Networking Lab Assignment 1

To familiarize and understand the use and functioning of System Calls used for Operating system and network programming in Linux.

Albin Antony

4 February 2019

1 Aim

To familiarize and understand the use and functioning of System Calls used for Operating system and network programming in Linux.

2 Theory

User programs sometimes need kernel services.These services are requested by the user programs via system calls.There are mainly 5 types of system calls

- Process control
- File management
- Device management
- Information maintenance.
- Communications

Some basic system calls used for Operating System and their usage are given below.

2.1 Process Control

2.1.1 fork()

Creates a duplicate of the process.The new process is called the child process which runs concurrently with process (which process called system call fork) and this process is called parent process.They both have different PID.
Return values:
Negative- Child creation failed
Zero- Return to child process
Positive- Return to parent process

2.1.2 exit()

The exit() function immediately terminates the process.Its children are inherited by init(1).

2.1.3 wait()

The function wait() pauses the calling process until its child terminates or a signal is received.Parent resumes from wait() when the child process calls its exit().

2.2 File Manipulation

2.2.1 open()

int open(const char *pathname, int flags);
open() system call opens the file in the "pathname".Flags specify the mode ie read-only,write etc.If no file is found in the path new file is created by open().

2.2.2 read()

size_t read (int fd, void* buf, size_t cnt);
Reads cnt bytes from the file pointed by fd to the memory location addressed by buf.Return number of bytes read on success,0 on reaching end,-1 on error.

2.2.3 write()

size_t write (int fd, void* buf, size_t cnt);
Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

2.2.4 close()

int close(int fd);
Tells the OS that you are done with file.Return 0 on success and -1 on error.

2.3 Device Manipulation

2.3.1 ioctl()

The ioctl() function manipulates the underlying device parameters of special files. The argument d is a file descriptor.The second argument is a device-dependent request code. The third argument is an untyped pointer to memory.

2.3.2 read() & write()

Same system call as those in File Manipulation,the files are the systems.

2.4 Information Maintenance

2.4.1 getpid()

Used to get the process id of the calling process.Never throws error.

2.4.2 alarm()

unsigned int alarm(unsigned int sec);
Arranges for sending a SIGALRM to the process in 'sec' seconds

2.4.3 sleep()

unsigned int sleep(unsigned int sec);
Makes the calling process sleep for 'sec' seconds or until a signal arrives.

2.5 Communication

2.5.1 pipe()

Pipe is used for one way communication between processes.The standard input from one process becomes the standard output of the other.

2.5.2 shmget()

Used to create new message queues and access existing queues.It can access the shared memory used for ipc.

3 Program

3.1 File Manipulation

```
#include<stdio.h>
#include<fcntl.h>
#include<stdlib.h>
#include<errno.h>
extern int errno;
int main()
{
        int fd = open("in.txt", O_RDWR | O_CREAT);

        int fd1 = open("out.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
        int sz;
        if (fd ==-1)
        {
                printf("Error Number % d\n", errno);
        }
        else {
                char *c = (char *) calloc(100, sizeof(char));
                sz = read(fd, c,10);
                c[sz] = '\0';
                sz = write(fd1,c, 8);
                if(close(fd) == 0 && close(fd1==0)) {
                        printf("File closed\n");
                }
        }
        return 0;
}
```

Output:-

```
~$cat in.txt

hello world
yoyoyoyo

~$cat out.txt
hello wo
```

3.2 Process Control

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int frk=fork();
    wait();
    printf("Fork Value : %d \t processId : %d\n",frk,getpid());
    exit(0);
    printf("Hello World");
    return 0;
}
```

Output:-

```
~$./a.out
Fork Value : 0    pid : 7857
Fork Value : 7857         pid : 7856
```

4 Result

Implemented system calls using C and compiled using gcc 6.3.0 on Debian 4.9.0 Kerenel 4.9.0 and the above output where obtained.