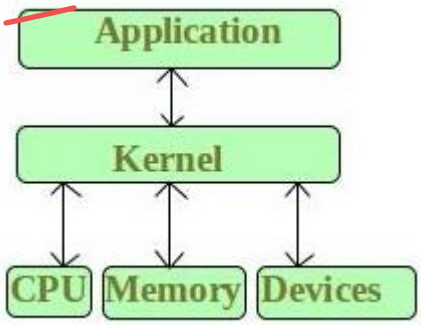
Microkernel in Operating Systems

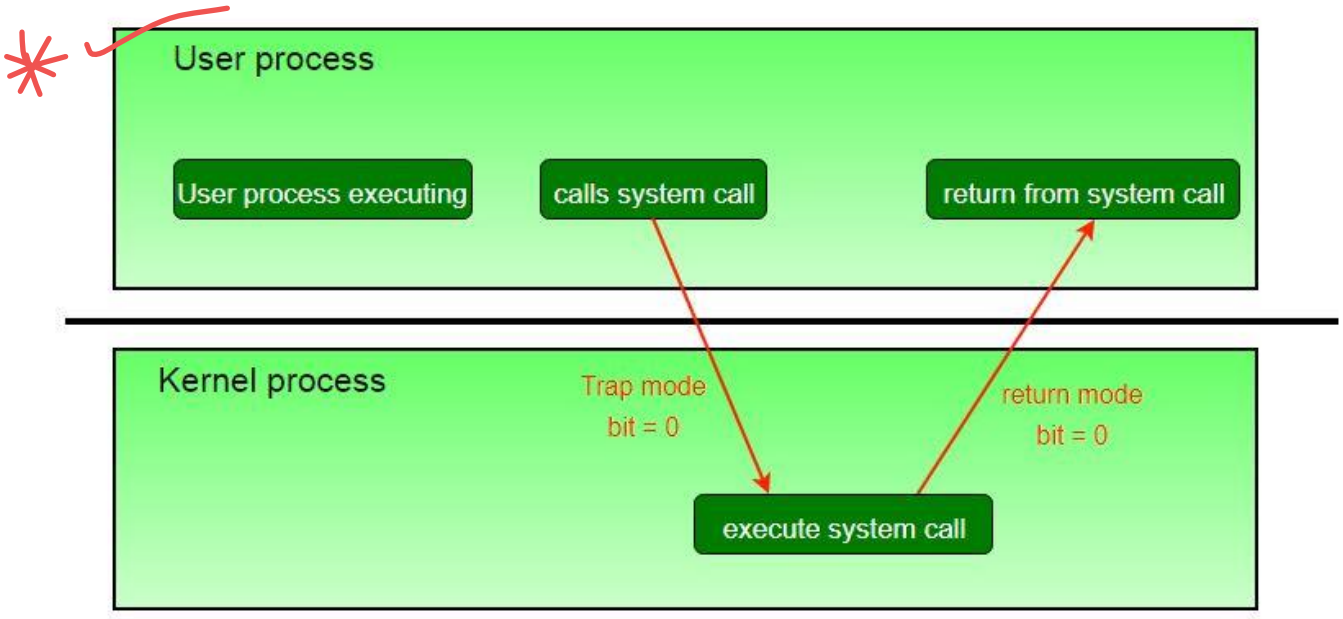
Kernel is the core part of an operating system that manages system resources. It also acts as a bridge between the application and hardware of the computer. It is one of the first programs loaded on start-up (after the Bootloader).



Kernel mode and User mode of CPU operation

The CPU can execute certain instructions only when it is in kernel mode. These instructions are called privilege instruction. They allow the implementation of special operations whose execution by the user program could interface with the functioning of the operating system or activity of another user program. For example, instruction for managing memory protection.

- The operating system puts the CPU in kernel mode when it is executing in the kernel so, that kernel can execute some special operation.
- The operating system puts the CPU in user mode when a user program is in execution so, that the user program cannot interface with the operating system program.
- User-level instruction does not require special privilege. Example are ADD,PUSH,etc.



