Named Pipes (FIFOs - First In First Out)

6.3.1 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.

6.3.2 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:

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

Also notice the vertical bar (``pipe sign'') located directly after the file name. Another great reason to run Linux, eh?

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)
```

I will leave a more detailed discussion of mknod() to the man page, but let's 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.

6.3.3 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:

```
/*************************************************************************
*
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett

*************************************************************************
 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);
}
```

Since a FIFO blocks by default, run the server in the background after you compile it:

```
        $ fifoserver&
```

We will discuss a FIFO's blocking action in a moment. First, consider the following simple client frontend to our server:

```
/*****************************************************************************
*
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett

*****************************************************************************
 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);
}
```

6.3.4 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.

6.3.5 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.