Communication Between Independent Processes by FIFO

Let's peek into the secrets of communication between independent processes by the First-In-First-Out method!

FIFO (First-In-First-Out) is a form of interprocess communication which can be used in a single host and is often referred to as a named pipe. Even though pipes are simple, flexible and efficient in communication, they have a major drawback in their inability to communicate with unrelated processes. A named pipe solves this problem and is generally used to communicate between unrelated processes. FIFO is a named file of a special type and is visibly stored in the file system for independent processes to communicate. Linux treats FIFO like a file of zero size with owner and access permissions. Once a FIFO is created, it can be opened, closed, read, written or unlinked like any other regular file.

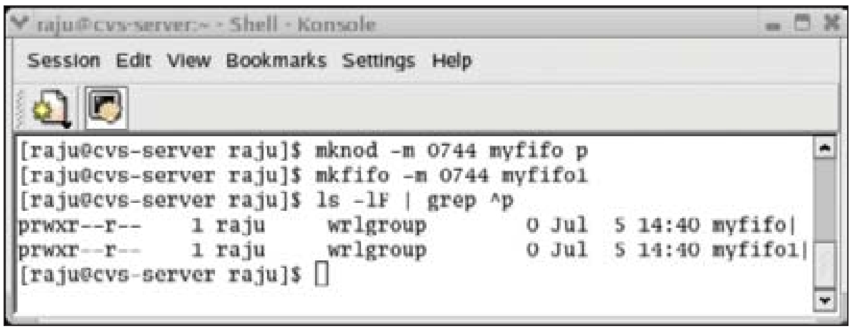
Once created, a FIFO exists till it is explicitly deleted, whereas a pipe can persist until the last process closes it. FIFO is less secure than a pipe because only a related process can use the pipe, while any process with permission to access FIFO can open it for communication. FIFO also consumes more resources than a pipe.

FIFO is similar to a device file. It has a disk inode (derived from Index NODE) and uses a kernel buffer to store data temporarily. The use of the kernel buffer makes FIFO much more efficient than using a temporary file. The inode has the same specific components for pipes as FIFO has. FIFO is used to duplicate an output stream in a series of shell commands, which avoids writing data to a temporary file. FIFO can be opened for multiple processes like reading or writing. During data transition between processes through itself, the kernel sends all the data internally without writing it to the file system. So FIFO has no content saved on the file system—the file system serves as a reference point so that any process can access the pipe using a name in the file system, and the size of the FIFO file is always zero. Before data can be passed, FIFOs at both the ends should be open. Remember, opening FIFO only at one end blocks till the other end is open.

The difference between FIFO and a regular file is that the data is transient in the former. The data cannot be read from FIFO, once it is already read. Even though FIFO can be used to communicate between unrelated processes, it is restricted only to the local file system. FIFO cannot be used on a NFS (Network File System) mounted file system since the communication between two different hosts needs a socket program.

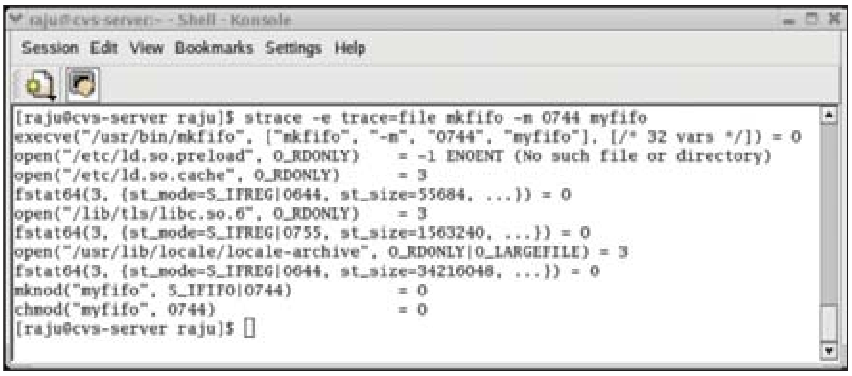
Ways of creating FIFO

A pipe can be created and opened with a single pipe system call, whereas FIFO is a special file and has to be handled like any regular file. First, it has to be created by an appropriate function (library or system call) or shell command and then opened by a process like reading or writing. Linux provides various ways to create



