Batch: A1	**Roll No.: 16010123012**
Experiment No. 03	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with	

TITLE: System calls

AIM: To understand the working Process based system calls.

Expected Outcome of Experiment:

CO 1. To introduce basic concepts and functions of operating systems.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **William Stallings "Operating Systems" Person, Seventh Edition Edition.**
3. **Sumitabha Das " UNIX Concepts & Applications", McGraw Hill Second Edition.**

Pre Lab/ Prior Concepts:

System Calls Provide the Interface between a process and the OS.

System calls are usually made when a process in user mode requires access to a resource.

Then it requests the kernel to provide the resource via a system call.

System calls are required in the following situations –

1) If a file system requires the creation or deletion of files.
2) Reading and writing from files also require a system call.
3) Creation and management of new processes.

4) Network connections also require system calls. This includes sending and receiving packets.

5) Access to a hardware devices such as a printer, scanner etc. requires a system call.

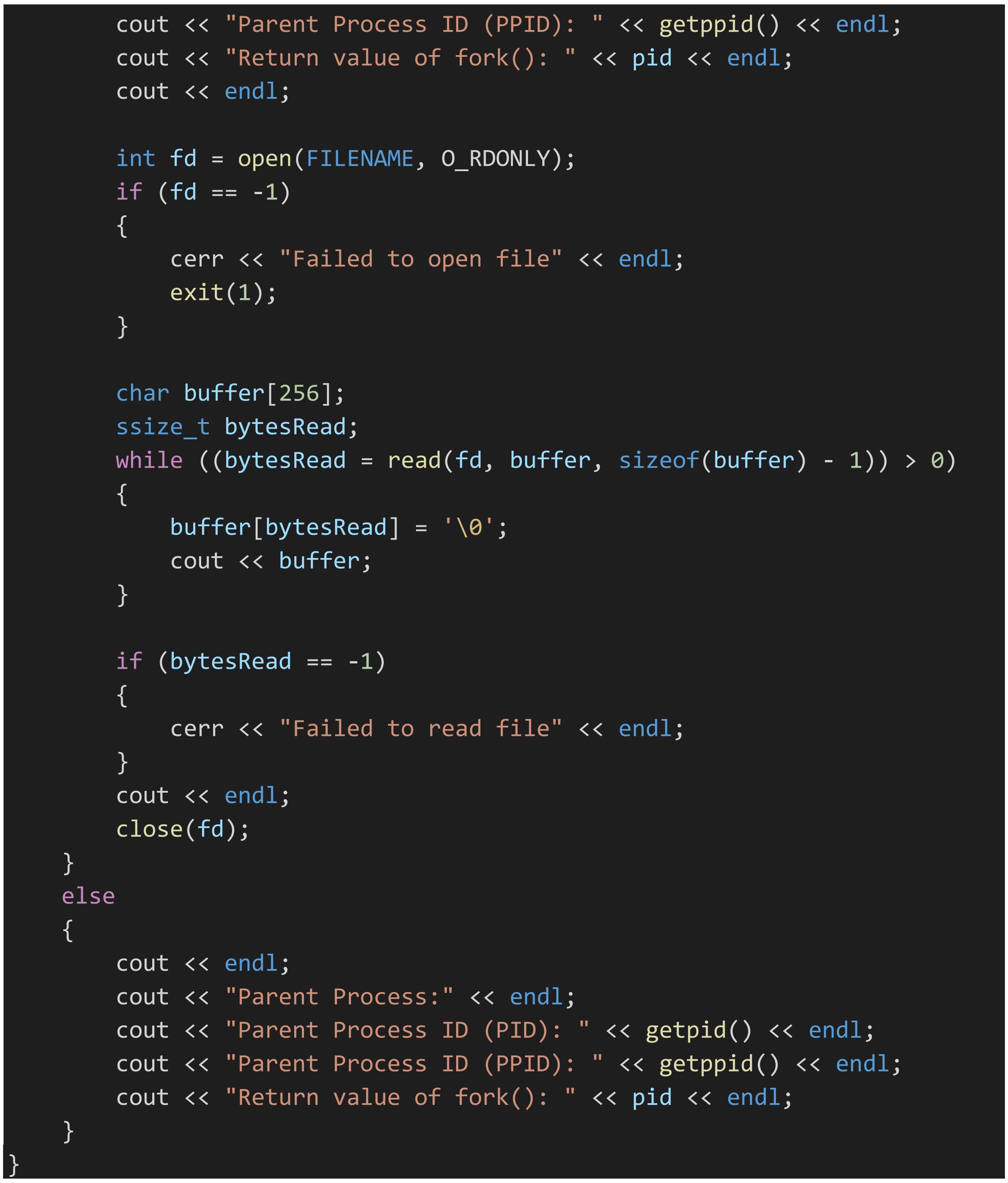
Description of the application to be implemented:

Program for System Call:

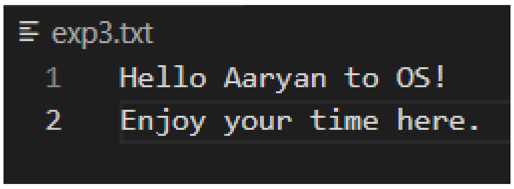
1. Write a Program for creating a process using System call (E.g fork()). Create a child process. Display the details about that process using getpid and getppid functions in each block. Also print the return value of fork() system call in each block.
 In the child process, Open a text file using file system calls and read the contents of the text file and display it.

