EXPERIMENT:- 02

AIM:- To write C Programs using the following system calls of UNIX operating system fork, exec, getpid, getppid, exit, wait, close, stat, opendir, readdir.

System Call:-

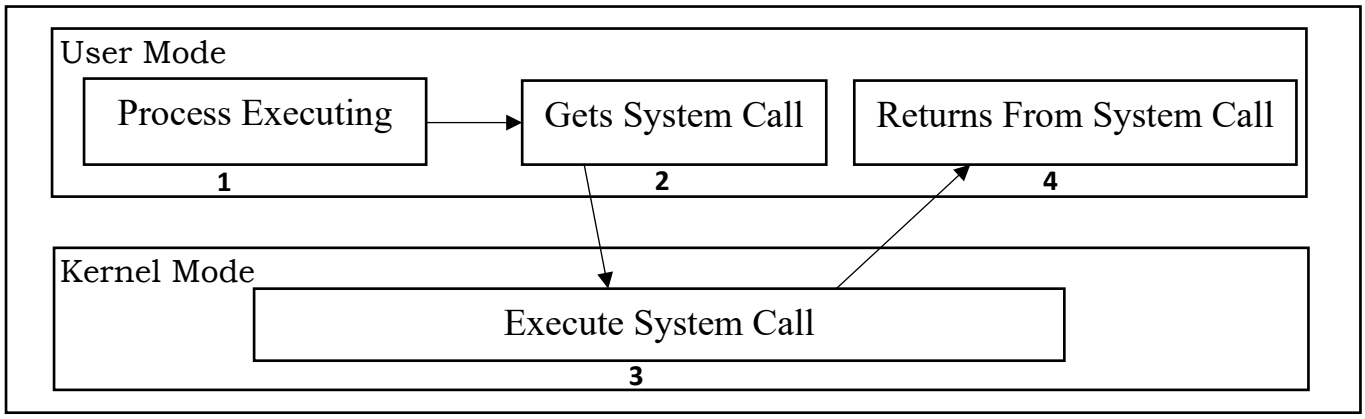
- It is a mechanism that provides the interface between a process and an OS.
- It is a programming method, which a computer program requests a service from kernel to OS.
- E.g., Lets copy contents from one file to another.

How System Calls Work ?

It works in 2 different modes:-

1. User Mode
   The user mode carries 3 different tasks:
   i. Process Execution
   ii. Gets System Call
   iii. Returns From System Call

2. Kernel Mode
   Kernel mode carries only one task i.e., Execute System Call.



Why System Call Is Required ?

In every OS system call plays a very important role:-

- Reading and Writing from a file.
- Used to create or delete files.
- Required to create and manage process.
- Communication through network for sending & receiving data/packets.
- Connecting with hardware devices or Communicate with any I/O device.

System Call Methods:-

It uses many pre-define methods:

- **fork()**- creates a new process (child) by duplicating the calling process (parent). Execution of parent suspends until child executes.
  - Ex-1

```
shankar@Shankar ~> touch fork1.c
shankar@Shankar ~> gedit fork1.c
```

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

int main() {
        fork();
        printf("Hello World!\n");
        return 0;
}
```

```
shankar@Shankar ~> gcc -o fork1 fork1.c
shankar@Shankar ~> ./fork1
Hello World!
Hello World!
```

  - Ex-2

```
shankar@Shankar ~> touch fork2.c
shankar@Shankar ~> gedit fork2.c
```

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

int main(){
        fork();
        fork();
        fork();
        printf("Hello\n");
        return 0;
}
```

```
shankar@Shankar ~> gcc -o fork2 fork2.c
shankar@Shankar ~> ./fork2
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

❖ Ex-3

```
shankar@Shankar ~> touch fork3.c
shankar@Shankar ~> gedit fork3.c
```

```c
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main(){
        int i , n = 10;
        for(i = 0; i < n; i++){
                fork();
                printf("Hello Shankar\n");
        }
        return 0;
}
```

```
shankar@Shankar ~> gcc -o fork3 fork3.c
shankar@Shankar ~> ./fork3
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
Hello Shankar
```

❖ **getpid()** - returns the process id (PID) of the calling process.