The concept of modes can be extended beyond two, requiring more than a single mode bit CPUs that support virtualization use one of these extra bits to indicate when the virtual machine

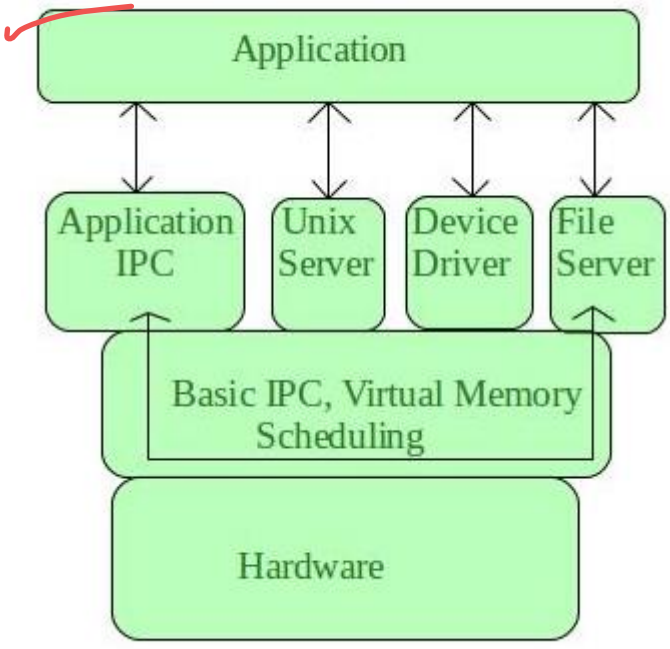
manager, VMM, is in control of the system. The VMM has more privileges than ordinary user programs, but not so many as the full kernel.

System calls are typically implemented in the form of software interrupts, which causes the hardware's interrupt handler to transfer control over to an appropriate interrupt handler, which is part of the operating system, switching the mode bit to kernel mode in the process. The interrupt handler checks exactly which interrupt was generated, checks additional parameters (generally passed through registers) if appropriate, and then calls the appropriate kernel service routine to handle the service requested by the system call.

User programs' attempts to execute illegal instructions (privileged or non-existent instructions), or to access forbidden memory areas, also generate software interrupts, which are trapped by the interrupt handler, and control is transferred to the OS, which issues an appropriate error message, possibly dumps data to a log (core) file for later analysis, and then terminates the offending program.

What is Microkernel?

A microkernel is one of the classifications of the kernel. Being a kernel it manages all system resources. But in a microkernel, the **user services** and **kernel services** are implemented in different address spaces. The user services are kept in **user address space**, and kernel services are kept under **kernel address space**, thus also reduces the size of kernel and size of an operating system as well.



It provides minimal services of process and memory management. The communication between client program/application and services running in user address space is established through message passing, reducing the speed of execution microkernel. The Operating System **remains unaffected** as user services and kernel services are isolated so if any user service fails it does not affect kernel service. Thus it adds to one of the advantages of a microkernel. It is easily **extendible** i.e. if any new services are to be added they are added to user address space and hence require no modification in kernel space. It is also portable, secure, and reliable. Examples of microkernel-based operating systems include L4, QNX, and MINIX

Microkernel Architecture –
Since the kernel is the core part of the operating system, so it is meant for handling the most important services only. Thus in this architecture, only the most important services are inside the kernel and the rest of the OS services are present inside the system application program. Thus users are able to interact with those not-so-important services within the system application. And the microkernel is solely responsible for the most important services of the operating system they are named as follows:

