**Introduction**

The Linux IPC (Inter-process communication) facilities provide a method for multiple pro-

cesses to communicate with one another. There are several methods of IPC available to

Linux C programmers:

**Half-duplex UNIX Pipes**

**Basic Concepts**

Simply put, a pipe is a method of connecting the standard output of one process to the

standard input of another. Pipes are the eldest of the IPC tools, having been around since

the earliest incarnations of the UNIX operating system. They provide a method of one-way

communications (hence the term half-duplex) between processes.

This feature is widely used, even on the UNIX command line (in the shell).

**ls | sort | lp**

The above sets up a pipeline, taking the output of **ls** as the input of **sort**, and the output

of sort as the input of **lp**. The data is running through a half duplex pipe, traveling (visually)

left to right through the pipeline.

When a process creates a pipe, the kernel sets up **two file descriptors** for use by the

pipe. One descriptor is used to allow a path of input into the pipe (write), while the other

is used to obtain data from the pipe (read).

**Creating Pipes in C**

To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline.

SYSTEM CALL: pipe();

PROTOTYPE: int pipe( int fd[2] );

HEADER : #include <unistd.h>

RETURNS: 0 on success

-1 on error: errno = EMFILE (no free descriptors)

EMFILE (system file table is full)

EFAULT (fd array is not valid)

NOTES: fd[0] is set up for reading, fd[1] is set up for writing

The ***first integer in the array (element 0)*** ***is set up and opened for reading***, while the

***second integer (element 1) is set up and opened for writing***. Visually speaking, the output

of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves

through the kernel.

**See pipe0.c**

main()

{

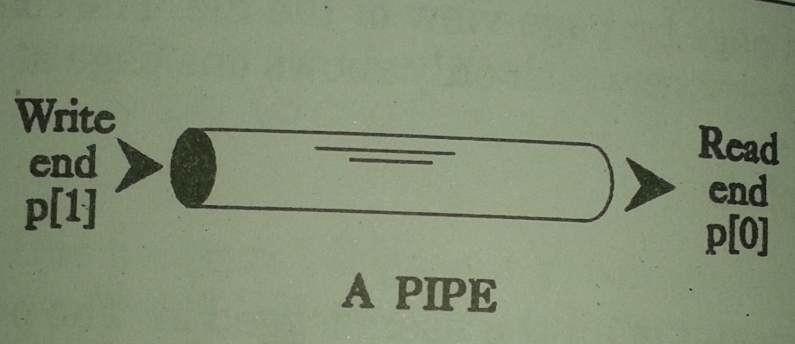
int p[2];

retval=pipe(p);

printf(“p[0] is %d and p[1] is %d\n “, p[0],p[1]);

}

**Figure 1**



We pass the starting address of array to pipe and values of two descriptors are returned in the array and those vales are printed. p[0] is read end and p[1] is write end.

Following program tells you how many pipes can be created at one time.

/\* How many pipes can be created? \*/ **See pipe3.c**

#include<stdio.h>

main()

{

int p[2],retval,i=0;

while(1)

{

retval=pipe(p);

if(retval==-1)

{

printf("pipe not created\n");

break;

}

i++;

printf("p[0] is %d and p[1] is %d i=%d\n ", p[0],p[1],i);

}

}

**Accessing Pipe**

To access a pipe directly, the same system calls that are used for low-level file I/O

can be used .

To send data to the pipe, we use the write() system call, and to retrieve data from the

pipe, we use the read() system call.

**Reading Pipe**

Reading pipe end can be done using read command which we have used already in files.

**Syntax : size\_t read(int fd, char \*buffer , size\_t bytes);**

**Example: read(fd[0], buf, 100);**

**include file: #include <sys/types.h>**

**Writing Pipe**

Writing pipe end can be done using write command which we have used already in files.

**size\_t write(int fd, char \*buffer, size\_t bytes);**

**Example: write(fd[1],buf, 100);**

**include file: #include <sys/types.h>**

**see pipe1.c**

**/\* usage of pipe read, write with in same process \*/**

#include<stdio.h>

#include<sys/types.h>

#define MSGSIZE 16

main()

{

char\* msg1="Hello World 1";

char\* msg2="Hello World 2";

char\* msg3="Hello World 3";

char inbuf[MSGSIZE];

int p[2],i;

pipe(p);

write(p[1] ,msg1 ,MSGSIZE);

write(p[1] ,msg2 ,MSGSIZE);

write(p[1] ,msg3 ,MSGSIZE);

for (i=0;i<3;i++)

