# Introduction

The Linux IPC (Inter-process communication) facilities provide a method for multiple processes to communicate with one another. There are several methods of IPC available to Linux C programmers:

* Half-duplex UNIX pipes
* FIFOs (named pipes)
* SYSV style message queues
* SYSV style semaphore sets
* SYSV style shared memory segments
* Networking sockets (Berkeley style) (not covered)
* Full-duplex pipes (STREAMS pipes) (not covered)

# Half-duplex UNIX Pipes

A pipe is a method of connecting the standard output of one process to the standard input of another. They provide a method of one-way communications between processes.

```
$ ls | sort | lp
```

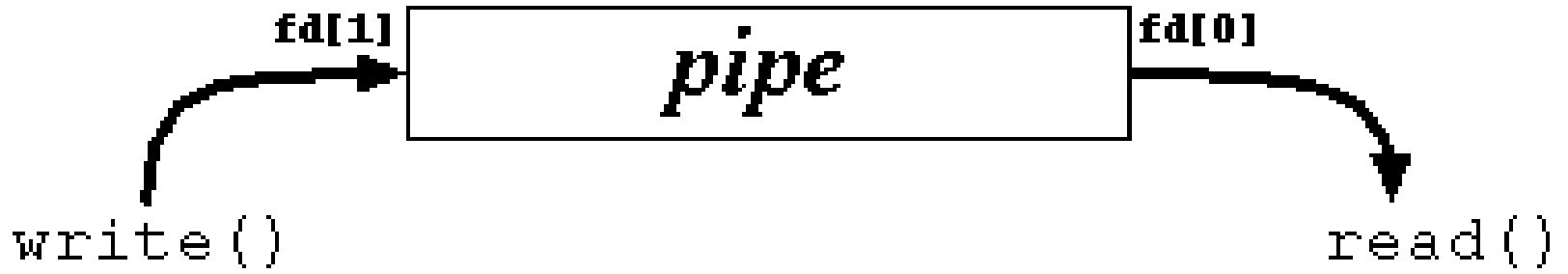
The above sets up a pipeline, taking the output of ls as the input of sort, and the output of sort as the input of lp.

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).

At this point, the pipe is of little practical use, as the creating process can only use the pipe to communicate with itself.



At this point, the pipe is fairly useless.

After all, why go to the trouble of creating a pipe if we are only going to talk to ourself?

At this point, the creating process typically forks a child process.

Since a child process will inherit any open file descriptors from the parent, we now have the basis for multiprocess communication (between parent and child).

Above, we see that both processes now have access to the file descriptors which constitute the pipeline.

It is at this stage, that a critical decision must be made. In which direction do we desire data to travel?

Does the child process send information to the parent, or vice-versa?

The two processes mutually agree on this issue, and proceed to ``close'' the end of the pipe that they are not concerned with.

For discussion purposes, let's say the child performs some processing, and sends information back through the pipe to the parent.

Construction of the pipeline is now complete!

The only thing left to do is make use of the pipe.

To access a pipe directly, the same system calls that are used for low-level file I/O can be used (recall that pipes are actually represented internally as a valid inode).

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.

Remember, low-level file I/O system calls work with file descriptors!

However, keep in mind that certain system calls, such as lseek (), do not work with descriptors to pipes.

## Creating Pipes in C

Creating ``pipelines'' with the C programming language can be a bit more involved than our simple shell example.

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.

After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).

SYSTEM CALL: pipe();

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

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.

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

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

Once we have established the pipeline, we then fork our new child process:

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

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.

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
        }
```

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
```

```
        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

## Pipes the Easy Way!

If all of the above ramblings seem like a very round-about way of creating and utilizing pipes, there is an alternative.

### LIBRARY FUNCTION: popen();

PROTOTYPE: FILE *popen ( char *command, char *type);

RETURNS: new file stream on success

NULL on unsuccessful fork() or pipe() call

NOTES: creates a pipe, and performs fork/exec operations using "command"

This standard library function creates a half-duplex pipeline by calling pipe() internally.

It then forks a child process, execs the Bourne shell, and executes the "command" argument within the shell.

Direction of data flow is determined by the second argument, "type". It can be "r" or "w", for "read" or "write". It cannot be both!

Under Linux, the pipe will be opened up in the mode specified by the first character of the "type" argument. So, if you try to pass "rw", it will only open it up in "read" mode.

Pipes which are created with popen() must be closed with pclose(). By now, you have probably realized that popen/pclose share a striking resemblance to the standard file stream I/O functions fopen() and fclose().

LIBRARY FUNCTION: pclose();

PROTOTYPE: int pclose( FILE *stream );

RETURNS: exit status of wait4() call

-1 if "stream" is not valid, or if wait4() fails

NOTES: waits on the pipe process to terminate, then closes the stream.

The pclose() function performs a wait4() on the process forked by popen(). When it returns, it destroys the pipe and the file stream. Once again, it is synonymous with the fclose() function for normal stream-based file I/O.

