

**Name:** Vishal Shashikant Salvi.

**UID:** 2019230069

**Batch:** C

**Class:** SE Comps

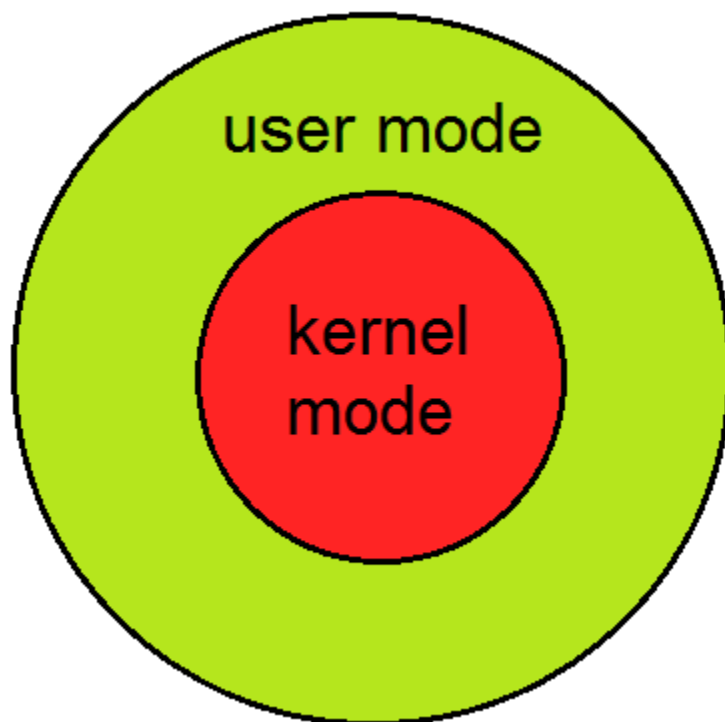
## Experiment No 1

**Aim:** To study different system calls related to process creation and usage.

**Theory:**

### Introduction to System Calls

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.



### Modes supported by the operating system

#### Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.

- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

### User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

### System Call

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via something called a **system call**.

When a program makes a system call, the mode is switched from user mode to kernel mode. This is called a **context switch**.

Then the kernel provides the resource which the program requested. After that, another context switch happens which results in change of mode from kernel mode back to user mode.

Generally, system calls are made by the user level programs in the following situations:

- Creating, opening, closing and deleting files in the file system.
- Creating and managing new processes.
- Creating a connection in the network, sending and receiving packets.
- Requesting access to a hardware device, like a mouse or a printer.

In a typical UNIX system, there are around 300 system calls. Some of them which are important ones in this context, are described below.

### Fork()

The `fork()` system call is used to create processes. When a process (a program in execution) makes a `fork()` call, an exact copy of the process is created. Now there are two processes, one being the **parent** process and the other being the **child** process.

The process which called the `fork()` call is the **parent** process and the process which is created newly is called the **child** process. The child process will be exactly the same as the

parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.

Both processes start execution from the next line of code i.e., the line after the `fork()` call. Let's look at an example:

```
// example.c
#include <stdio.h>
void main()
{
    int val;
    val = fork(); // line A
    printf("%d", val); // line B
}
```

When the above example code is executed, when **line A** is executed, a child process is created. Now both processes start execution from **line B**. To differentiate between the child process and the parent process, we need to look at the value returned by the `fork()` call.

The difference is that, in the parent process, `fork()` returns a value which represents the **process ID** of the child process. But in the child process, `fork()` returns the value 0.

This means that according to the above program, the output of parent process will be the **process ID** of the child process and the output of the child process will be 0.

## Exec()

The `exec()` system call is also used to create processes. But there is one big difference between `fork()` and `exec()` calls. The `fork()` call creates a new process while preserving the parent process. But, an `exec()` call replaces the address space, text segment, data segment etc. of the current process with the new process.

It means, after an `exec()` call, only the new process exists. The process which made the system call, wouldn't exist.

There are many flavors of `exec()` in UNIX, one being `exec1()` which is shown below as an example:

```
// example2.c
#include <stdio.h>
void main()
{
    exec1("/bin/ls", "ls", 0); // line A
    printf("This text won't be printed unless an error occurs in exec().");
}
```

}

As shown above, the first parameter to the `execl()` function is the address of the program which needs to be executed, in this case, the address of the `ls` utility in UNIX. Then it is followed by the name of the program which is `ls` in this case and followed by optional arguments. Then the list should be terminated by a NULL pointer (0).