{

read(p[0],inbuf,MSGSIZE);

printf("%s\n",inbuf);

}

exit(0);

}

Example:

**Fork and Pipe**

/\* usage of close(),read() and write() , fork() in pipe()\*/

#include<stdlib.h>

#include<stdio.h>

#define MSGSIZE 16

main()

{

char\* msg1="Hello World 1";

char inbuf[MSGSIZE];

int p[2],pid;

pipe(p);

pid=fork();

if (pid>0)

{

printf("Parent : p[0]=%d and p[1]=%d\n", p[0],p[1]);

}

if (pid==0)

{

printf("Child :p[0]=%d and p[1]=%d\n", p[0],p[1]);

}

exit(0);

}

Example: See pipe???

/\* parent process write data to pie and child reads from pipe, fork() used to create parent and child\*/

#include<stdio.h>

#define MSGSIZE 16

main()

{

int p[2],pid;

char\* msg1="Hello World 1";

char inbuf[MSGSIZE];

pipe(p);

pid=fork();

if(pid>0)

{

printf("In Praent \n");

write(p[1] ,msg1 ,MSGSIZE);

}

else

{

printf("\nIn child \n");

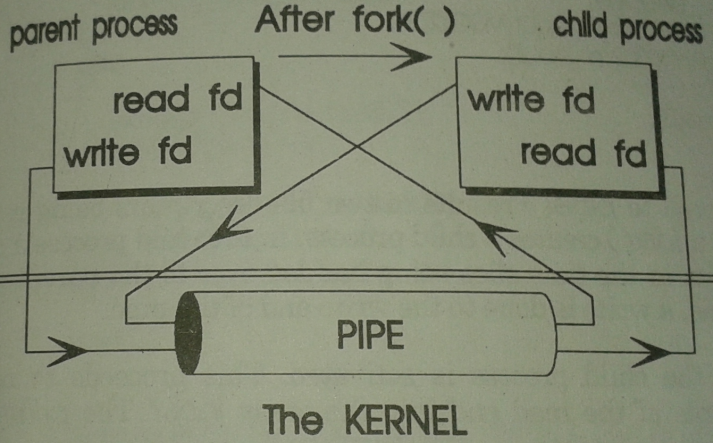
read(p[0],inbuf,MSGSIZE);

printf("\n%s\n",inbuf);

}

}

**figure 2 and fig 3**



By doing as above in the program, it is possible for the both parent and child to write to the write end of pipe (see fig 2). This will not be a communication, because we considered it as half duplex communication, i.e. parent writes first to write end of pipe and next child reads from read end of pipe or child writes and then parent reads. To ensure this we have take some precautions of closing and opening the write end , read end of pipe.

**Parent has to close read end before writing to write end, and child has to close write end before reading from read end (see Figure 3).**

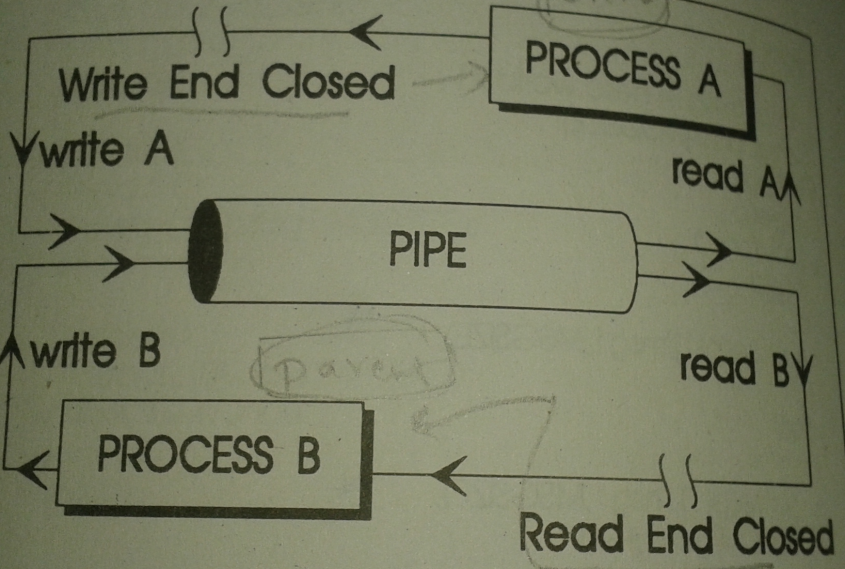


Figure 3

We need to use close()

(If the parent wants to receive data from the child, it should close fd1, and the child should close fd0.

