

Communicate Between Related Processes Through Pipe

Part-II

In the second part of this series on communication between related processes through pipe, we will look into their basics and working procedure.

The term 'pipe' means connecting or sending data, which is flown from one process to another. A pipe can be created by executing the *pipe* system call.

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

You can declare an array of two file descriptors and pass the pointer to the array *fd* as an argument of the *pipe* system call. If a pipe is executed successfully, it returns 0 and creates two file descriptors, namely *fd[0]* and *fd[1]*. If a pipe fails, it returns with -1 and the corresponding errors are:

EMFILE—if the process has too many open files or,

ENFILE—if there are too many open files in the system.

As the name suggests, *fd[0]* (standard input 0) is for reading and *fd[1]* is for writing (standard output 1). The integer values 3 and 4 are given to the pipe descriptors of *fd[0]* and *fd[1]*, respectively, if any other file is not open before calling the *pipe* system call.

Figure 1 is a simple example of the pipe's descriptors. The pipe function is declared in *unistd.h*. When a process calls *read()* from a pipe, the *read()* returns with data immediately, if the pipe is not empty. If the pipe is empty then the read call is blocked until some process writes to the pipe—as long as some process has the pipe open for writing. If no process has the pipe open for writing, the read returns 0. On the other hand, if a process call writes to the pipe but if no process has opened the pipe for reading, then write returns with the *EPIPE* error, creating a *SIGPIPE* signal and displays the famous "broken pipe" message. A pipe can be imagined as the circular buffer—the data enters from one end, and the process reads the data from the pipe and gets the data on the other end in FIFO order.

The blocking nature of read and write calls effectively synchronises the processes. The usual implementation of pipes uses the file system for data storage and virtual file system objects, since it doesn't associate any named device. So, it is temporary, in the sense that the pipe descriptors are valid as long as the pipe is open. During pipe creation, the kernel assigns an *inode* and allocates a pair of user file descriptors, a data block and corresponding file table entries. Since it uses the file table, the interfaces for the I/O related system calls, like read, write and *ioctl*, are consistent with the interface for regular files. The *inode* records pipe's byte offset, which shows that the pipe will point to the next read/write. This, in turn, helps convenient FIFO access to the pipe. This is different from regular files, where the offset is maintained in the file table. Reading and writing to the pipe is an atomic operation if the data size is less than or equal to the *PIPE_BUF*, which is usually a data block size (generally 4 KB, but it is architecture-

dependent on the size of a memory page). If the data size is more than the system limit, then the read/write processes are not atomic. If no space is available for writing, the write process is blocked until some process reads from the pipe.

To explain the pipe creation from the system call point of view, the kernel creates an *inode* object and two file objects (reading/writing). The *Pipe* system call is serviced by the *sys_pipe* function, which in turn calls the *do_pipe* function. The *do_pipe* function creates a file object and file descriptors for the read and write channel, and sets the *O_RDONLY* and *O_WRONLY* flags correspondingly. Then the *do_pipe* function invokes the *get_pipe_inode* function, which initialises an *inode* object for the pipe. At last, if everything is fine, the *pipe* system call returns two pipe descriptors to the user process. The curious reader can get more information about the working procedure of a pipe from the source code of *pipe.c*. The *file_operations* structure of a pipe is as shown below...

```
struct file_operations rdwr_pipe_fops = {
    .llseek      = no_llseek,
    .read        = pipe_read,
    .readv       = pipe_readv,
    .write       = pipe_write,
    .writev      = pipe_writev,
    .poll        = pipe_poll,
    .ioctl       = pipe_ioctl,
    .open        = pipe_rdwr_open,
    .release     = pipe_rdwr_release,
    .fsync       = pipe_rdwr_fsync,
};
from /usr/src/linux-2.6.8/fs/pipe.c
```

From this structure, we can use open, close, read, write, poll and ioctl calls to manipulate the pipe. Since a pipe doesn't have any name, an open system call can be used only in a named pipe. The *file_operations* structure is also used in *fifo.c* to do operations on FIFOs. From the structure, it is evident that we can't access the data randomly since the *lseek* (no_llseek) function is not implemented.

