SYSTEM CALLS
Network Programming Lab (CS334)

April 9, 2021

Sheen Xavier A
Roll No : 57
University Roll No : TVE18CS058
Department of Computer Science and Engineering
College of Engineering, Trivandrum

Contents

System Calls

1.1 Aim

To familiarize and understand the use and functioning of system calls used for operating system and network programming in Linux.

1.2 System Calls

When a program in user mode requires access to RAM or a hardware resource, it must ask the kernel to provide access to that resource. This is done via the so called system calls. Whenever a program makes a system call, the mode will be switched from the user mode to kernel mode. Then the kernel satisfies the program request. After that, another switch takes place which results in change of mode from kernel mode back to user mode. There are several types of system calls depending on the purpose of the call.

1.2.1 Process Control

System calls that deal with processes such as process creation, process termination etc. Most common process control system calls are fork(), exit(), wait().

fork()

Description

It is used to create processes, which becomes the child process of the caller. It takes no arguments and returns a process ID, which helps to distinguish the parent and child processes. This is one of the major methods of process creation in operating systems. When a parent process creates a child process and the execution of the parent process is

suspended until the child process executes. When the child process completes execution, the control is returned back to the parent process.

Syntax

```
#include <unistd.h>

int var_name = fork(); // Creates a child process and return a value
/* If the value is negative, then the child process creation was
   unsuccessful.
   If the value is 0, then it indicates the child process.
   If the value is positive, then the process ID of the child process
   passes to the parent. */
```

Usage

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int pid = 0;
    pid = fork();

    // Parent process
    if(pid > 0)
        printf("The Parent process has ID %d\n", getpid());

    // Child process
    if(pid == 0)
        printf("The Child process has ID %d\nThe Parent process has ID
   %d\n", getpid(), getppid());

    // Process creation error
    if(pid < 0)
        printf("Error in creating child process!\n");

    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc fork.c
sheenxavi004@Beta-Station:~$ ./a.out
The Parent process has ID 9081
The Child process has ID 9082
The Parent process has ID 9081
```

Figure 1.1: Usage of fork()

exit()

Description

It is used to end or terminate the calling process immediately. In the case of a multi threading system, it is used to terminate a single thread.

Syntax

```c
#include <stdlib.h>

exit(value) // value is the status value returned to the parent process
```

Usage

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

int main()
{
    printf("Starting the program !\n");
    printf("Exiting the program !\n");

    // When we execute this step the thread is terminated
    exit(0);

    // Won't be run
    printf("Content not meant to be printed!\n");
    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc exit.c
sheenxavi004@Beta-Station:~$ ./a.out
Starting the program !
Exiting the program !
```

Figure 1.2: Usage of exit()

wait()

Description

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. It blocks the calling process until one of its child processes exits. It takes the address of some integer variable and returns the process ID of the completed process.

Syntax

```
#include <unistd.h>

int var_name = wait(value) // Returns the process ID of the completed
   process
                        // value is the address of some integer variable
```

Usage

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

int main()
{
    if (fork()== 0)
        printf("Now we're in the child process!\n");
    else
    {
        printf("Now we're in the parent process!\n");

        wait(NULL); // Freezing the parent and moving to child process
```

```
14
15          printf("Exiting the child process!\n");
16      }
17
18      printf("End of Process!\n");
19      return 0;
20  }
```

Output

```
sheenxavi004@Beta-Station:~$ cc wait.c
sheenxavi004@Beta-Station:~$ ./a.out
Now we're in the parent process!
Now we're in the child process!
End of Process!
Exiting the child process!
End of Process!
```

Figure 1.3: Usage of wait()

1.2.2 File Manipulation

System calls which are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

open()

Description

An open() call creates a new open file description, and is used to open a file. The open() function returns a new file descriptor. Unsuccessful attempt returns -1 and sets the global variable errno to indicate the error type. The available access modes are O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_BINARY, O_TEXT. The permissions are S_IWRITE, S_IREAD, S_IWRITE | S_IREAD.

Syntax

```
1  #include <fcntl.h>
2
3  int value = open(filename, access mode, permission)
4  /* value is -1 if the file opening operation is successful
```

```
   filename is the name of the file
   access is the value corresponding to file opening mode
   permission is value corresponding to file permission */
```

Usage

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

int main()
{
    // This mode creates the file if it doesn't exist
    int ftry = open("file.c", O_RDONLY | O_CREAT);

    if(ftry == -1)
        printf("Error in opening or creating the file \n"); // Success
    else
        printf("File Descriptor Used : %d\n", ftry); // Returns file descriptor used
    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc openf.c
sheenxavi004@Beta-Station:~$ ./a.out
File Descriptor Used : 3
```

Figure 1.4: Usage of open()

close()

Description

It tells the operating system you are done with a file descriptor and closes the file which pointed by corresponding file descriptor. Successful operation returns 0 and error returns -1 as the output.

