

## **Lab 6: Pipe System Call**

A user programme can interact with the operating system using a system call. A number of services are requested by the programme, and the OS replies by launching a number of system calls to fulfill the request. A system call can be written in high-level languages like C or Pascal or in assembly code. If a high-level language is employed, the operating system may directly invoke system calls, which are predefined functions.

The operating system's features are linked to user programmes using the Application Programme Interface (API). It serves as a conduit between a process and the operating system, enabling user-level programmes to ask for operating system services. System calls are the only means of communication with the kernel system. Any programme that uses resources must use system calls.

One of the most prevalent system calls is the pipe system call. Pipe is a communication medium between two or more related or interrelated processes. It can be either within one process or a communication between the child and the parent processes. Communication can also be multi-level such as communication between the parent, the child, and the grand-child, etc. Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.

Pipe mechanism can be viewed with a real-time scenario such as filling water with the pipe into some container, say a bucket, and someone retrieving it, say with a mug. The filling process is nothing but writing into the pipe and the reading process is nothing but retrieving from the pipe. This implies that one output (water) is input for the other (bucket).

### **Pipe() call**

The pipe() system function is used to open file descriptors, which are used to communicate between different Linux processes. It creates a pipe, a unidirectional data channel that can be used for inter-process communication. The array *pipefd* is used to return two file descriptors referring to the ends of the pipe. *pipefd[0]* refers to the read end of the pipe. *pipefd[1]* refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the rear end of the pipe.

### **Syntax:**

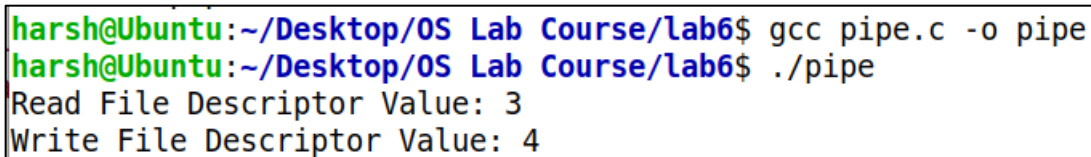
```
int pipe(int pipefd[2]);
```

Here, the pipe() function creates a unidirectional data channel for inter-process communication. You pass in an int (Integer) type array *pipefd* consisting of 2 array elements to the function *pipe()*. Then the pipe() function creates two file descriptors in the *pipefd* array.

The first element of the *pipefd* array, *pipefd[0]* is used for reading data from the pipe. The second element of the *pipefd* array, *pipefd[1]* is used for writing data to the pipe. On success, the *pipe()* function returns 0. If an error occurs during pipe initialization, then the *pipe()* function returns -1. The pipe() function is defined in the header *unistd.h*. In order to use the *pipe()* function in C program, the header *unistd.h* must be included.

## Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
    int pipefds[2];
    if(pipe(pipefds) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    printf("Read File Descriptor Value: %d\n", pipefds[0]);
    printf("Write File Descriptor Value: %d\n", pipefds[1]);
    return EXIT_SUCCESS;
}
```



```
harsh@Ubuntu:~/Desktop/OS Lab Course/lab6$ gcc pipe.c -o pipe
harsh@Ubuntu:~/Desktop/OS Lab Course/lab6$ ./pipe
Read File Descriptor Value: 3
Write File Descriptor Value: 4
```

The following example illustrates the use of pipe for inter-process communication. A PIN is sent from the child process to the parent process using a pipe. It is then read the PIN from the pipe in the parent process and printed it from the parent process. The code is shown below:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(void)
{
    int pipefds[2];
    char buffer[5];
    if (pipe(pipefds) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    char *pin = "4128\0";
    printf("Writing PIN to pipe\n");
    write(pipefds[1], pin, 5);
    printf("Done\n\n");
    printf("Reading PIN from pipe\n");
```

```
    read(pipefds[0], buffer, 5);  
    printf("Done\n\n");  
    printf("PIN from pipe: %s\n", buffer);  
    return EXIT_SUCCESS;  
}
```

```
harsh@Ubuntu:~/Desktop/OS Lab Course/lab6$ gcc pipe.c -o pipe  
harsh@Ubuntu:~/Desktop/OS Lab Course/lab6$ ./pipe  
Writing PIN to pipe  
Done  
  
Reading PIN from pipe  
Done  
  
PIN from pipe: 4128
```