When the above example is executed, at line A, the `ls` program is called and executed and the current process is halted. Hence the `printf()` function is never called since the process has already been halted. The only exception to this is that, if the `execl()` function causes an error, then the `printf()` function is executed.

### Example of System Call

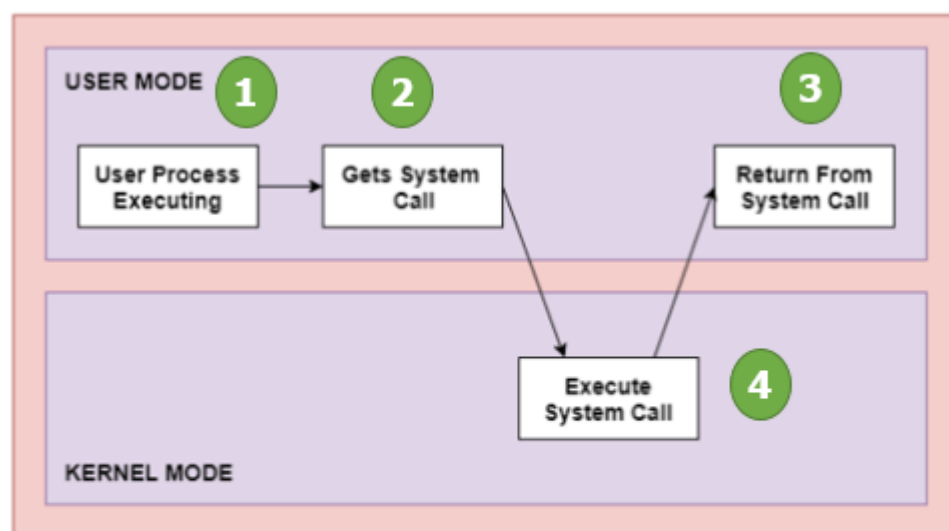
For example if we need to write a program code to read data from one file, copy that data into another file. The first information that the program requires is the name of the two files, the input and output files.

In an interactive system, this type of program execution requires some system calls by OS.

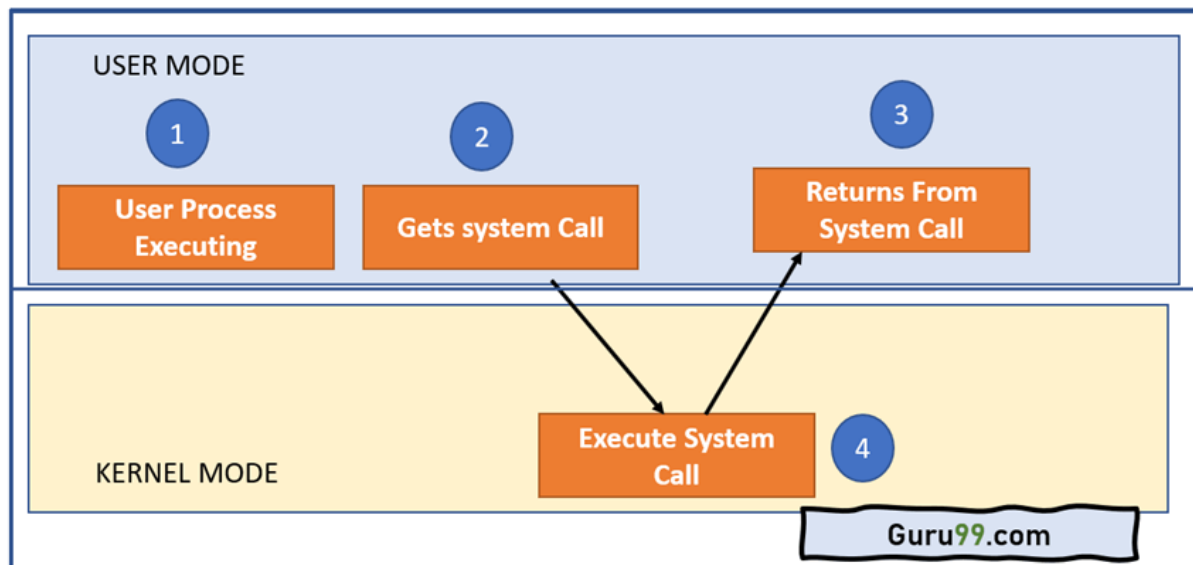
- First call is to write a prompting message on the screen
- Second, to read from the keyboard, the characters which define the two files.