Syntax

```
#include <fcntl.h>

int val = close(fd)
/* Returns 0 if the file closing is successful
   Returns -1 if the file closing is unsuccessful
   fd is the file descriptor corresponding to the file to be closed /*
```

Usage

```
#include <stdio.h>
#include <fcntl.h>

int main()
{
    // This mode creates the file if it doesn't exist
    int ftry = open("file.c", O_RDONLY | O_CREAT);
    int closeres = -1;

    if(ftry == -1)
        printf("Error in opening or creating the file \n"); // Success
    else
        printf("File opened with descriptor : %d\n", ftry); // Returns
   file descriptor used

    // Closing the file with descriptor 'ftry'

    closeres = close(ftry);
    if(closeres == 0)
        printf("File has been successfully closed!\n");
    else
        printf("Error in closing file!\n");
    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ ./a.out
File opened with descriptor : 3
File has been successfully closed!
```

Figure 1.5: Usage of close()

read()

Description

Used to retrieve data from a file indicated by the file descriptor. The value returned is the number of bytes read (or -1 for error) and the file position is moved forward by this number.

Syntax

```
#include <fcntl.h>

val = read(fd, buf, size)
/* Returns the number of bytes read or 0 at the end of the file and -1
   if an error is encountered
   fd is the file descriptor
   buf is the data structure which contains the read contents
   size is the number of bytes to be read */
```

Usage

```
#include <stdio.h>
#include <fcntl.h>

int main()
{
    // This mode creates the file if it doesn't exist
    int ftry = open("file.txt", O_RDONLY | O_CREAT);
    int readb = 0;
    char *content = (char *)calloc(1000, sizeof(char));

    if(ftry == -1)
        printf("Error in opening or creating the file \n"); // Success
    else
```

```
14          printf("File opened with descriptor : %d\n", ftry); // Returns
    file descriptor used
15
16      // Reading and printing contents
17
18      readb = read(ftry, content, 20);
19      content[readb] = '\0';
20
21      printf("Read %d bytes. The contents are : \n%s\n",readb, content);
22
23      // Closing the file with descriptor 'ftry'
24      close(ftry);
25      printf("File closed!\n");
26      return 0;
27  }
```

Output

```
sheenxavi004@Beta-Station:~$ cc read.c -w
sheenxavi004@Beta-Station:~$ ./a.out
File opened with descriptor : 3
Read 20 bytes. The contents are :
hello this is a test
File closed!
```

Figure 1.6: Usage of read()

write()

Description

It writes specified number of bytes from a buffer to a file/device. The value returned is the number of bytes written (or -1 for error) successfully. If the number of bytes to be written is zero, write() simply returns 0 without attempting any other action.

Syntax

```
1  #include <fcntl.h>
2  int val = read(fd, buf, size)
3  /* Returns the number of bytes successfully written or 0 at the end of
       the file and -1 if an error is encountered
```

```
   fd is the file descriptor
   buf is the data structure which contains the contents to be written
   size is the number of bytes to be written */
```

Usage

```
#include <stdio.h>
#include <fcntl.h>

int main()
{
    // This mode creates the file if it doesn't exist
    int ftry = open("filer.txt", O_RDWR | O_CREAT);
    int writeb = 0;

    if(ftry == -1)
        printf("Error in opening or creating the file \n"); // Success
    else
        printf("File opened with descriptor : %d\n", ftry); // Returns
   file descriptor used

    // Writing content

    writeb = write(ftry, "Good Day!\n", 10);
    if(writeb > -1)
        printf("Content written successfully!\n");

    // Closing the file with descriptor 'ftry'
    close(ftry);
    printf("File closed!\n");
    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cat filer.txt
sheenxavi004@Beta-Station:~$ cc writef.c -w
sheenxavi004@Beta-Station:~$ ./a.out
File opened with descriptor : 3
Content written successfully!
File closed!
sheenxavi004@Beta-Station:~$ cat filer.txt
Good Day!
```

Figure 1.7: Usage of write()

1.2.3 Information Maintenance

System calls that handle information and its transfer between the operating system and the user program comes under this category.

getpid()

Description

It returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames. It never throws any errors and hence is always successful.

Syntax

```c
#include <unistd.h>

int pid = getpid(); // Returns the process ID of the calling process
```

Usage

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

int main()
{
    int pid = getpid(); // Returns the pid of the process
    printf("The Process ID is %d\n", pid);
    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc gpid.c
sheenxavi004@Beta-Station:~$ ./a.out
The Process ID is 16266
```

Figure 1.8: Usage of getpid()

alarm()

Description

The alarm() function is used to generate a SIGALRM signal after a specified amount of time elapsed. Setting a new alarm cancels the previously defined alarms. It returns the number of seconds remaining until any previously scheduled alarm was due, or zero if none.

Syntax

```c
#include <signal.h>
#include <unistd.h>

int val = alarm(t);
/* Returns the number of seconds remaining until any previously
   scheduled alarm was due, or zero if none
   t is duration for which the alarm is scheduled */
```

Usage

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void alarm_handler(int signm)
{
    printf("Beeping....!\n");
}

int main()
{
    int i;
    signal(SIGALRM, alarm_handler); // Registering signal handler
```

```
14
15      alarm(4); // Scheduled alarm after 4 seconds
16
17      for(i = 0 ; i < 8 ; i++)
18      {
19          printf("Processing...!\n");
20          sleep(1); // Sleep is executed to delay so that the alarm is
    executed
21                    // after 4th second
22      }
23      return 0;
24  }
```

Output

```
sheenxavi004@Beta-Station:~$ cc alarm.c
sheenxavi004@Beta-Station:~$ ./a.out
Processing...!
Processing...!
Processing...!
Processing...!
Beeping....!
Processing...!
Processing...!
Processing...!
Processing...!
```

Figure 1.9: Usage of alarm()

sleep()

Description

It is used to place a process in a suspended state for a specified interval of time in seconds. Expiration of the time or an interrupt causes program to resume execution.

