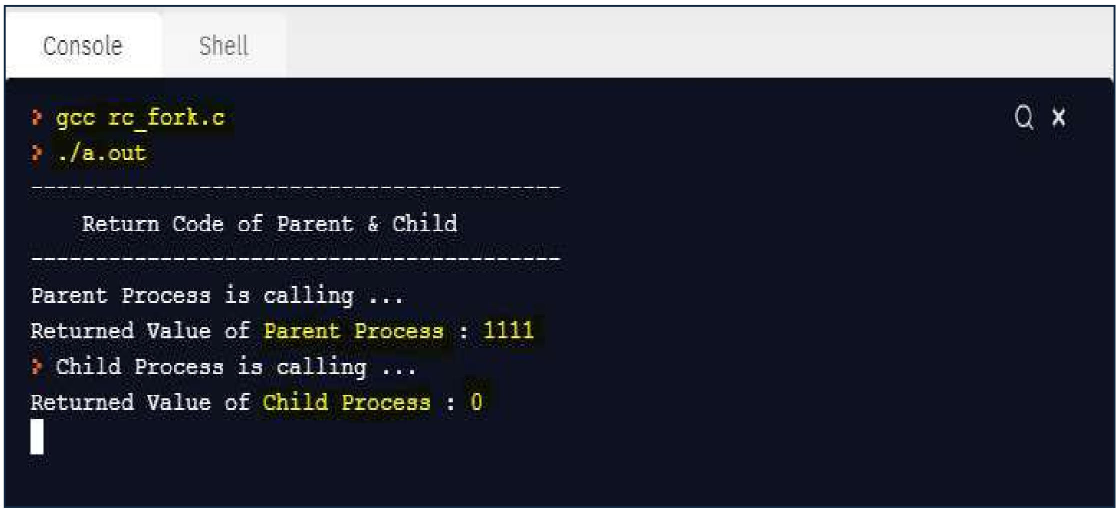
Child Process: It is identified by 0

```c
    {
        printf("Child Process is calling ...\n");
        printf("Returned Value of Child Process : %d\n",id);
    }
    else
```

Parent Process: It is identified by >0

```c
    {
        printf("Parent Process is calling ...\n");
        printf("Returned Value of Parent Process : %d\n",id);
    }
    return 0;
}
```

OUTPUT



```
> gcc rc_fork.c
> ./a.out
-----------------------------------------
    Return Code of Parent & Child
-----------------------------------------
Parent Process is calling ...
Returned Value of Parent Process : 1111
> Child Process is calling ...
Returned Value of Child Process : 0
```

Usage of getpid() and getppid()

- Two major functions which are used to get the process ids. They are
    1. getpid()
    2. getppid()

1. getpid()

- It is a built-in function and available in #include<unistd.h> header file
- It is used to return the process ID of child process (newly created process)
- Return value: pid_t

2. getppid()

- It is a built-in function and available in #include<unistd.h> header file
- It is used to return the process ID of parent process (caller of the newly created process)
- Return value: pid_t