If the parent wants to send data to the child, it should close fd0, and the child should close fd1.

Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren't concerned with.)

int p[2];

pipe(p);

we can use close(p[0]) to close read end

close(p[1]); to close write end.

**See pipe2.c**

/\* usage of close (), read (), fork () and write () in pipe ()\*/

#include<stdio.h>

#define MSGSIZE 16

main()

{

char\* msg1="Hello World 1";

char inbuf[MSGSIZE];

int p[2],pid;

pipe(p);

pid=fork();

if (pid>0)

{

close(p[0]);

write(p[1] ,msg1 ,MSGSIZE);

}

if (pid==0)

{

close(p[1]);

read(p[0] ,inbuf ,MSGSIZE);

printf(" In child %s",inbuf);

}

exit(0);

}

Sort numbers in a array using parent child process created using fork() and use pipe() for communication between parent and child process.

/\* Sorting array of numbers using pipe \*/

#include <stdio.h>

#define MAXSIZE 10

void main()

{

int pid,array[MAXSIZE],a[10];

int i, j, num, temp;

int p[2],o;

pipe(p);

printf("Enter the value of num \n");

scanf("%d", &num);

printf("Enter the elements one by one \n");

for (i = 0; i < num; i++)

{

scanf("%d", &array[i]);

}

printf("Input array is \n");

for (i = 0; i < num; i++)

{

printf("%d\n", array[i]);

}

pid=fork();

/\* Bubble sorting begins \*/

if (pid==0)

{ printf("Child -Sorting \n");

for (i = 0; i < num; i++)

{

for (j = 0; j < (num - i - 1); j++)

{

if (array[j] > array[j + 1])

{

temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}

}

}

for (i = 0; i < num; i++)

{

printf("Child %d\n", array[i]);

}

close(p[0]);

write(p[1],array,sizeof(int)\*num);

printf(" child ends \n");

}

else

{

close(p[1]);

read(p[0],a,sizeof(int)\*num);

printf("Sorted array is...\n");

for (i = 0; i < num; i++)

{

read(p[0],a[i],sizeof(int));

printf("Parent %d\n", a[i]);

}

}

}

|  |  |
| --- | --- |
| Network Programming | Unix Pipes |

A pipe is used for one-way communication of a stream of bytes. The command to create a pipe is ***pipe()***, which takes an array of two integers. It fills in the array with two file descriptors that can be used for low-level I/O.

We can **use two pipes to make Duplex communication**.

**Figure 4**

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

main()

{

int pp[2],pc[2], pid;

char msg1[20];

char msg2[20];

char msg3[20];

pipe(pp);

pipe(pc);

pid=fork();

if(pid==0)

{

close(pc[0]);

write(pc[1],"Hi Dad",6);

msg2[6]='\0';

close(pp[1]);

read(pp[0],msg2,11);

msg2[11]='\0';

printf("chiild reading-%s\n",msg2);

close(pc[0]);

write(pc[1],"Thank you Dad",13);

}

else

{

close(pc[1]);

read(pc[0],msg1,6);

msg1[6]='\0';

printf("Parent Reading-%s\n",msg1);

### close(pp[0]);

### write(pp[1],"Hi My Child",11);

### close(pc[1]);

### read(pc[0],msg3,13);

### msg3[13]='\0';

### printf("Parent reading-%s\n",msg3);

### }

### }

### EXAMPLE 2:

### #include<stdio.h>

### #include<stdlib.h>

### #include<sys/types.h>

### #include<unistd.h>

### main(){

### int pp[2],pc[2], pid;

### int a[10],b[10],c[10],d[10];

### pipe(pp);

### pipe(pc);

### pid=fork();

### if(pid==0) {

### sleep(5);

### close(pc[1]); //pc - write close

### read(pc[0],b,5\*sizeof(int));

### printf(" CHILD - received array b \n");

### for(int j=0;j<5;j++){

### printf(" %d \t",b[j]);

### c[j]=b[j]\*b[j];

### }

### close(pp[0]);

### write(pp[1],c,5\*sizeof(int));

### }

### else {

### printf(" Enter elements to array \n");

### for(int i=0;i<5;i++){

### scanf("%d",&a[i]);

### }

### close(pc[0]); // pc -read close

### write(pc[1],a,5\*sizeof(int));

### close(pp[1]);

### read(pp[0],d,5\*sizeof(int));

### printf("\n PARENT - Received squared arrayd \n");

### for(int k=0;k<5;k++){

### printf(" %d \t",d[k]);

### }

### printf("\n");

### wait(0);

### }

### }

### 

### 

**NAMED PIPES (FIFOs)**

In unnamed pipes (**half duplex** discussed previously) has **some drawbacks**, such as-

1. These pipes (half duplex) can be used only in process which have **common ancestry like parent and child processes.**

2. These **pipes are not permanent**, if a Process which creates them terminated then pipes also get terminated.

To overcome these Named Pipes are used.