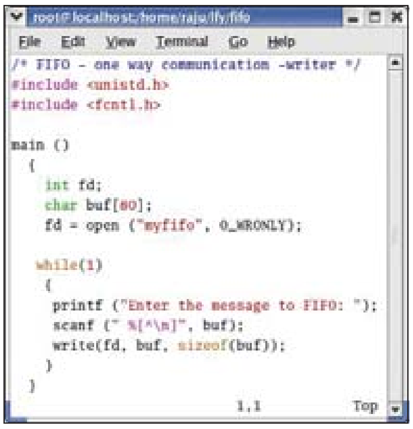
```
[raju@cvs-server raju]$ mknod -m 0744 myfifo p
[raju@cvs-server raju]$ mkfifo -m 0744 myfifo1
[raju@cvs-server raju]$ ls -lF | grep ^p
prwxr--r--    1 raju     wrlgroup        0 Jul  5 14:40 myfifo|
prwxr--r--    1 raju     wrlgroup        0 Jul  5 14:40 myfifo1|
[raju@cvs-server raju]$ 
```

Figure 1: Screen shot of FIFO creation



```
[raju@cvs-server raju]$ strace -e trace=file mkfifo -m 0744 myfifo
execve("/usr/bin/mkfifo", ["mkfifo", "-m", "0744", "myfifo"], [/* 32 vars */]) = 0
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=55684, ...}) = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
fstat64(3, {st_mode=S_IFREG|0755, st_size=1563240, ...}) = 0
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=34216048, ...}) = 0
mknod("myfifo", S_IFIFO|0744)           = 0
chmod("myfifo", 0744)                   = 0
[raju@cvs-server raju]$ 
```

Figure 2: Screen shot of *mknod* system call trace by *strace* tool

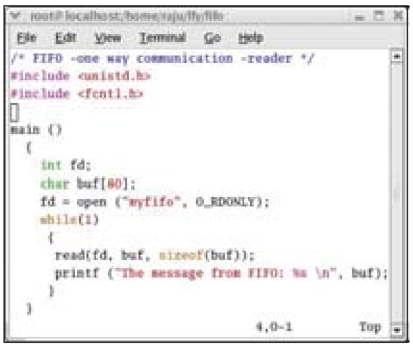


```
/* FIFO - one way communication -writer */
#include <unistd.h>
#include <fcntl.h>

main ()
 {
   int fd;
   char buf[80];
   fd = open ("myfifo", O_WRONLY);

  while(1)
   {
    printf ("Enter the message to FIFO: ");
    scanf (" %[^\n]", buf);
    write(fd, buf, sizeof(buf));
   }
 }
                              1,1          Top
```

Figure 3 (a): Program for one-way communication (writer)



```
/* FIFO -one way communication -reader */
#include <unistd.h>
#include <fcntl.h>

main ()
 {
   int fd;
   char buf[80];
   fd = open ("myfifo", O_RDONLY);
   while(1)
    {
     read(fd, buf, sizeof(buf));
     printf ("The message from FIFO: %s \n", buf);
    }
 }
                              4,0-1        Top
```

Figure 3 (b): Program for one-way communication (reader)

FIFO as we will discover in the following sections of this article.

Shell commands

mknod and *mkfifo* are used either as commands in a shell prompt or as a library function and a system call, respectively, in a C program to create FIFO. First, let us analyse how to create FIFO through the command-line.

$mkfifo m 744 myfifo—This will create a FIFO file named as *myfifo* with access permission 744 (read, write and execute permission to the owner and only read permission to the group and others). *mkfifo* is a special command, which is used only to create a FIFO.

$mknod m 744 myfifo p—This will create a FIFO file as *myfifo* with the specified access permission. In general, *mknod* command is used to create a special device file. The syntax for mknod is...

```
mknod [OPTION]... NAME TYPE [MAJOR MINOR]
```

mknod is a command, OPTION is used to specify access permission, NAME is name of a FIFO, TYPE is the type of a device-specific file-like character (c), block (b) or FIFO (p), and MAJOR MINOR is used to specify the major and minor number of a device, correspondingly. Being a pseudo device, the major and minor number are not needed in FIFO. Figure 1 shows the creation of a FIFO through the command-line in a shell using *mkfifo* and *mknod*. Long listing of files with an *F* option shows *prwxr--r--*, where *p* is the type of a file (here it is a pipe) and the | symbol at the end of the file name indicates that *myfifo* is a named pipe.