### How System Call Works?

Here are steps for System Call:



Guru99.com



Architecture of the System Call

As you can see in the above-given diagram.

**Step 1)** The processes executed in the user mode till the time a system call interrupts it.

**Step 2)** After that, the system call is executed in the kernel-mode on a priority basis.

**Step 3)** Once system call execution is over, control returns to the user mode.,

**Step 4)** The execution of user processes resumed in Kernel mode.

### Why do you need System Calls in OS?

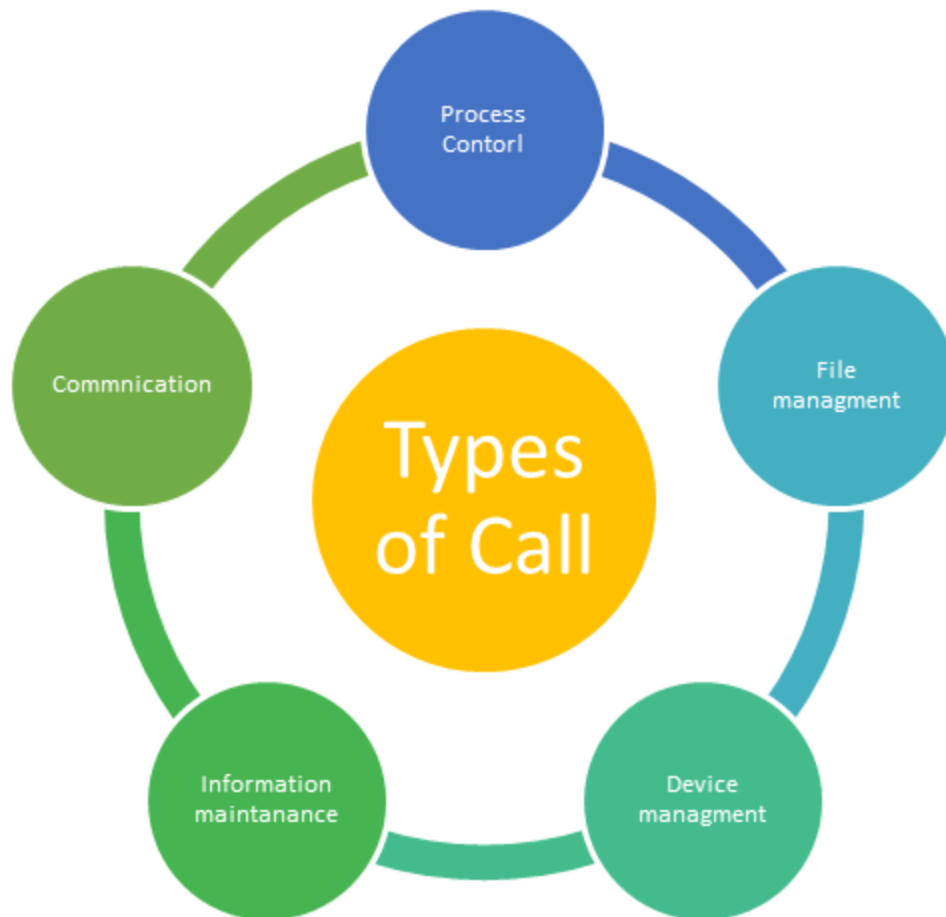
Following are situations which need system calls in OS:

- Reading and writing from files demand system calls.
- If a file system wants to create or delete files, system calls are required.
- System calls are used for the creation and management of new processes.
- Network connections need system calls for sending and receiving packets.
- Access to hardware devices like scanner, printer, need a system call.

### Types of System calls

Here are the five types of system calls used in OS:

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communications



### **Process Control**

This system calls perform the task of process creation, process termination, etc.