Working with a pipe

Figure 2 explains how to pass data using a pipe within a process, but here the usage is very limited. In a general scenario, after a pipe creation, the process executes the *fork* system call and the child process inherits the pipe descriptors. Now we have two processes, parent and child, one pipe but a pair (fd[0] and fd[1]) of descriptors for each process. Even though we have two pairs, we can't use them for two-way communications since a pipe allows only a unidirectional flow of data.

Figure 3 shows the screen dump of *ls -Rl | wc -l* processes file descriptors. The output of the *ls -Rl* program is fed into the input of the *wc -l* program

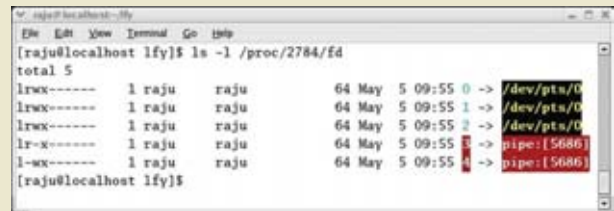


Figure 1: A simple example of pipe's descriptors



Figure 2: Program to communicate within a process

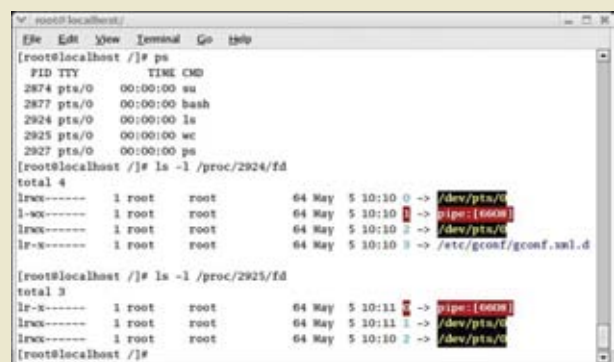


Figure 3: Screen dump of 'ls -Rl | wc -l' processes standard and pipe's descriptors

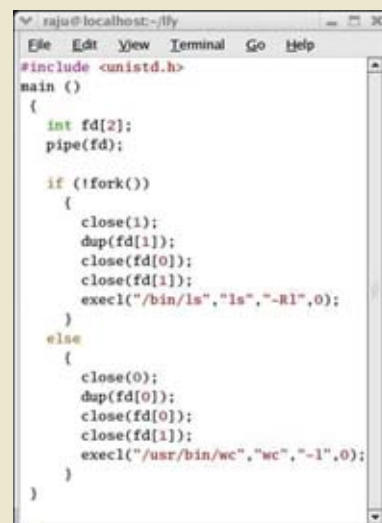


Figure 4: Program to execute 'ls -Rl | wc -l'

```

root@localhost:/home/lfy
File Edit View Terminal Go Help
/* This program explains two way communication
 * between parent and child
 */

#include <unistd.h>

main ()
{
    int fd1[2], fd2[2];
    char buf1[80], buf2[80];
    pipe(fd1);
    pipe(fd2);

    if (!fork())
    {
        close(fd1[0]);
        close(fd2[1]);
        write(fd1[1], "I am a Child", 13);
        read(fd2[0], buf2, sizeof(buf2));
        printf (" Message from Parent: %s\n", buf2);
    }
    else
    {
        close(fd1[1]);
        close(fd2[0]);
        read(fd1[0], buf1, sizeof(buf1));
        printf (" Message from child: %s\n", buf1);
        write(fd2[1], "I am a parent", 14);
        wait(0);
    }
}
13,0-1 All