```
/*************************************************************
 MODULE: popen1.c
 *************************************************************/
#include <stdio.h>

#define MAXSTRS 5

int main(void)
{
	int  cntr;
	FILE *pipe_fp;
	char *strings[MAXSTRS] = { "echo", "bravo", "alpha",
				"charlie", "delta"};

	/* Create one way pipe line with call to popen() */
	if (( pipe_fp = popen("sort", "w")) == NULL)
	{
		perror("popen");
		exit(1);
	}

	/* Processing loop */
	for(cntr=0; cntr<MAXSTRS; cntr++) {
		fputs(strings[cntr], pipe_fp);
		fputc('\n', pipe_fp);
	}

	/* Close the pipe */
	pclose(pipe_fp);

	return(0);
```

Notes on half-duplex pipes:

* Two way pipes can be created by opening up two pipes, and properly reassigning the file descriptors in the child process.

* The pipe() call must be made BEFORE a call to fork(), or the descriptors will not be inherited by the child! (same for popen()).

* With half-duplex pipes, any connected processes must share a related ancestry. Since the pipe resides within the confines of the kernel, any process that is not in the ancestry for the creator of the pipe has no way of addressing it. This is not the case with named pipes (FIFOS).

# Named Pipes (FIFO) : Basic Concepts

A named pipe works much like a regular pipe, but does have some noticeable differences.

* Named pipes exist as a device special file in the file system.

* Processes of different ancestry can share data through a named pipe.

* When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

## Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

The above two commands perform identical operations, with one exception.

The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation.

With mknod, a quick call to the chmod command will be necessary.

FIFO files can be quickly identified in a physical file system by the ``p'' indicator seen here in a long directory listing:

To create a FIFO in C, we can make use of the mknod() system call:

LIBRARY FUNCTION: mknod();

PROTOTYPE: int mknod( char *pathname, mode_t mode, dev_t dev);

RETURNS: 0 on success,

-1 on error: errno = EFAULT (pathname invalid)

EACCES (permission denied)
ENAMETOOLONG (pathname too long)

ENOENT (invalid pathname)
ENOTDIR (invalid pathname)
(see man page for mknod for others)

NOTES: Creates a filesystem node (file, device file, or FIFO)

consider a simple example of FIFO creation from C:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In this case, the file ``/tmp/MYFIFO'' is created as a FIFO file. The requested permissions are ``0666'', although they are affected by the umask setting as follows:

```
final_umask = requested_permissions &
~original_umask
```

A common trick is to use the umask() system call to temporarily zap the umask value:

```
umask(0);
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

# FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception.

An ``open'' system call or library function should be used to physically open up a channel to the pipe.

With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem.

In our examples, we will treat the pipe as a stream, opening it up with fopen(), and closing it with fclose().

Consider a simple server process:

```
/*******
 MODULE: fifoserver.c
 ******/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"
```

```
int main(void)
{
	FILE *fp;
	char readbuf[80];

	/* Create the FIFO if it does not exist */
	umask(0);
	mknod(FIFO_FILE, S_IFIFO|0666, 0);

	while(1)
	{
		fp = fopen(FIFO_FILE, "r");
		fgets(readbuf, 80, fp);
		printf("Received string: %s\n", readbuf);
		fclose(fp);
	}

	return(0);
}
```

```
/****
 MODULE: fifoclient.c
 ****/

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

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

## Blocking Actions on a FIFO

Normally, blocking occurs on a FIFO.
In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing.
This action works vice-versa as well.
If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

In the case with our simple server, we just shoved it into the background, and let it do its blocking there.
The alternative would be to jump to another virtual console and run the client end, switching back and forth to see the resulting action.

## The Infamous SIGPIPE Signal

On a last note, pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.