Library function

mkfifo can be used as a Library function in a C program to create a FIFO. To use this function, we need to include the following two header files: *<sys/types.h>* and *<sys/stat.h>*. The syntax for the function is...

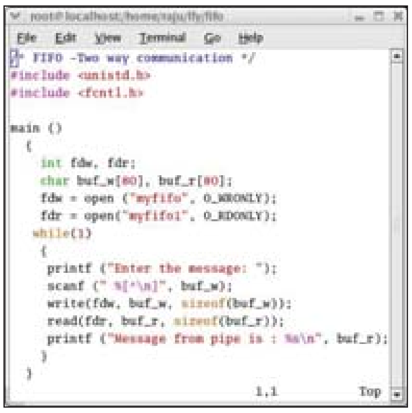
```
int mkfifo(const char *pathname, mode_t
mode);
```

The arguments, *const char *pathname* specifies the name of FIFO and *mode_t mode* specifies file access permission. On the successful creation of a FIFO, the *mkfifo* function returns 0 else -1 and errno is set appropriately.

System call

```
int mknod(const char *pathname, mode_t
mode, dev_t dev);
```

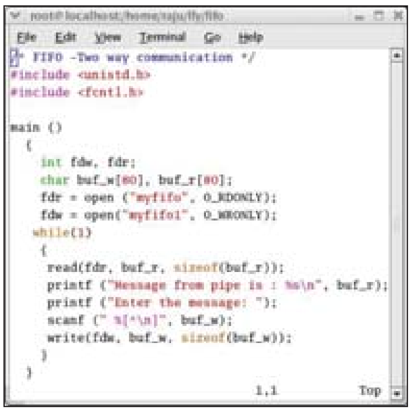
mknod system call creates a device special file, ordinary file or named pipe with the name as specified in a pathname argument, with attributes specified by the arguments *mode* and *dev*. To use the *mknod* system call, we need to include the following four header files; *<sys/types.h>*, *<sys/stat.h>*, *<fcntl.h>* and *<unistd.h>*. In the mode argument, we need to specify both the access permission and file type with bitwise OR. The file type must



```
/* FIFO -Two way communication */
#include <unistd.h>
#include <fcntl.h>

main ()
  {
    int fdw, fdr;
    char buf_w[80], buf_r[80];
    fdw = open ("myfifo", O_WRONLY);
    fdr = open("myfifo1", O_RDONLY);
   while(1)
    {
     printf ("Enter the message: ");
     scanf (" %[^\n]", buf_w);
     write(fdw, buf_w, sizeof(buf_w));
     read(fdr, buf_r, sizeof(buf_r));
     printf ("Message from pipe is : %s\n", buf_r);
    }
  }
```

Figure 4 (a): Program for two-way communication (writer)



```
/* FIFO -Two way communication */
#include <unistd.h>
#include <fcntl.h>

main ()
  {
    int fdw, fdr;
    char buf_w[80], buf_r[80];
    fdr = open ("myfifo", O_RDONLY);
    fdw = open("myfifo1", O_WRONLY);
   while(1)
    {
     read(fdr, buf_r, sizeof(buf_r));
     printf ("Message from pipe is : %s\n", buf_r);
     printf ("Enter the message: ");
     scanf (" %[^\n]", buf_w);
     write(fdw, buf_w, sizeof(buf_w));
    }
  }
```

Figure 4 (b): Program for two-way communication (reader)

be either one of *S_IFREG* (or zero file type), *S_IFCHR, S_IFBLK, S_IFIFO* or *S_IFSOCK* to specify a normal file, character special file, block special file, FIFO (named pipe), or the Unix domain socket, respectively. If the file type is a block or a character device file, then *dev* specifies the major and minor number of the newly created device and for other files, zero is passed. On successful execution of *mknod,* the system call returns zero else -1 for an error and sets errno appropriately.

The following *mknod* system calls illustrate different ways of creating FIFO files in a C program.

```
mknod (myfifo, S_IFIFO|0744, 0);
```

Usual way to call *mknod.*

(or)

```
mknod (myfifo, 0x1000|0744, 0);
```

Instead of *S_IFIFO,* we can use 0x1000.

(or)

```
mknod (myfifo, 010744, 0);
```

Here 010 specifies a named pipe and 744 specifies access permission.