pid_t

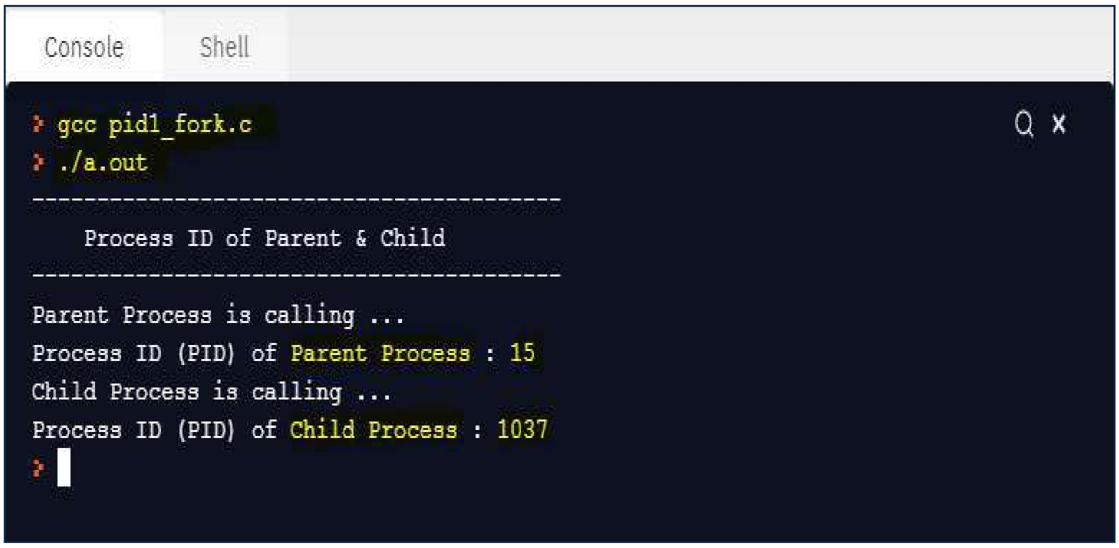
- It stands for process id type and built-in variable to store the process ids of parent and child function
- It is the type of the process ID which returns an unsigned integer value.
- It is available in #include<sys/types.h>

V. DISPLAY THE PROCESS ID (PID) OF PARENT AND CHILD PROCESSES USING GETPID() AND GETPPID()

SOURCE CODE

```
#include<unistd.h>
#include<stdio.h>
#include  <sys/types.h>
int main()
{
    int id;
    printf("----------------------------------------\n");
    printf("\tProcess ID of Parent & Child\n");
    printf("----------------------------------------\n");
// calling fork()
    id=fork();
    if(id==0)
    {
        printf("Child Process is calling ...\n");
        printf("Process ID (PID) of Child Process : %d\n",getpid());
    }
    else
    {
        printf("Parent Process is calling ...\n");
        printf("Process ID (PID) of Parent Process : %d\n",getppid());
    }
    return 0;
}
```

OUTPUT



```
> gcc pid1_fork.c
> ./a.out
----------------------------------------
    Process ID of Parent & Child
----------------------------------------
Parent Process is calling ...
Process ID (PID) of Parent Process : 15
Child Process is calling ...
Process ID (PID) of Child Process : 1037
>
```

Wait()

- It is system call and available in #include<sys/wait.h> file
- It blocks the current process (calling process), until one of its child processes terminate or a signal is received
- It takes one argument which is the address of an integer variable (stores the information of the process) and returns the process ID (PID) of completed child process
- Return type: pid_t
- If only one child process is terminated (finished its execution), then it returns the process ID of the terminated child process
- If more than one child processes are terminated, then it returns process ID of any terminated arbitrary child process.

The execution of wait() could have two possible situations.

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. It returns -1 immediately.

NOTABLE POINTS

- wait(NULL) will block the parent process until any of its children has finished their execution (parent process will be blocked until child process returns an exit status to the operating system which is then returned to parent process)
- If child finishes before parent reaches wait(NULL) then it will read the exit status, release the process entry in the process table and continue execution until it finishes as well.
- Wait can be used to make the parent process wait for the child to terminate (finish) but not the other way around
- Wait(NULL) simply making the parent wait for the child.
- On success, wait() returns the process ID of terminated child process while on failure it returns -1
- Once child process finishes, parent resumes and prints the rest of the statements of parent process

exit()