```

Figure 5: Program for bi-directional communication between a parent and child

```

root@localhost:/home/lfy
File Edit View Terminal Go H
/* This program explains how to execute
 * ls -Rl | grep ^d | wc -l
 */

#include <unistd.h>
main () {
    int fdl[2], fd2[2];
    pipe(fdl);
    pipe(fd2);

    if (!fork()) {
        close(1);
        dup (fdl[1]);
        close(fdl[0]);
        close(fd2[0]);
        close(fd2[1]);
        execl("/bin/ls", "ls", "-Rl", 0);
    }
    else {
        if (!fork()) {
            close(0);
            dup(fdl[0]);
            close(1);
            dup(fd2[1]);
            close(fdl[1]);
            close(fd2[0]);
            execl("/bin/grep", "grep", "^d", 0);
        }
        else
        {
            close(0);
            dup(fd2[0]);
            close(fdl[0]);
            close(fdl[1]);
            close(fd2[1]);
            execl("/usr/bin/wc", "wc", "-l", 0);
        }
    }
}
6,1 All

```

Figure 6: Program to execute 'ls -Rl | grep ^d | wc -l'

through the pipe. How is this possible? *ls -Rl* and *wc -l* programs have separate file descriptor tables, and once the pipe is created, it has two descriptor values—3 and 4, for reading and writing, respectively. First, the *ls -Rl* program closes the standard output. We know that if a *dup* is called after the closing standard output, the kernel allocates the minimum possible value from the file descriptor table, which is nothing but the standard output as the new descriptor. First, close the standard output and *dup* the write end of the pipe of the parent process. So the write end of the parent process gets the standard output descriptor. As a consequence of this, the process writes to the write end of the pipe. In the child process, close the read end of the pipe and duplicate the standard input. So the process, instead of reading data from the standard input (say keyboard), will read it from the read end of the pipe. However, we haven't closed the standard output of the child process, so the output of the *ls -Rl | wc -l* is displayed on the monitor.

This can be further explained in Figure 4 through a program. Figure 5 explains how to perform two-way communication between a child and parent. Figure 6 explains how to execute '*ls -Rl | grep ^d | wc -l*' and it also tells how many total directories we have from the given directory. Finally, how many pipes are open within a single process can be found by executing a program shown in Figure 7.

Working with the *popen* library function

The easiest way to pass data between two programs is by using the *popen* and *pclose* library functions. The syntax for *popen* and *pclose* function is as follows.

```

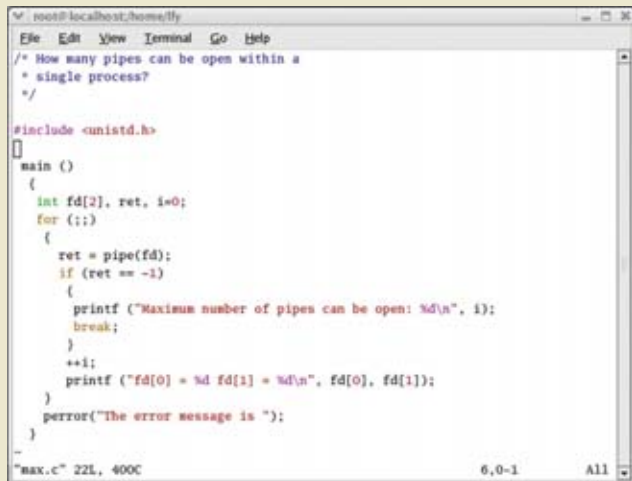
FILE *popen (const char *command, const char *type);
int pclose (FILE *stream);

```

In *FILE *stream*, *FILE* is the data type used to represent stream objects. The *FILE* object holds all the information about the file. It can be managed by the I/O library function. Streams provide a higher level interface, which is layered on top of the low-level file descriptor interface. Streams provide a powerful formatted I/O function.

The *popen* and *pclose* function is closely related to the *system ()* function. The *system* function gives a simple, portable mechanism to run a given executable program or shell script. It executes all the steps like *fork*, *dup* and *exec* automatically.