Actually, the *mkfifo* command internally calls the *mknod* system call. This can be viewed through a *strace* tool. In general, the *strace* command is used to trace system calls and signals. *strace* is a useful debugging and diagnostic tool. It runs till the specified command exits, and gives details of each system call, its arguments, return values, etc. This tool is useful for a programmer to find information about system calls and signals, since that can happen at the user and kernel interface. So a careful examination of this will be helpful for bug isolation, sanity checking and capturing race conditions. Figure 2 shows the execution of *strace -e trace=file mkfifo -m 0744 myfifo***.**

The *e* option modifies the events to trace. Here, the trace file, and the remaining commands and options are already explained. The output clearly shows the *mkfifo* command inside it calling the *mknod* system calls and the *m* option calls *chmod* system calls. Once the FIFO file is created, it can be opened by any process for reading or writing. The communication between different processes is explained in the next section.

Communication through FIFO

After creating a FIFO file in one window, you can open a *myfifo* file for reading by using the *cat < myfifo* command and in another window, you can open *myfifo* for writing by using the *cat > myfifo* command. Then, write some text and hit *ENTER.* The text will be sent through *myfifo* and displayed on to the first window. If you want to close the file, hit CTRL D in the second (writer) window, which will automatically close the first window also. If you want to close the reader first, then you need to kill the process by hitting CTRL C. This time, the writer will not close automatically. Once you hit ENTER, it will receive a 'broken pipe' signal since there is no reader.

In a C program, a FIFO file can be opened either by a system (low-level I/O functions) call like *open, write, read, close* or by using C library functions like *fopen, fscanf, fprintf* and *fclose.* Since FIFO is a one-way communication channel, the opening of a FIFO file should be either *read* or *write.* If the file is open for *read* or *write* modes, then the result is unspecified. The difference between a regular file and FIFO is that both the *read* and *write* ends of a FIFO file should be open before it starts communication. For a FIFO file, the kernel stores the *read* and *write* offsets in the inode instead of in the file table, so that processes can share their values. This sharing is not possible if the kernel keeps the offset information in the file table, since a process gets a new file table entry for each open system call.

Opening a FIFO file for *read* or *write* is possible both in blocking and non-blocking modes. When a file is open, we can specify the O-NONBLOCK mode, along with the opening mode either as *O_RDONLY* or *O_WRONLY.* The non-blocking mode brings a process into existence without waiting for the other end to be ready. If a file is already open, then the file descriptor can be set to non-blocking mode using the *fcntl* system call. For example, if we want to set the non-block mode of an unnamed pipe, then we can implement it with the following instructions...

```
int mode;
mode = fcntl(fd, F_GETFL, 0);
mode |= O_NONBLOCK;
fcntl (fd, F_SETFL, mode);
```

For more information about the arguments and non-blocking handling of a FIFO, please refer to *man 2 fcntl* and *man 4 fifo*.

By default, FIFO is open in the blocking mode to synchronise *read* and *write*. When a process wants to write and there is no reader, then the process receives the SIGPIPE signal, which terminates the process. This is true when both the *write* and *read* are open and suddenly the *read* end is closed—then the *write* process will receive the SIGPIPE signal. Figures 3a and 3b show how to communicate between independent processes in one direction with the *myfifo* file. Figure 3(a) can be executed in one window and Figure 3(b) can be executed in another window. If we send some text from the first window, the text is received from the second window and displayed. This program is written with an infinite loop so that when we want to close the communication, pressing CTRL C in the first window will automatically close the second window. One of the main drawbacks of a pipe (both named and unnamed) is half-duplex, but if we want to communicate bi-directionally then we need to use two FIFOs—one for sending data and another one to receive data. Figures 4a and 4b demonstrate two-way communication by two FIFOs. The algorithm for *open*, *read* and *write* for a FIFO is the same as a regular file, but it has some differences like the kernel incrementing the size of the pipe after every *write* and decrementing after every *read*. In a regular file, the kernel increments the file size when it wants to write beyond the end of a file.