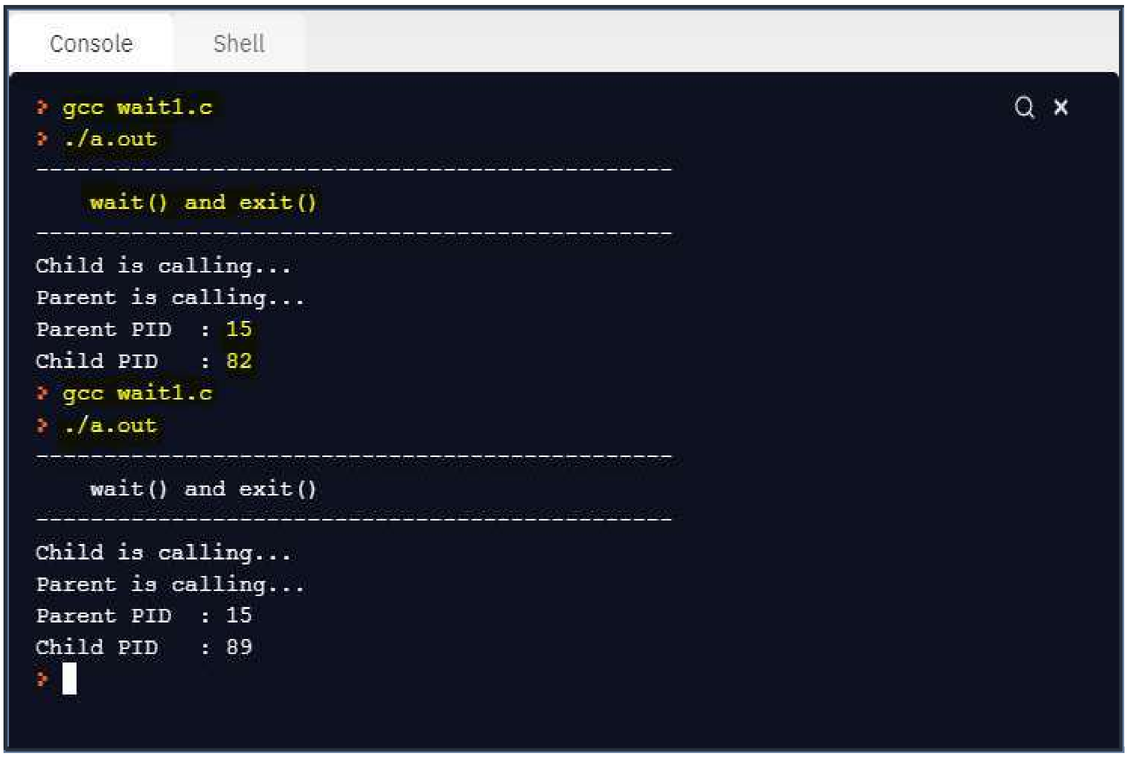
- It is system call and available in #include<stdlib> file
- It takes only one parameter which is exit status as a parameter
- It is used to close all files, sockets, frees all memory and then terminates the process.
- The parameter 0 indicates that the termination is normal.

VI. EXAMPLE OF WAIT AND EXIT SYSTEM CALLS

SOURCE CODE

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
int main()
{
    pid_t pd;
    printf("---------------------------------------------\n");
    printf("\twait() and exit()\n");
    printf("---------------------------------------------\n");
// execution of fork() call
    if (fork()== 0)
    {
        printf("Child is calling...\n");
// normal termination
        exit(0);
    }
    else
    {
// get the PID of terminated child process
        pd = wait(NULL);
        printf("Parent is calling...\n");
// print the process IDs of parent and child processes
        printf("Parent PID\t: %d\n", getppid());
        printf("Child PID\t: %d\n", pd);
    }
  return 0;
}
```

OUTPUT



```
> gcc wait1.c
> ./a.out
-----------------------------------------------
    wait() and exit()
-----------------------------------------------
Child is calling...
Parent is calling...
Parent PID  : 15
Child PID   : 82
> gcc wait1.c
> ./a.out
-----------------------------------------------
    wait() and exit()
-----------------------------------------------
Child is calling...
Parent is calling...
Parent PID  : 15
Child PID   : 89
> 
```

VII. SUM OF NUMBERS IN ARRAY USING CHILD AND PARENT PROCESS

Task

Write a linux c program to find the sum of the numbers in array in child process and execute the parent process after the execution of child process using system calls.

Used System calls:

fork(), wait()

SOURCE CODE

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
int main()
{
  int i,a[]={1,5,7,8,9};
```