Figure 8 shows the usage of the *system ()* function. The advantage of the *system* function is its ease of use. But the user can't get as much control in it as the low level functions; for example, the pipe and user have to wait until the sub-program terminates before



```

root@localhost/home/lfy
File Edit View Terminal Go Help
/* How many pipes can be open within a
 * single process?
 */

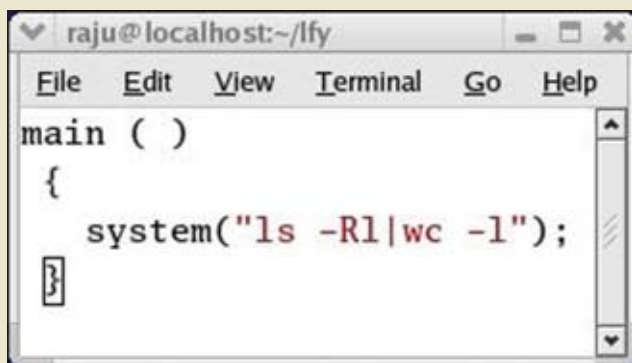
#include <unistd.h>

main ()
{
    int fd[2], ret, i=0;
    for (;;)
    {
        ret = pipe(fd);
        if (ret == -1)
        {
            printf ("Maximum number of pipes can be open: %d\n", i);
            break;
        }
        ++i;
        printf ("fd[0] = %d fd[1] = %d\n", fd[0], fd[1]);
        perror("The error message is ");
    }
}

"max.c" 22L, 400C
6,0-1 All

```

Figure 7: Program to find maximum number of pipes can be open within a process



```

raju@localhost:~/lfy
File Edit View Terminal Go Help

main ( )
{
    system("ls -Rl|wc -l");
}

```

Figure 8: Program to implement system () function



```

root@localhost/home/lfy
File Edit View Terminal Go Help

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

main ()
{
    FILE *output;
    output = popen("ls -Rl|wc -l", "w");
    pclose(output);
}

3,0-1 Top

```

Figure 9: Program to execute 'ls -Rl | wc -l' through popen function

doing anything. The syntax for the system command is shown below.

```
system (const char *command);
```

The system function executes it as a shell command. It searches the file in the PATH environment variable to find programs to execute. If it is successful, it returns the status of the shell, otherwise it returns -1 for error. It is declared in *stdlib.h*.

The *popen* function opens a process by creating a pipe, forking and invoking the shell. The *popen* function creates a pipe to the sub-process and returns a stream that corresponds to that pipe. It allows the parent process to communicate with the standard input and output. There are only two types of mode arguments—either 'r' or 'w' (for reading/writing to the pipe). If the mode is 'r', the standard output is stored into the stream object that can be read by the user. If the mode is 'w', the user can write to the stream to send data to the standard input of the sub process. If *popen* is failed due to failure of the fork creation or a problem with the stream or a wrong executable file, then it returns with the NULL pointer. Once the created sub process is completed, then the *pclose* call will be returned. If *pclose* is called before completion of the associated process, the *pclose* will wait for the process to finish. Figure 9 shows the usage of *popen* and *pclose* library functions.

The working procedure of the *popen* call is that it runs the executable program by invoking the shell, and then sends the command as an argument. The *popen* call not only invokes the executable program but also a shell, which is expensive. Even though the system or *popen* does the job of a pipe, if we want to get more control over a program, then we have to use the low level routine like a pipe.

In this concluding part of the article on communications related to pipes, we covered the usage of the system and *popen* functions. Since a pipe is anonymous, i.e., it is not associated with any storage device, we can't use a pipe between unrelated processes. The other available form of pipes is called 'named' pipes, and these are created in the file system. So they are visible to any process and can be opened by one process for reading and by another process for writing. In the next article, we will see how to communicate between unrelated processes within a system through a named pipe (FIFO). **LFY**

The author would like to thank T. Anandraj, Embedded & Product Engineering Solutions, Wipro Technologies, for his critical reading of the manuscript.

By: Dr B. Thangaraju is working as a tech lead in Embedded and Product Engineering Solutions (E&PE), Wipro Technologies, Bangalore.