Introduction of System Call

In computing, a **system call** is a programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and an operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.
A user program can interact with the operating system using a system call. A number of services are requested by the program, and the OS responds by launching a number of systems calls to fulfill the request. A system call can be written in high-level languages like C or Pascal or in assembly language. If a high-level language is used, the operating system may directly invoke system calls, which are predefined functions.
A system call is a mechanism used by programs to request services from the operating system (OS). In simpler terms, it is a way for a program to interact with the underlying system, such as accessing hardware resources or performing privileged operations.
A system call is initiated by the program executing a specific instruction, which triggers a switch to kernel mode, allowing the program to request a service from the OS. The OS then handles the request, performs the necessary operations, and returns the result back to the program.

Services Provided by System Calls

- Process creation and management
- Main memory management
- File Access, Directory, and File system management
- Device handling(I/O)
- Protection
- Networking, etc.
    - **Process control:** end, abort, create, terminate, allocate, and free memory.
    - **File management:** create, open, close, delete, read files,s, etc.
    - **Device management**
    - **Information maintenance**
    - **Communication**

Features of System Calls

- **Interface:** System calls provide a well-defined interface between user programs and the operating system. Programs make requests by calling specific functions, and the operating system responds by executing the requested service and returning a result.

- **Protection:** System calls are used to access privileged operations that are not available to normal user programs. The operating system uses this privilege to protect the system from malicious or unauthorized access.
- **Kernel Mode:** When a system call is made, the program is temporarily switched from user mode to kernel mode. In kernel mode, the program has access to all system resources, including hardware, memory, and other processes.
- **Context Switching:** A system call requires a context switch, which involves saving the state of the current process and switching to the kernel mode to execute the requested service. This can introduce overhead, which can impact system performance.
- **Error Handling:** System calls can return error codes to indicate problems with the requested service. Programs must check for these errors and handle them appropriately.
- **Synchronization:** System calls can be used to synchronize access to shared resources, such as files or network connections. The operating system provides synchronization mechanisms, such as locks or semaphores, to ensure that multiple programs can access these resources safely.

System Calls Advantages

- **Access to hardware resources:** System calls allow programs to access hardware resources such as disk drives, printers, and network devices.
- **Memory management:** System calls provide a way for programs to allocate and deallocate memory, as well as access memory-mapped hardware devices.
- **Process management:** System calls allow programs to create and terminate processes, as well as manage inter-process communication.
- **Security:** System calls provide a way for programs to access privileged resources, such as the ability to modify system settings or perform operations that require administrative permissions.
- **Standardization:** System calls provide a standardized interface for programs to interact with the operating system, ensuring consistency and compatibility across different hardware platforms and operating system versions.

How does System Call Work?

Here is the detailed explanation step by step how system call work:

- **User need special resources :** Sometimes programs need to do some special things which can't be done without the permission of OS like reading from a file, writing to a file , getting any information from the hardware or requesting a space in memory.
- **Program makes a system call request :** There are special predefined instruction to make a request to the operating system. These instruction are nothing but just a "system call". The program uses these system calls in its code when needed.

- **Operating system sees the system call :** When the OS sees the system call then it recongnises that the program need help at this time so it temporarily stop the program execution and give all the control to special part of itself called ‘Kernel’ . Now ‘Kernel’ solve the need of program.
- **Operating system performs the operations :**Now the operating system perform the operation which is requested by program . Example : reading content from a file etc.
- **Operating system give control back to the program :** After performing the special operation, OS give control back to the program for further execution of program .

open(): Accessing a file on a file system is possible with the open() system call. It gives the file resources it needs and a handle the process can use. A file can be opened by multiple processes simultaneously or just one process. Everything is based on the structure and file system.

read(): Data from a file on the file system is retrieved using it. In general, it accepts three arguments:

1. A description of a file.
2. A buffer for read data storage.
3. How many bytes should be read from the file
   Before reading, the file to be read could be identified by its file descriptor and opened using the open() function.

wait(): In some systems, a process might need to hold off until another process has finished running before continuing. When a parent process creates a child process, the execution of the parent process is halted until the child process is complete. The parent process is stopped using the wait() system call. The parent process regains control once the child process has finished running.

write(): Data from a user buffer is written using it to a device like a file. A program can produce data in one way by using this system call. generally, there are three arguments:

1. A description of a file.
2. A reference to the buffer where data is stored.
3. The amount of data that will be written from the buffer in bytes.

What is Open API in UNIX?