Functions:

- End and Abort
- Load and Execute
- Create Process and Terminate Process
- Wait and Signed Event
- Allocate and free memory

### **File Management**

File management system calls handle file manipulation jobs like creating a file, reading, and writing, etc.

Functions:

- Create a file
- Delete file
- Open and close file
- Read, write, and reposition
- Get and set file attributes

## **Device Management**

Device management does the job of device manipulation like reading from device buffers, writing into device buffers, etc.

Functions

- Request and release device
- Logically attach/ detach devices
- Get and Set device attributes

## **Information Maintenance**

It handles information and its transfer between the OS and the user program.

Functions:

- Get or set time and date
- Get process and device attributes

## **Communication:**

These types of system calls are specially used for interprocess communications.

Functions:

- Create, delete communications connections
- Send, receive message
- Help OS to transfer status information
- Attach or detach remote devices

## **Rules for passing Parameters for System Call**

Here are general common rules for passing parameters to the System Call:

- Parameters should be pushed on or popped off the stack by the operating system.
- Parameters can be passed in registers.
- When there are more parameters than registers, it should be stored in a block, and the block address should be passed as a parameter to a register.

## **Important System Calls Used in OS**

### **wait()**

In some systems, a process needs to wait for another process to complete its execution. This type of situation occurs when a parent process creates a child process, and the execution of the parent process remains suspended until its child process executes.

The suspension of the parent process automatically occurs with a wait() system call. When the child process ends execution, the control moves back to the parent process.

### **fork()**

Processes use this system call to create processes that are a copy of themselves. With the help of this system Call parent process creates a child process, and the execution of the parent process will be suspended till the child process executes.

### **exec()**

This system call runs when an executable file in the context of an already running process that replaces the older executable file. However, the original process identifier remains as a new process is not built, but stack, data, head, data, etc. are replaced by the new process.

### **kill():**

The kill() system call is used by OS to send a termination signal to a process that urges the process to exit. However, a kill system call does not necessarily mean killing the process and can have various meanings.

### **exit():**

The exit() system call is used to terminate program execution. Specially in the multi-threaded environment, this call defines that the thread execution is complete. The OS reclaims resources that were used by the process after the use of exit() system call.



**Fork****Input:**

```
#include <stdio.h>
void main()
{
    fork();
    fork();
    fork();
    fork();
    printf("hello world\n");

}
```

**Output:**

```
students@CE-Lab6-608-U09:~$
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
```

**Input:**

```
#include<stdio.h>
void main(){
    int pid;
    printf("Before fork \n");
    printf("Parent %d, Child %d \n" , getpid(),getppid());
    pid = fork();
    printf("After Fork \n");
    printf("Parrent %d,Child %d \n" , getpid(),getppid());

}
```

**Output:**

```
students@CE-Lab6-608-U09:~$ ./a.out
Before fork
Parent 3243, Child 2954
After Fork
Parrent 3243,Child 2954
After Fork
Parrent 3244,Child 1577
```

## **Programs:**

### **Task 1: Baby steps to forking.**

Write a program p1.c that forks a child and prints the following (in the parent and child process),

Parent : My process ID is: 12345  
Parent : The child process ID is: 15644  
and the child process prints  
Child : My process ID is: 15644  
Child : The parent process ID is: 12345.

### **Input:**

```
#include<stdio.h>
void main(){
    int pid;

    printf(" Parent: My process ID is: %d \n ", getpid());
    printf(" Parent : The child process ID is: %d \n", getppid());
    pid = fork();
    if(pid<0)
    {
        printf(" ERROR!! \n ");
    }

    else if(pid==0)
    {
        printf("\n ");
    }

    else{
        printf(" Child: My process ID is: %d \n", getppid());
        printf(" Child: The parent process ID is: %d \n",getpid());
    }

}
```

### **Output:**

```
students@CE-Lab6-608-U09:~$ ./a.out
Parent: My process ID is: 6597
Parent : The child process ID is: 2954
Child: My process ID is: 2954
Child: The parent process ID is: 6597
```

**Task 2:**

Write another program p2.c that does exactly same as in previous exercise, but the parent process prints it's messages only after the child process has printed its messages and exited. Parent process waits for the child process to exit, and then prints its messages and a child exit message,

Parent : The child with process ID 12345 has terminated.