- Inter process-Communication
- Memory Management
- CPU-Scheduling

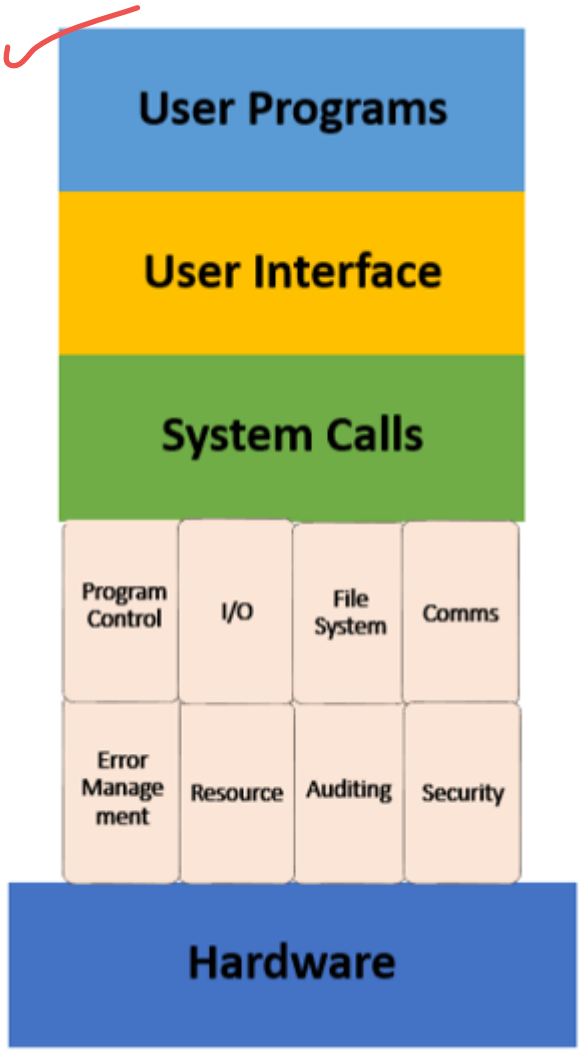
Advantages of Microkernel –

- **Modularity:** Because the kernel and servers can be developed and maintained independently, the microkernel design allows for greater modularity. This can make adding and removing features and services from the system easier.
- **Fault isolation:** The microkernel design aids in the isolation of faults and their prevention from affecting the entire system. If a server or other component fails, it can be restarted or replaced without causing any disruptions to the rest of the system.
- **Performance:** Because the kernel only contains the essential functions required to manage the system, the microkernel design can improve performance. This can make the system faster and more efficient.
- **Security:** The microkernel design can improve security by reducing the system's attack surface by limiting the functions provided by the kernel. Malicious software may find it more difficult to compromise the system as a result of this.
- **Reliability:** Microkernels are less complex than monolithic kernels, which can make them more reliable and less prone to crashes or other issues.
- **Scalability:** Microkernels can be easily scaled to support different hardware architectures, making them more versatile.
- **Portability:** Microkernels can be ported to different platforms with minimal effort, which makes them useful for embedded systems and other specialized applications.

What is System Call in Operating System?

A **system call** is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.

System call offers the services of the operating system to the user programs via API (Application Programming Interface). System calls are the only entry points for the kernel system.



Example of System Call

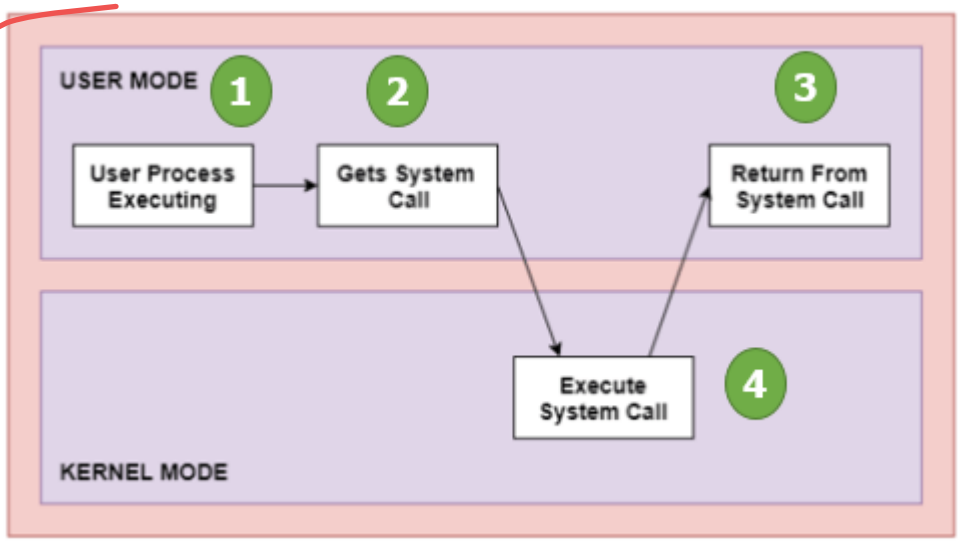
For example, if we need to write a program code to read data from one file, copy that data into another file. The first information that the program requires is the name of the two files, the input and output files.