These Named pipes are Permanent fixtures in Unix. Unix gives them size, owner, access permissions like files. It cane be opened ,closed or deleted.

**Experiment with Named Pipe**

We can create named pipe using *mknod() or mkfifo()* sytem call from command prompt.

We use the the command **ls -l > cat > a1.txt**  to display list of files/folders details and it pipelined to cat which in turn store in a1.txt. Similar kind of activity we can do using pipes and displaying ls-l output to the screen instead of text file.

Type the following series of commands in terminal(first)-

**$ mknod testpipe p**

mknod creates pipe named testpipe

You can check it

**$ ls -l testpipe\***

Make the cat command which takes input from pipe as background process as below-

Open two terminals.

In first terminal type as below-

**$ cat < testpipe &**

By **making it background process** , cat process will not hang if there is no input.

Now in the second terminal execute following command-

**$ ls -l >testpipe**

This will send the display of files/folder listing to pipe and in-turn in second terminal(receives output) which was running **cat** in background receives output of **ls -l** through **testpipe** and displays output on screen.

**Example: p217.c and p218.c**

Initially create pipe testpipe.

**$ mknod testpipe p**

**Terminal 1**

$ cc p217.c -o p217

$ cc p218.c -o p218

Terminal 1(client)

$ ./p218 hai hello

$

**Terminal 2(Server)**

$ ./p217

Message Received hai

Message Received hello

-------------p217.c--------------------

#include<fcntl.h>

#include<stdio.h>

#include <strings.h>

#include<errno.h>

#define MSGSIZ 63

main()

{

int fd;

char msgbuf[MSGSIZ+1];

int n;

bzero(msgbuf, MSGSIZ);

if(( fd=open("testpipe",O\_RDWR))<0)

perror("pipe open failed in p217\n");

for(;;)

{

if((n=read(fd,msgbuf,MSGSIZ+1))>0)

printf(" Message Received %s \n",msgbuf);

}

}

------------------p218.c---------------------------

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

#include<errno.h>

#include <strings.h>

#include<string.h>

#define MSGSIZ 63

main(argc,argv)

int argc;

char\* argv[];

{

int fd,j,nwrite;

char msgbuf[MSGSIZ];

bzero(msgbuf, MSGSIZ);

if(argc<2)

{

printf(" Type as - Filename msg1 msg2 ..\n");

exit(1);

}

if((fd=open("testpipe",O\_WRONLY))<0)

{

printf("pipe open failed in p218\n");

exit(1);

}

printf("FD:%d",fd);

for(j=1;j<argc;j++)

{

strcpy(msgbuf,argv[j]);

if((nwrite=write(fd,msgbuf,MSGSIZ+1))<0)

printf("pipe wirte failed in p218 \n");

}

exit(0);

}

Linux Programmer's Manual MKFIFO(3)

## NAME        MKFIFO() [top](http://man7.org/linux/man-pages/man3/mkfifo.3.html#section_dir)

mkfifo - make a FIFO special file (a named pipe)

Creates Named pipe inside c program.

## SYNOPSIS         [top](http://man7.org/linux/man-pages/man3/mkfifo.3.html#section_dir)

**#include <sys/types.h>**

**#include <sys/stat.h>**

**int mkfifo(const char \****pathname***, mode\_t** *mode***);**

## DESCRIPTION         [top](http://man7.org/linux/man-pages/man3/mkfifo.3.html#section_dir)

**mkfifo**() makes a FIFO special file with name *pathname*. *mode*

specifies the FIFO's permissions. It is modified by the process's

**umask** in the usual way: the permissions of the created file are **(***mode*

**& ~umask)**.

A FIFO special file is similar to a pipe, except that it is created

in a different way. Instead of being an anonymous communications

channel, a FIFO special file is entered into the file system by

calling **mkfifo**().