**Input:**

```
#include<stdio.h>
void main(){
    int pid;

    pid = fork();
    if(pid<0)
    {
        printf(" ERROR!! \n ");
    }

    else if(pid==0)
    {
        printf(" Child: My process ID is: %d \n ", getppid());
        printf(" Child : The child process ID is: %d \n", getpid());
    }

    else{
        wait();

        printf(" Parent: My process ID is: %d \n ", getpid());
        printf(" Parent : The child process ID is: %d \n", getppid());
        printf(" Parent : The child with process ID %d has terminated \n" , getpid());

    }

}
```

**Output:**

students@CE-Lab6-608-U09:~\$ ./a.out

Child: My process ID is: 7069

Child : The child process ID is: 7070

Parent: My process ID is: 7069

Parent : The child process ID is: 2954

Parent : The child with process ID 7069 has terminated

### **Task 3:**

A program mycat.c, available as part of this lab, reads input from stdin and writes output to stdout. Write a program p3.c that executes the binary program mycat (compiled from mycat.c) as a child process of p3.

Hint: The program forks a child process, the child process executes mycat binary, and the parent process waits for the child process to exit.

Also, check what happens when input is redirected to p3.

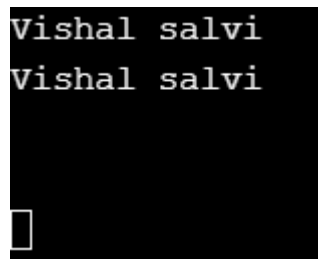
### **Input:**

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

int main()
{
    char buf[512];
    int n;

    for(;;){
        n = read(0, buf, sizeof buf);
        if(n == 0) break;
        if(n < 0){
            fprintf(stdout, "read error\n");
            exit(-1);
        }
        if(write(1, buf, n) != n)
        {
            fprintf(stdout, "write error\n");
            exit(-1);
        }
    }
    return 1;
}
```

### **Output:**

A screenshot of a terminal window with a black background and white text. The text 'Vishal salvi' is printed on two separate lines. At the bottom left, there is a small white cursor icon.

//Ctrl+Z to exit the program

**Task 4a: Writing to a file without opening it**

Write a program p4a.c which takes a file name as command line argument. Parent opens file and forks a child process. Both processes write to the file, "hello world! I am the parent" and "hello world! I am the child". Verify that the child can write to the file without opening it. The parent process should wait for the child process to exit, and it should display and child exit message.

**Input:**

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

int main(int argc,char* argv[]){
    int childpid=fork();
    int fd;
    FILE *fptr;

    fptr=fopen(argv[1],"a+");
    fd=fileno(fptr);

    if(childpid<0){
        printf("Uncessful to create child.");
    }
    else if(childpid==0){
        printf("Child %d: Writing to file %d\n",getpid(),fd);
        fprintf(fptr,"\n%s","Hello World! I am the Child");
        fclose(fptr);
    }
    else{
        printf("Parent: File Opened. fd=%d\n",fd);
        printf("Parent %d: Writing to file %d\n",getpid(),fd);
        fprintf(fptr,"%s","Hello World! I am the Parent");
        fclose(fptr);
        wait(NULL);
        printf("Parent: The child has terminated.\n");
    }
    return 0;
}
```

**Output:**

```
Parent: File Opened. fd=3
Parent 6207: Writing to file 3
Child 6208: Writing to file 3
Parent: The child has terminated.
```

**Task 4b: Input file re-direction magic**

Write a program p4b.c which takes a file name as a command line argument. The program should print content of the file to stdout from a child process. The child process should execute the mycat program.

Cannot use any library functions like printf, scanf, cin, read, write in the parent of child process.

Hint: close of a file results in its descriptor to be reused on a subsequent open.