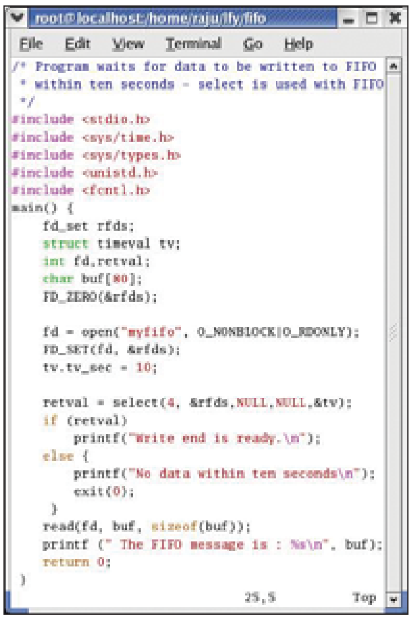
The blocking mode makes the processes wait for a long time till the other end responds. For example, a reader will wait until a writer writes some data into the FIFO. In some situations, we need to wait for a specific time period to get data—the *select* system call is used to perform this. Writing a FIFO is an atomic operation if the data size is equal to or lower than the pipes capacity. In the next section, the implementation of *select* system call with FIFO and some of the system limitations are explained.

Select and system limitations

***Select* system call:** *Select* is a system call used to handle more than one file descriptor in an efficient manner. Its main arguments are three arrays of file descriptors, namely, *readfds*, *writefds* and *exceptfds*. The syntax for the select system call is:

```
int select (int n, fd_set *readfds,
fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

The *select* call is generally used to block while waiting for a change of status on any one or more of the file descriptors. The argument 'n' is the highest-numbered descriptor in any of the three sets, plus 1. *fd_set* is the file descriptor set, which has the arrays of file descriptors. Its contents are declared with the *FD_CLR*, *FD_ISSET*, *FD_SET* and *FD_ZERO* macros. The individual file descriptors, which we are interested in, should be included in a newly declared *FD_SET*. *readfds* is watched if any data is available for reading

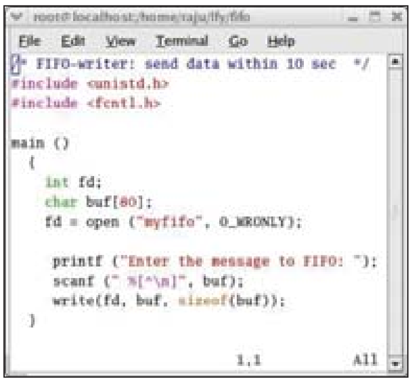

root@localhost:/home/raju/lfy/fifo
File Edit View Terminal Go Help

```
/* Program waits for data to be written to FIFO
 * within ten seconds - select is used with FIFO
 */
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
main() {
    fd_set rfds;
    struct timeval tv;
    int fd,retval;
    char buf[80];
    FD_ZERO(&rfds);

    fd = open("myfifo", O_NONBLOCK|O_RDONLY);
    FD_SET(fd, &rfds);
    tv.tv_sec = 10;

    retval = select(4, &rfds,NULL,NULL,&tv);
    if (retval)
        printf("Write end is ready.\n");
    else {
        printf("No data within ten seconds\n");
        exit(0);
     }
    read(fd, buf, sizeof(buf));
    printf (" The FIFO message is : %s\n", buf);
    return 0;
 }
                                  25,5          Top
```

Figure 5 (a): Program for using *select* to wait for a reader


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

```
/* FIFO-writer: send data within 10 sec   */
#include <unistd.h>
#include <fcntl.h>

main ()
  {
    int fd;
    char buf[80];
    fd = open ("myfifo", O_WRONLY);