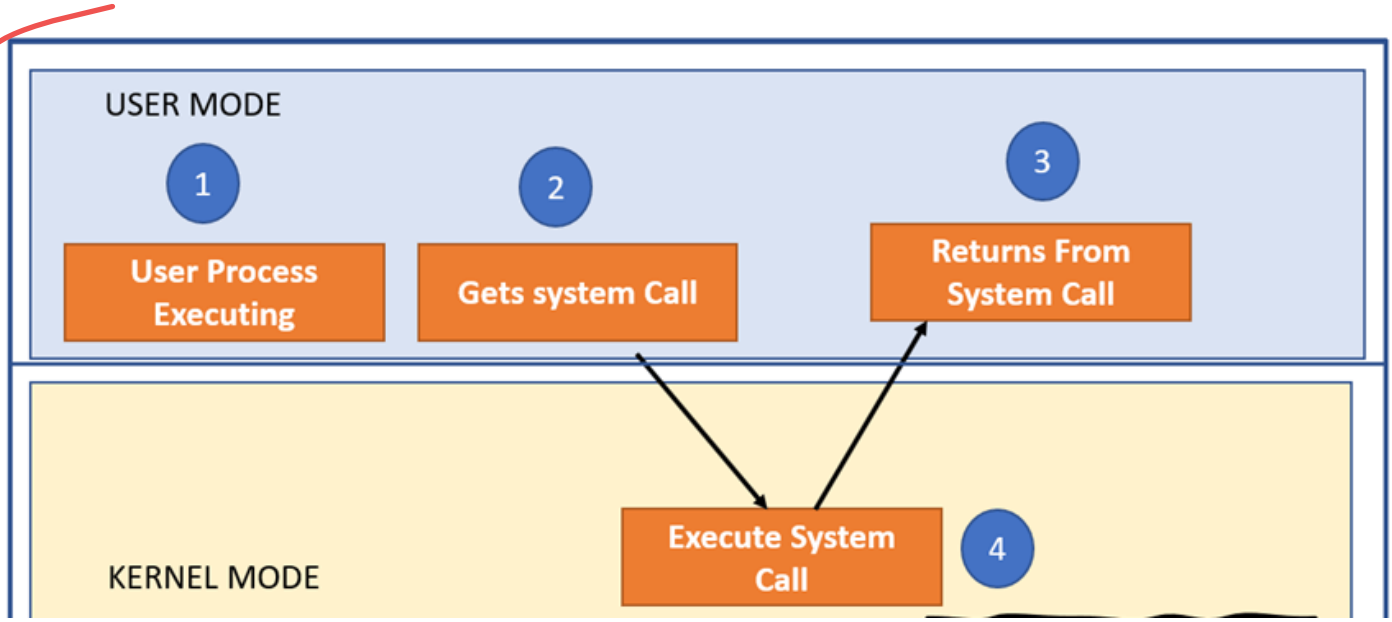
In an interactive system, this type of program execution requires some system calls by OS.

- 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 the steps for System Call in OS:





Architecture of the System Call

As you can see in the above-given System Call example 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:

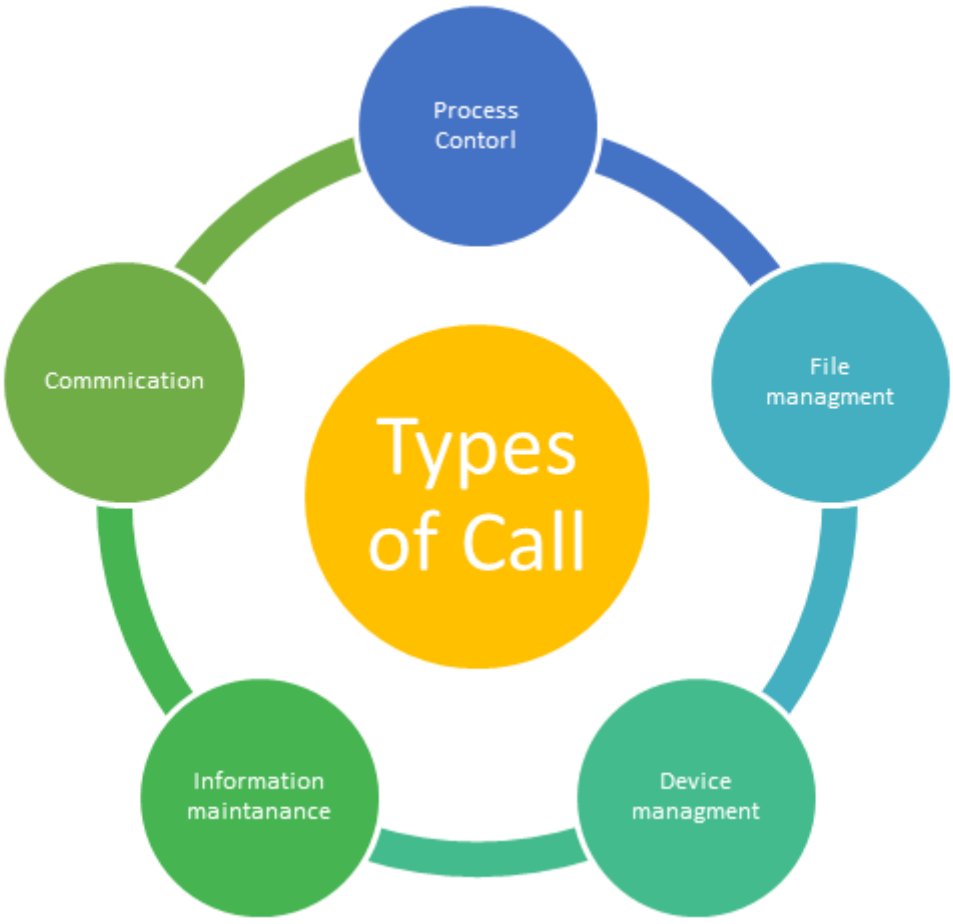
- Reading and writing from files demand system calls.
- If a file system wants to create or delete files, system calls are required.