Implementation details: (Screen shots)

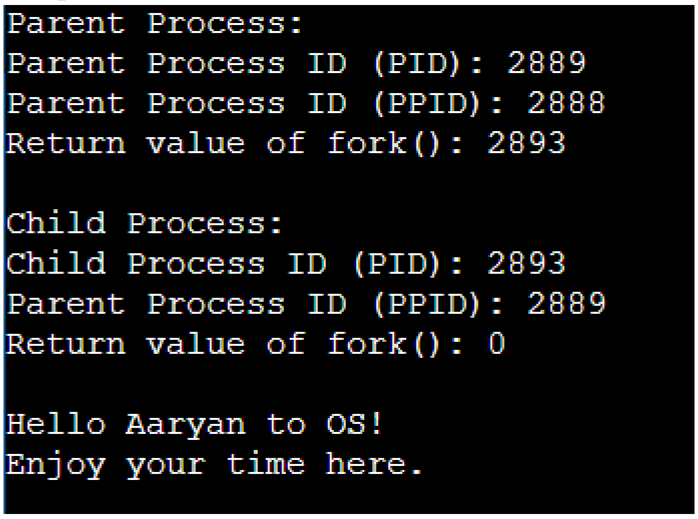
```
#include <iostream>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <cstdlib>

#define FILENAME "exp3.txt"
#define endl '\n';
using namespace std;

int main()
{
    pid_t pid;
    pid = fork();

    if (pid < 0)
    {
        cerr << "Fork failed" << endl;
        exit(1);
    }
    else if (pid == 0)
    {
        cout << endl;
        cout << "Child Process:" << endl;
        cout << "Child Process ID (PID): " << getpid() << endl;
```


```
        cout << "Parent Process ID (PPID): " << getppid() << endl;
        cout << "Return value of fork(): " << pid << endl;
        cout << endl;

        int fd = open(FILENAME, O_RDONLY);
        if (fd == -1)
        {
            cerr << "Failed to open file" << endl;
            exit(1);
        }

        char buffer[256];
        ssize_t bytesRead;
        while ((bytesRead = read(fd, buffer, sizeof(buffer) - 1)) > 0)
        {
            buffer[bytesRead] = '\0';
            cout << buffer;
        }

        if (bytesRead == -1)
        {
            cerr << "Failed to read file" << endl;
        }
        cout << endl;
        close(fd);
    }
    else
    {
        cout << endl;
        cout << "Parent Process:" << endl;
        cout << "Parent Process ID (PID): " << getpid() << endl;
        cout << "Parent Process ID (PPID): " << getppid() << endl;
        cout << "Return value of fork(): " << pid << endl;
    }
}
```

Content of file -


exp3.txt
```
Hello Aaryan to OS!
Enjoy your time here.
```

Output:



```
Parent Process:
Parent Process ID (PID): 2889
Parent Process ID (PPID): 2888
Return value of fork(): 2893

Child Process:
Child Process ID (PID): 2893
Parent Process ID (PPID): 2889
Return value of fork(): 0

Hello Aaryan to OS!
Enjoy your time here.
```

Conclusion:

In this experiment, I used the fork() system call in C++ to create a child process. The parent and child processes executed independently with the parent process receiving the child's process ID, and the child process receiving a return value of 0 from fork(). I used getpid() and getppid() to display the process IDs. Additionally, in the child process, I opened and read a text file (exp3.txt), displaying its content. Through this experiment, I gained a practical understanding of process creation, process identification, and file handling in a multi-process environment.

Post Lab Descriptive Questions

1) Describe System Call Interface.

The System Call Interface (SCI) is the interface through which user applications interact with the operating system kernel. It provides a way for programs to request services like file operations, process control, memory management, and hardware access. This mechanism helps in providing a controlled and secure way for user programs to access critical system resources, ensuring that the applications cannot directly manipulate hardware or kernel-level operations.
System Calls: These are requests made by applications to the kernel. For example, reading a file or creating a process.
Mode Switching: When making a system call, the program switches from user mode to kernel mode. The OS then performs the requested service and returns the result to the program in user mode.
Types of System Calls: Common categories include file management, process control, memory allocation, and device management.

Example: A program calling open() for a file results in a system call to the OS to open that file.

2) List the types of System Calls.

Here are the main types of System Calls:

1. **Process Control**: Manage processes (e.g., fork(), exit(), wait(), exec())
2. **File Management**: Handle files (e.g., open(), read(), write(), close())
3. **Memory Management**: Allocate and free memory (e.g., mmap(), brk(), sbrk())
4. **Device Management**: Manage input/output devices (e.g., ioctl(), read(), write())
5. **Information Maintenance**: Get or set system info (e.g., getpid(), getppid(), alarm())
6. **Communication**: Inter-process communication (e.g., pipe(), msgget(), semop(), shmget())

Date: 11/02/2025 **Signature of faculty in-charge**