Note: Do not make any modifications in mycat.c

**Input:**

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

int main(int argc,char* argv[]){

    int childpid=fork();
    int fd;
    char s;
    FILE *fptr;

    fptr=fopen(argv[1],"rf");
    fd=fileno(fptr);

    if(childpid<0){
        printf("Uncessful to create child.");
    }
    else if(childpid==0){

        while((s=fgetc(fptr))!=EOF) {
            printf("%c",s);
        }

        fclose(fptr);

        char *args[] = { "./MYCAT",NULL};
        execvp(args[0], args);
    }
    else{
        printf("Parent: File Opened. fd=%d\n",fd);
        wait(NULL);
        printf("\nParent: The child has terminated.\n");
    }
    return 0;
}
```

**Output:**

```
Parent: File Opened. fd=3
Hello World! I am the Parent
MYCAT Program
MYCAT Program
```

### **Task 5: Orphan**

Write a program p5.c to demonstrate the state of process as an orphan. The program forks a process, and the parent process prints the following messages.

Parent : My process ID is: 12345

Parent : The child process ID is: 15644

The child process prints message

Child : My process ID is: 15644

Child : The parent process ID is: 12345

sleeps for few seconds, and it prints the same messages one more time. By the time the child process wakes from sleep, the parent process should have exited.

### **Input:**

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

int main()
{

    int pid = fork();

    if (pid > 0) {
        printf("Parent : My process ID is: %d\n",getpid());
        printf("Parent : The child process ID is: %d\n",pid);
        sleep(5);
        printf("Parent : The parent has terminated. \n");
    }

    else if (pid == 0)
    {
        printf("Child : My process ID is: %d\n",getpid());
        printf("Child : The parent process ID is: %d\n",getppid());
        sleep(10);
        printf("Child : My process ID is: %d\n",getpid());
        printf("Child : The parent process ID is: %d\n",getppid());
    }

    return 0;
}
```

### **Output:**

Parent: My Process ID is: 6310

Parent: The Child Process ID is: 6311

Child: My Process ID is: 6311  
Child: The Parent Process ID is: 6310  
Parent: The parent has terminated.  
Child: My Process ID is: 6311  
Child: The Parent Process ID is: 1

### **Task 6: Zombie**

Write a program p6.c to demonstrate the presence of zombie processes. The program forks a process, and the parent process prints the message

Parent : My process ID is: 12345  
Parent : The child process ID is: 15644  
and the child process prints the message  
Child : My process ID is: 15644  
Child : The parent process ID is: 12345

After printing the messages, the parent process sleeps for 1 minute, and then waits for the child process to exit. The child process waits for some keyboard input from user after displaying the messages, and then exits.

Display the process state of child process while it was waiting for input and after the input using ps command.

ps -o pid,stat --pid <child's PID>.

Refer man ps for the details of different process states. Reason about the output.

### **Input:**

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

int main()
{

    int pid = fork();

    if (pid > 0) {
        printf("Parent : My process ID is: %d\n",getpid());
        printf("Parent : The child process ID is: %d\n",pid);
        sleep(5);
        wait(NULL);
        printf("Parent : The parent has terminated. \n");
    }

    else if (pid == 0)
    {

        printf("Child : My process ID is: %d\n",getpid());
```



```
printf("Child : The parent process ID is: %d\n",getppid());

}

return 0;
}
```

**Output:**

```
Parent : My process ID is: 21168
Parent : The child process ID is: 21169
Child : My process ID is: 21169
Child : The parent process ID is: 21168
Parent : The parent has terminated.

...Program finished with exit code 0
Press ENTER to exit console.█
```

**Task 7.a: Recursive fork**

Write a program p7a.c that takes a number n as a command line argument and creates n child processes recursively, i.e., parent process creates first child, the first child creates second child, and so on. The child processes should exit in the reverse order of the creation, i.e., the inner most child exits first, then second inner most, and so on. Print the order of creation of the child process, and their termination order as shown in sample output

**Input:**

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

void createProcess(int n){
if(n==0){
printf("Child %d exited \n",getpid());
return;
}

int pid = fork();
if (pid > 0){
wait();
printf("Child %d existed \n",getpid());
}
else if (pid == 0){

printf("Child %d is created \n",getpid());
n--;
createProcess(n);
}
}

int main()
{
int n ;
printf("Parent is: %d \n",getpid());
printf("Numbaer of Children:");
scanf("%d",&n);

    int pid = fork();
    if (pid > 0)
    {
        wait();
        printf("Parent exited\n");
        return 0;
    } else if(pid == 0){
printf("Child %d is created \n",getpid());
n--;
createProcess(n);
}
}
```