     printf ("Enter the message to FIFO: ");
     scanf (" %[^\n]", buf);
     write(fd, buf, sizeof(buf));
  }
                                  1,1           All
```

Figure 5 (b): Program for writer

from any of its file descriptors, *writefds* is watched if there is space to write data to any of its file descriptors and *exceptfds* is watched for exceptions or errors on any of the file descriptors. The timeout argument is used to wait before returning, even if nothing interesting happens or if this value is NULL, and then the *select* call blocks indefinitely, waiting for an event.

Figure 5 explains the implementation of the *select* system call with FIFO. In this program, the reader will wait for 10 seconds. The waiting is declared in the *select* system call as a timeout argument. If any writer writes data to the FIFO within 10 seconds, the reader reads the data from the FIFO and displays it, or else it returns with error. There is one more *select* call, which is *pselect*. This is used if we are waiting for a signal as well as data from a descriptor. You can get more information about *select* and *pselect* from *man 2 select*, *man 2 select_tut* and *man 2 pselect*.

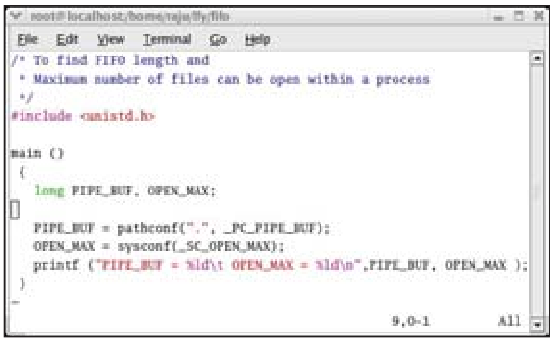
System limitations: The pipe (both named and unnamed) size is declared as *PIPE_BUF*, which is usually the size of a data block of 4KB. Linux has imposed two system limitations on a pipe—*OPEN_MAX*, the maximum number of files that can be opened by a process, and *PIPE_BUF*. If any process writes to a FIFO and the data size is within *PIPE_BUF*, then the operation is guaranteed to be atomic. An atomic operation implies that whether the process writes complete data or not, the system doesn't intermix the data from the two processes, if the two processes want to write simultaneously. The non-blocking opening of a FIFO has no effect on the atomicity of *writes* to a pipe.

The maximum file descriptor value can be got through querying by calling the *sysconf* function or executing the *ulimit* command or *setrlimit* function in a C program. If you compile and execute the program, which is shown in Figure 6, it gives the value of *OPEN_MAX* and *PIPE_BUF* of your system. Constants beginning with *_SC_* are used as arguments to *sysconf*, and arguments beginning with *_PC_* are used as arguments to either *pathconf* or *fpathconf*.

*PC_*PIPE_BUF: maximum number of bytes that can be written atomically to a pipe.

_SC_OPEN_MAX: maximum number of open files per process.

Figure 7 is the snapshot of the querying *PIPE_BUF* value and modifying the value of *OPEN_MAX* from the shell, using *getconf* and *ulimit* commands.

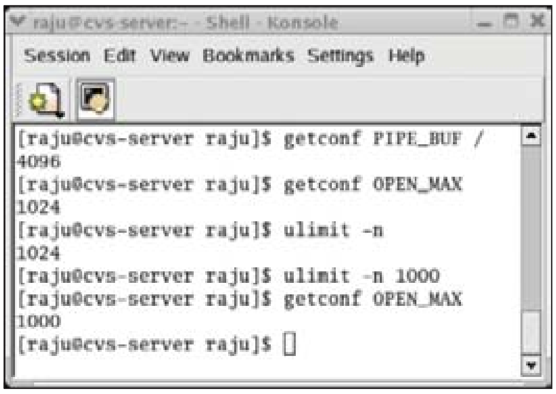

root@localhost:/home/raju/lfy/fifo
File Edit View Terminal Go Help

```
/* To find FIFO length and
 * Maximum number of files can be open within a process
 */
#include <unistd.h>

main ()
 {
   long PIPE_BUF, OPEN_MAX;

   PIPE_BUF = pathconf(".", _PC_PIPE_BUF);
   OPEN_MAX = sysconf(_SC_OPEN_MAX);
   printf ("PIPE_BUF = %ld\t OPEN_MAX = %ld\n",PIPE_BUF, OPEN_MAX );
 }
~
                                                9,0-1          All
```

Figure 6: Program to find system limitations


raju@cvs-server:~ - Shell - Konsole
Session Edit View Bookmarks Settings Help

```
[raju@cvs-server raju]$ getconf PIPE_BUF /
4096
[raju@cvs-server raju]$ getconf OPEN_MAX
1024
[raju@cvs-server raju]$ ulimit -n
1024
[raju@cvs-server raju]$ ulimit -n 1000
[raju@cvs-server raju]$ getconf OPEN_MAX
1000
[raju@cvs-server raju]$ 
```

Figure 7: Screen shot for the usage of getconf and ulimit commands

So far, we have seen the primitive interprocess communication of pipes and of FIFO. The main drawback of FIFO is its zero buffering capacity. We can't store any data inside the buffer—the moment we send the data, it will be transferred to the other end. In many situations, we need more flexible mechanisms to communicate between different processes. System V Inter Process Communications (Sys V IPC) deals with three advanced mechanisms for communication and synchronisation, namely, message queues, shared memory and semaphores. In the next article, we will see a big picture about Sys V IPC mechanisms. LFY

Acknowledgement: The author would like to thank T. Anandaraj and S. Jaishankar for their help during the preparation of this manuscript.

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