There is a set of generic APIs in UNIX which is used to manipulate files. One of these APIs is the **open API**. The ***open*** API is used to create new files and also to establish a connection between a process and a file. After a file is created, any process can call the open function and it gets a file descriptor to refer to the file, which contains the inode information. The **prototype** of the ***open*** function is given below:

```
#include <sys/types.h>

#include <fcntl.h>

int open(const char *path_name, int access_mode, mode_t permission);
```

- If **successful**, the open function returns a non-negative integer representing the open file descriptor.
- If **unsuccessful**, the open function returns -1

Use of the arguments in the function prototype:

First Argument: path_name
This specifies the pathname of the file to be created or opened. This may be a relative pathname or an absolute pathname. If the pathname is a symbolic link, the open function will resolve the link refers to a non-symbolic link file to which the link refers.

Second Argument: access_mode
It is an integer value that specifies how the file is to be accessed by the calling process. This value should be one of the manifested constants described below as defined in the <fcntl.h> header:

Access mode flag	**Use**
O_RDONLY	Opens the file for read-only
O_WRONLY	Opens the file for write-only
O_RDWR	Opens the file for read and write

These access node flags can be bitwise-ORed with the access modifier flags given below to change the access mechanism of the file:

Access modifier flag	Use
O_APPEND	Appends data to the end of the file (Applicable only to Regular files)
O_CREAT	Creates file if it does not exist (Applicable only to Regular files)
O_EXCL	It causes open function to fail if the named file exists already. Also, it can be used with the O_CREAT flag only. (Applicable only to Regular files)
O_NONBLOCK	Specifies that any subsequent read or write on the file should be nonblocking (Applicable only to FIFO and device files)
O_NOCTTY	Specifies not to use named terminal device file as the calling process control terminal (Applicable only to terminal device files)
O_TRUNC	Discards the file content and sets the file size to 0 bytes if the file exists (Applicable only to Regular files)

Note:

- If a file is to be opened for read-only, then the file should already exist and also, no other modifier flags can be used with it.
- If a file is opened for read-write or write-only, then any modifier flags can be specified.
- If the named file does not exist and the O_CREAT file is not specified, then the open function will abort with a failure return status.

Third Argument: permission

This argument is used only when a new file is being created. This is required only if the O_CREAT flag is set in the access_mode argument. It specifies the access permission o the file for its owner, group member, and others. This argument is defined as mode_t and its value should be constructed based on the manifested constants defined in the <sys/stat.h> header. The symbolic names for file permission are given in the table below:

SYMBOL	MEANING
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write and execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write and execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write and execute by others

Example 1:

int fdesc= open("abc/xyz/geeksforgeeks",O_RDWR | O_APPEND, 0);

This example opens an already existing file called /abc/xyz/geeksforgeeks for read and write in append mode

The access mode flag O_RDWR is bitwise ORed "|" with access modifier flag O_APPEND

1. Write a C/C++ program to implement the cat command using general file API's.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
main( int argc,char *argv[3] )
{
int fd,i;
char buf[2];
fd=open(argv[1],O_RDONLY,0777);
if(fd==-argc)
{
printf("file open error");
}
else
{
while((i=read(fd,buf,1))>0)
{
printf("%c",buf[0]);
}
close(fd);
}
}
```

```
Output: [UNIXLAB@localhost ~]$ gcc -0 p.out prgm1.c
UNIXLAB@localhost ~]$cat >h.txt /*creating a text file*/
hello
UNIXLAB@localhost ~]$./p.out h.txt /*appending text file to display contents on the terminal
without using cat command*/
hello
```

2. Write a C/C++ program to implement the cp (copy) command using general file API's.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
```

```
#define BUFF_SIZE 1024

int main(int argc, char* argv[])
{
   int srcFD, destFD, nbread, nbwrite;
   char buff[BUFF_SIZE];

   if(argc != 3)
   {
      printf("\nUsage: cpcmd source_file destination_file\n");
      exit(EXIT_FAILURE);
   }

   srcFD = open(argv[1], O_RDONLY);
   if(srcFD == -1)
   {
      printf("\nError opening file %s errno = %d\n", argv[1], errno);
      exit(EXIT_FAILURE);
   }

   destFD = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR |
S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
   if(destFD == -1)
   {
      printf("\nError opening file %s errno = %d\n", argv[2], errno);
      exit(EXIT_FAILURE);
   }

   while((nbread = read(srcFD, buff, BUFF_SIZE)) > 0)
   {
      if(write(destFD, buff, nbread) != nbread)
      {
         printf("\nError in writing data to %s\n", argv[2]);
         exit(EXIT_FAILURE);
      }
   }

   if(nbread == -1)
      printf("\nError in reading data from %s\n", argv[1]);

   if(close(srcFD) == -1)
      printf("\nError in closing file %s\n", argv[1]);

   if(close(destFD) == -1)
      printf("\nError in closing file %s\n", argv[2]);
```

```
    exit(EXIT_SUCCESS);
}
```