Once you have created a FIFO special file in this way, any process

can open it for reading or writing, in the same way as an ordinary

file. However, it has to be open at both ends simultaneously before

you can proceed to do any input or output operations on it. Opening

a FIFO for reading normally blocks until some other process opens the

same FIFO for writing, and vice versa.

RETURN VALUE         [top](http://man7.org/linux/man-pages/man3/mkfifo.3.html#section_dir)

On success **mkfifo**() returns 0. In the case of an error, -1 is

returned (in which case, [*errno*](http://man7.org/linux/man-pages/man3/errno.3.html) is set appropriately).

The mode argument is used to set the file's permissions.

**Example**: octal values such as 0666, 0777 etc. tells about Read, Write, Execute permissions to Owner, Group, Others. Also we can use as below using constants defined - S\_IRUSR (owner Read), S\_IWUSR(Owner Write),S\_IRGRP(Read permission for Group)S\_IROTH (Read for Others) S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH

The file mode, stored in the st\_mode field of the file attributes, contains two kinds of information: the **file type** code, and the **access permission** bits. This section discusses only the access permission bits, which control who can read or write the file.

The file type must be one of **S\_IFREG**, **S\_IFCHR**, **S\_IFBLK**, **S\_IFIFO**, or **S\_IFSOCK** to specify a regular file (which will be created empty),character special file, block special file, FIFO (named pipe), or UNIX domain socket, respectively. (Zero file type is equivalent to

type **S\_IFREG**.)

Access permissions are the permissions bits like -0664 which indicates rwx to owner ;rwx to group and rw- to others. (See file attributes document)

RETURN VALUE         top

**// Creating pipe using mkfifo() and opening pipe for writing and writing some messgae , in another program n\_pipe2.c we open pipe for reading and read the message and dipslay.**

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

int main()

{

int fd,fi,nwrite;

char msgbuf[64]="Hello";

fi=mkfifo("vmfifo",S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IROTH);

if(fi<0)

{

perror("vmfifo creation failed");

exit(1);

}

fd=open("vmfifo",O\_WRONLY);

if (fd<0)

{

perror("vmfifo open failed");

exit(1);

}

nwrite=write(fd,msgbuf,sizeof(msgbuf));

if(nwrite<=0)

perror("message write failed \n");

close(fd);

return(0);

}

**// In n\_pipe1.c named pipe is created using mkfifo() and message is written into pipe , in this program n\_pipe2.c we open pipe for reading and read the message and dipslay.**

#include<fcntl.h>

#include<stdio.h>

#include<stdlib.h>

main()

{

int fd,fi,nread;

char msgbuf[64];

//open named pipe created in n\_pipe1.c

fd=open("vmfifo",O\_RDWR);

if (fd<0)

{

perror("vmfifo open failed");

exit(1);

}

fd=open("vmfifo",O\_RDWR);

nread=read(fd,msgbuf,sizeof(msgbuf));

if(nread>0)

printf(" message received : %s\n",msgbuf);

else

printf(" Not reading from pipe\n");

close(fd);

return(0);

}

**unlink()**

unlink() **deletes a name from the filesystem**. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

Ex: unlink(“nameof\_FIFO”);

# **access()**

In Linux, *access* command is used to check whether the calling program has access to a specified file. It can be used to check whether a file exists or not. The check is done using the calling process’s real UID and GID.

int access(const char \*pathname, int mode);

Here, the first argument takes the path to the *directory/file* and the second argument takes flags *R\_OK, W\_OK, X\_OK or F\_OK*.

* **F\_OK flag** : Used to check for existence of file.
* **R\_OK flag** : Used to check for read permission bit.
* **W\_OK flag** : Used to check for write permission bit.
* **X\_OK flag** : Used to check for execute permission bit.

*Note:* If access() cannot access the file, it will return -1 or else it will be 0

**Example**: int fd = access("sample.txt", F\_OK);

Since 2nd argument is F\_OK, access () command checks for existence of sample.txt file.

Similarly, we can use the command to check existence of Named Pipe(fifo) before creation.

**Example**:

#define FIFO\_NAME "MyFifo"

int res;

if (**access**(FIFO\_NAME, F\_OK) == -1) {

res = mkfifo(FIFO\_NAME, 0777);

if (res != 0) {

fprintf(stderr, “Could not create fifo %s\n”, FIFO\_NAME);

exit(EXIT\_FAILURE);

}

}

printf(“Process %d opening FIFO O\_WRONLY\n”, getpid());

int pipe\_fd = open(FIFO\_NAME, O\_WRONLY);