```
shankar@Shankar ~> touch pid.cpp
shankar@Shankar ~> gedit pid.cpp
```

```
1 #include<iostream>
2 #include<unistd.h>
3 using namespace std;
4 int main(){
5         int pid = fork();
6         if(pid == 0 )
7         cout << "\nCurrent process id of Process: " << getpid() << endl;
8         return 0;
9 }
```

```
shankar@Shankar ~> g++ -o pid pid.cpp
shankar@Shankar ~> ./pid

Current process id of Process: 1849
```

❖ **getppid()** – returns the process ID of the parent of the calling process.

```
shankar@Shankar ~> touch ppid.cpp
shankar@Shankar ~> gedit ppid.cpp
```

```
1 #include<iostream>
2 #include<unistd.h>
3 using namespace std;
4 int main(){
5         int pid;
6         pid = fork();
7         if (pid ==0)
8         {
9                 cout << "\n Parent Process id: " << getpid() << endl;
10                cout << "\n Child Process with parent id: " <<getppid() << endl;
11        }
12        return 0;
13 }
```

```
shankar@Shankar ~> g++ -o ppid ppid.cpp
shankar@Shankar ~> ./ppid

 Parent Process id: 2010

 Child Process with parent id: 9
```

❖ **exec()** – runs when an executable file in the context of an already running process that replaces the older executable file.

```
shankar@Shankar ~> touch exec.c
shankar@Shankar ~> gedit exec.c
```

```c
#include<stdio.h>
#include<unistd.h>
int main(){
        char *binaryPath = "/bin/ls";
        char *arg1 = "-1h";
        char *arg2 = "/home";
        execl(binaryPath, binaryPath, arg1, arg2, NULL);
        return 0;
}
```

```
shankar@Shankar ~> gcc -o exec exec.c
shankar@Shankar ~> ./exec
shankar
```

❖ **opendir()** - function opens a directory stream corresponding to the directory name, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

```
shankar@Shankar ~> touch opendir.c
shankar@Shankar ~> gedit opendir.c
```

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

int main() {
        DIR *dir;
        dir = opendir(".");     // open current directory
        if (dir == NULL) {     // check if directory was opened successfully
                fprintf(stderr, "Error Opening Directory\n");
                exit(-1);
        }
        else {
                printf("The directory was opened successfully.\n");
                closedir(dir);     // close directory
        }
        return 0;
}
```

```
shankar@Shankar ~> gcc -o opendir opendir.c
shankar@Shankar ~> ./opendir
The directory was opened successfully.
```

❖ **readdir()** – function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or if an error occurred.

```
shankar@Shankar ~> touch readdir.c
shankar@Shankar ~> gedit readdir.c
```

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

int main(){
        DIR *dir;
        struct dirent *entry;
        dir = opendir(".");     // open current directory
        if(dir == NULL){     // check if directory was opened successfully
                fprintf(stderr, "Error Opening Directory");
                exit(-1);
        }
        else {
                printf("The files and directories in the current directory are:\n");
                while((entry = readdir(dir)) != NULL){     // read directory entries
                        printf("%s\n", entry->d_name);     // print name of each entry
                }
                closedir(dir);     // close directory
        }
        return 0;
}
```

```
shankar@Shankar ~> gcc -o readdir readdir.c
shankar@Shankar ~> ./readdir
The files and directories in the current directory are:
opendir.c
opendir
.sudo_as_admin_successful
pid.cpp
fork2.c
.bash_history
readdir
exec
fork3.c
.motd_shown
.config
.profile
exec.c
.bash_logout
fork3
fork1.c
.bashrc
readdir.c
fork1
pid
fork2
.local
.lesshst
ppid.cpp
ppid
..
.
.dbus
```

❖ **stat()** - display file or file system status.