- 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 in OS:

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



Types of System calls in OS

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 Signal 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.

Summary:

Categories	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()

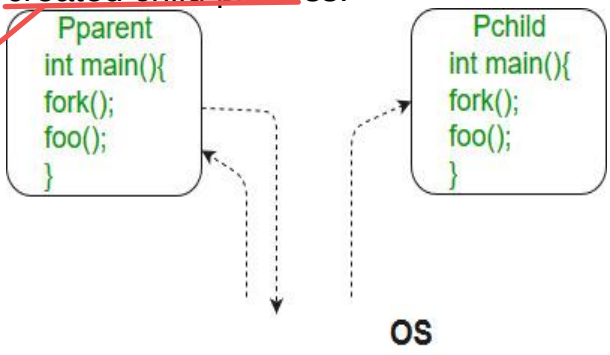
Categories	Windows	Unix
Device manipulation	SetConsoleMode() ReadConsole() WriteConsole()	loctl() read() write()
File manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() write() close!)
Information maintanence	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup ()	Chmod() Umask() Chown()

Fork() system call

https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.



Predict the Output of the following program:.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // make two process which run same
    // program after this instruction
    fork();
    printf("Hello world!\n");
    return 0;
}
```

Output:

```
Hello world!

Hello world!
```

Calculate number of times hello is printed:

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

1. Output:
2. hello
3. hello
4. hello
5. hello
6. hello
7. hello
8. hello
9. hello

The number of times 'hello' is printed is equal to number of process created. Total Number of Processes = 2^n, where n is number of fork system calls. So here n = 3, $2^3 = 8$

Let us put some label names for the three lines:

```
fork ();    // Line 1

fork ();    // Line 2

fork ();    // Line 3
```

```
     L1       // There will be 1 child process
  /     \     // created by line 1.
 L2      L2    // There will be 2 child processes
/  \    /  \   //  created by line 2
L3  L3  L3  L3  // There will be 4 child processes
                // created by line 3
```

So there are total eight processes (new child processes and one original process).

If we want to represent the relationship between the processes as a tree hierarchy it would be the following:

The main process: P0
Processes created by the 1st fork: P1
Processes created by the 2nd fork: P2, P3
Processes created by the 3rd fork: P4, P5, P6, P7

```
        P0
    /   |   \
   P1    P4   P2
  /  \          \
P3    P6         P5
/
P7
```

Predict the Output of the following program:

```
#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:

```
1.

Hello from Child!

Hello from Parent!
```

(or)

```
2.

Hello from Parent!

Hello from Child!
```

In the above code, a child process is created. fork() returns 0 in the child process and positive integer in the parent process.
Here, two outputs are possible because the parent process and child process are running concurrently. So we don't know whether the OS will first give control to the parent process or the child process.

Important: Parent process and child process are running the same program, but it does not mean they are identical. OS allocate different data and states for these two processes, and the control flow of these processes can be different. See next example:

Practice Questions:

https://www.geeksforgeeks.org/fork-practice-questions/