Syntax

```
1 sleep(t); // t is the duration of time in seconds for which the process
      is suspended
```

Usage

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

int main()
{
    printf("Sleep for 2 seconds!\n");
    sleep(2); // sleep for 2 seconds

    printf("Waking Up and snoozing for another 2 seconds!\n");
    sleep(2); // sleep for 2 seconds

    printf("Finally, Woke Up!\n");
    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc alarm.c
sheenxavi004@Beta-Station:~$ ./a.out
Processing...!
Processing...!
Processing...!
Processing...!
Beeping....!
Processing...!
Processing...!
Processing...!
Processing...!
```

Figure 1.10: Usage of sleep()

1.2.4 Communication

System calls that are useful for inter-process communication. They also deal with creating and deleting a communication connection.

pipe()

Description

A pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. It facilitates inter-process communication where one process can write to and another related process can read from a so called 'virtual file'. It returns -1 on error.

Syntax

```
#include <unistd.h>

int val = pipe(flow);
/* Return -1 on error otherwise it returns two file descriptors in flow
   Flow is a array of length 2 which contains the two file descriptors
   on successful execution of pipe */
```

Usage

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

int main()
{
    int flow[2];
    char buff[18];

    // Creating the pipe for flow of data
    if(pipe(flow) == -1)
    {
        printf("Error in piping!\n");
        exit(0);
    }

```

```c
    // Data to be inserted into the pipe
    char *data = "Wings of Freedom\n";

    // Inserting data into the pipe at write end
    printf("Inserting data into pipe at one end...\n");
    write(flow[1], data, 18);
    printf("Finished inserting data!\n");

    // Fetching data previously inserted into the pipe from read end
    printf("Fetching data from the pipe at the other end...\n");
    read(flow[0], buff, 18);
    printf("Finished fetching data!\n");

    // Aquired data
    printf("Data from the pipe: %s\n", buff);

    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc pipe.c
sheenxavi004@Beta-Station:~$ ./a.out
Inserting data into pipe at one end...
Finished inserting data!
Fetching data from the pipe at the other end...
Finished fetching data!
Data from the pipe: Wings of Freedom
```

Figure 1.11: Usage of pipe()

shmget()

Description

It requests for a shared memory segment. Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process. On a successful operation, its return value is the shared memory ID, otherwise, the return value is negative.

Syntax

```c
#include <sys/ipc.h>
#include <sys/shm.h>

int val = shmget(skey, ssize, shmflg);
/* skey is the key for the segment
   ssize is the size of the segment
   shmflg is the create/use flag  */
```

Usage

```c

// Writer Process

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
    // ftok to generate unique key
    int key = ftok("shmfile", 65);

    // shmegt returns an ID which is sid
    int sid = shmget(key, 1024, 0666 | IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(sid, (void*)0, 0);
    printf("Data to be written : ");
    scanf("%[^\n]", str);

    // shmdt to detach from shared memory
    shmdt(str);

    return 0;
}

// Reader Process

#include <stdio.h>
#include <sys/ipc.h>
```

```c
#include <sys/shm.h>

int main()
{
    // ftok to generate unique key
    int key = ftok("shmfile", 65);

    // shmegt returns an ID which is sid
    int sid = shmget(key, 1024, 0666 | IPC_CREAT);

    // shmat to attach to shared memory
    char *str = (char*) shmat(sid, (void*)0, 0);

    // Data retrieved
    printf("Data read from memory : %s\n", str);

    // shmdt to detach from shared memory
    shmdt(str);

    //destroy the shared memory
    shmctl(sid, IPC_RMID, NULL);

    return 0;
}
```

Output

```
sheenxavi004@Beta-Station:~$ cc writer.c
sheenxavi004@Beta-Station:~$ ./a.out
Data to be written : Wings of Freedom
sheenxavi004@Beta-Station:~$ cc reader.c
sheenxavi004@Beta-Station:~$ ./a.out
Data read from memory : Wings of Freedom
```

Figure 1.12: Usage of shmget()

1.2.5 Device Manipulation

System calls that are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

ioctl()

Description

Input/Output Control(ioctl) is a device-specific system call used for terminal I/O, hardware device control, and kernel extension operations. Successful operation returns 0 and error returns -1.

Syntax

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

int val = ioctl(fd, request, ...);
/* Returns 0 in case of a successful operation and error returns -1
   fd is the file descriptor
   request is a device dependant request code */
```

1.3 Result

The use and functioning of various system calls were studied. They were implemented in C programs and their corresponding outputs were verified.