```
shankar@Shankar ~> touch stat.c
shankar@Shankar ~> gedit stat.c
```

```c
#include<stdio.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>

void sfile(char const filename[]);
int main(){
        ssize_t read;
        char* buffer = 0;
        size_t buf_size = 0;
        printf("Enter the name of a file to check:- ");
        read = getline(&buffer, &buf_size, stdin);
        if(read <= 0) {
                printf("getline failed\n");
                exit(1);
        }
        if(buffer[read-1] == '\n'){
                buffer[read-1] = 0;
        }
        int s = open(buffer, O_RDONLY);
        if(s == -1){
                printf("File doesn't exist\n");
                exit(1);
        }
        else{
                sfile(buffer);
        }
        free(buffer);
        return 0;
}
void sfile(char const filename[]){
        struct stat sfile;
        if(stat(filename, &sfile)==1){
                printf("Error Occurred\n");
        }
        // Accessing data members of stat struct
        printf("\nFile st_uid: %d \n",sfile.st_uid);
        printf("\nFile st_blksize: %ld \n",sfile.st_blksize);
        printf("\nFile st_gid: %d \n",sfile.st_gid);
        printf("\nFile st_blocks: %ld \n",sfile.st_blocks);
        printf("\nFile st_size: %ld \n",sfile.st_size);
        printf("\nFile st_nlink: %u \n",(unsigned int)sfile.st_nlink);
        printf("\nFile Permissions User:\n");
        printf((sfile.st_mode & S_IRUSR)? "r":"-");
        printf((sfile.st_mode & S_IWUSR)? "w":"-");
        printf((sfile.st_mode & S_IXUSR)? "x":"-");
        printf("\n");
        printf("\nFile Permissions Group:\n");
        printf((sfile.st_mode & S_IRGRP)? "r":"-");
        printf((sfile.st_mode & S_IWGRP)? "w":"-");
        printf((sfile.st_mode & S_IXGRP)? "x":"-");
        printf("\n");
        printf("\nFile Permissions Other:\n");
        printf((sfile.st_mode & S_IROTH)? "r":"-");
        printf((sfile.st_mode & S_IWOTH)? "w":"-");
        printf((sfile.st_mode & S_IXOTH)? "x":"-");
        printf("\n");
}
```

```
shankar@Shankar ~> gcc -o stat stat.c
shankar@Shankar ~> ./stat
Enter the name of a file to check:- pid.cpp

File st_uid: 1000

File st_blksize: 4096

File st_gid: 1000

File st_blocks: 8

File st_size: 186

File st_nlink: 1

File Permissions User:
rw-

File Permissions Group:
r--

File Permissions Other:
r--
```

❖ **wait()** - blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction.

```
shankar@Shankar ~> touch wait.c
shankar@Shankar ~> gedit wait.c
```

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

int main() {
    pid_t pid;
    int status;
    pid = fork();   // create a child process
    if (pid < 0) {  // error occurred
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) {    // child process
        printf("Child Process: PID=%d\n", getpid());
        exit(0);
    }
    else { // parent process
        printf("Parent Process: PID=%d\n", getpid());
        wait(&status); // wait for child process to terminate
        printf("Child Process Terminated with Status %d\n", status);
    }
    return 0;
}
```

```
shankar@Shankar ~> gcc -o wait wait.c
shankar@Shankar ~> ./wait
Parent Process: PID=3501
Child Process: PID=3502
Child Process Terminated with Status 0
```

❖ **close()** - used to close a file descriptor by the kernel.

```
shankar@Shankar ~> touch close.c
shankar@Shankar ~> gedit close.c
```

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

int main() {
    int fd;
    fd = open("file.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644); // create or truncate file
    if (fd == -1) { // error occurred
        fprintf(stderr, "Error Opening File");
        exit(-1);
    }
    else {
        printf("File Opened Successfully\n");
        close(fd); // close file
        printf("File Closed Successfully\n");
    }
    return 0;
}
```

```
shankar@Shankar ~> gcc -o close close.c
shankar@Shankar ~> ./close
File Opened Successfully
File Closed Successfully
```

❖ **exit()** - forcefully terminates the current program and transfers the control to the operating system to exit the program.

```
shankar@Shankar ~> touch exit.c
shankar@Shankar ~> gedit exit.c
```

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

int main() {
    printf("Before calling exit()\n");
    exit(0);
    printf("After calling exit()\n");   // This line will not be executed
    return 0;
}
```

```
shankar@Shankar ~> gcc -o exit exit.c
shankar@Shankar ~> ./exit
Before calling exit()
```