}

**Output:**

```
Parent is: 13926
Numbaer of Children:10
Child 13927 is created
Child 13928 is created
Child 13929 is created
Child 13930 is created
Child 13931 is created
Child 13932 is created
Child 13933 is created
Child 13934 is created
Child 13935 is created
Child 13936 is created
Child 13936 exited
Child 13935 existed
Child 13934 existed
Child 13933 existed
Child 13932 existed
Child 13931 existed
Child 13930 existed
Child 13929 existed
Child 13928 existed
Child 13927 existed
Parent exited
```

**Task 7.b: Sequential fork**

Write a program p7b.c that takes a number n as command line argument and creates n child processes sequentially, i.e., the first parent process (p7b) creates all children in a loop without any delays. Let the child processes sleep for a small random duration (use the urand\_r() call, and print the creation and exit order of the child processes. Note that the random numbers used for sleep should be different across the child processes.

**Input:**

```
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<stdlib.h>

#include<time.h>

void

main (int argc, char *argv[])

{

    srand (time (0));

    int n = atoi (argv[1]);

    printf ("Parent is : %d\n", getpid ());

    int parentpid = getpid ();

    printf ("Number of children: %d\n", n);

    int temp;

    for (int i = 0; i < n; i++)

    {

        int pid = fork ();

        temp = 1 + rand () % 10;

        if (pid == 0)

        {

            printf ("Child %d is created\n", getpid ());

            sleep (temp);

            printf ("Child %d exited\n", getpid ());

            break;

        }

    }

}
```

```

        }

    }

    if (getpid () == parentpid)
    {
        for (int i = 0; i < n; i++)
            wait ();

        printf ("Parent exited\n");
    }

}

```

### **Output:**

```

/p7b 10
Parent is : 7461
Number of children: 10
Child 7462 is created
Child 7463 is created
Child 7464 is created
Child 7465 is created
Child 7466 is created
Child 7467 is created
Child 7468 is created
Child 7469 is created
Child 7470 is created
Child 7471 is created
Child 7469 exited
Child 7470 exited
Child 7463 exited
Child 7462 exited
Child 7467 exited
Child 7465 exited
Child 7468 exited
Child 7464 exited
Child 7466 exited
Child 7471 exited
Parent exited

```

### **Task 7.c:**

Write another program p7c.c, that creates n child processes similar to sequential creation of p7b.c. Further, each child also does the similar sleep action for a random duration.

In the parent process, set it up so that child termination is in the reverse order of sequential creation.

#### **Input:**

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<time.h>
#include<signal.h>
#include<sys/types.h>

pid_t child_pid = -1;
void main(int argc, char *argv[]) {
    int p;
    int i = 0;
    unsigned int seed = time(NULL);

    if(argc == 1) {
        printf("Value of n is not passed.\n");
        exit(0);
    } else {
        int n = atoi(argv[1]);
        printf("PROCESS...PID = %d; PPID = %d :
        CREATED\n",getpid(),getppid());
        for(i = 1; i < n; i++) {
            child_pid = fork();
            wait(0);
            if(child_pid == 0 && child_pid != -1) {
                printf("PROCESS...PID = %d; PPID = %d :
                CREATED\n",getpid(),getppid());
                seed = time(NULL);
                sleep(rand_r(&seed)%10+1);
            } else if(child_pid > 0) {
                printf("PROCESS...PID = %d; PPID = %d :
                EXITED\n",getpid(),getppid());
                exit(0);
            }
        }
        printf("PROCESS...PID    =    %d;    PPID    =    %d    :
        EXITED\n",getpid(),getppid());
    }
}
```

#### **Output:**

```
PROCESS...PID = 6937; PPID = 6859 : CREATED
PROCESS...PID = 6938; PPID = 6937 : CREATED
```

```
PROCESS...PID = 6939; PPID = 6938 : CREATED  
PROCESS...PID = 6940; PPID = 6939 : CREATED  
PROCESS...PID = 6941; PPID = 6940 : CREATED  
PROCESS...PID = 6941; PPID = 6940 : EXITED  
PROCESS...PID = 6940; PPID = 6939 : EXITED  
PROCESS...PID = 6939; PPID = 6938 : EXITED  
PROCESS...PID = 6938; PPID = 6937 : EXITED  
PROCESS...PID = 6937; PPID = 6859 : EXITED
```

**Conclusion:** Thus from this experiment we studied about System call as well as Types of system call also implement some system calls.