FIFOs or Named Pipes

- Pipes are an elegant and powerful Interprocess communication mechanism. However, they have several serious drawbacks.
- Firstly, and most seriously, pipes can only be used to **connect processes** that share a **common ancestry**, such as a parent process and its child.
- This drawback becomes apparent when trying to develop a 'server' program that remains permanently in existence in order to provide a system-wide service.
- Ideally, client processes should be able to come into being, communicate with an unrelated server process via a pipe and then go away again. Unfortunately, this cannot be done using conventional pipes.
- Secondly, pipes cannot be permanent. They have to be created every time they are needed and then destroyed when the processes accessing them terminate.
- To address these deficiencies, a variant of the pipe was introduced. This IPC mechanism is called the **FIFO** or **named pipe**.
- As far as read and write are concerned, FIFOs are identical to pipes, acting as one-way, **First-In-First-Out** communications channels between processes.
- Unlike a pipe, however, a FIFO is a permanent fixture and is given a UNIX file name. A FIFO also has an owner, a size and associated access permissions. It can be opened, closed and deleted like any other UNIX file, but displays properties identical to pipes when read or written.
- First let us look at the use of FIFOs at command level before examining the programming interface. The command ***mknod*** is used to create a FIFO:

  $ /bin/mknod channel p

- Here **channel** is the name of the FIFO (it can be any valid UNIX pathname). The second argument **p** tells ***mknod*** to create a FIFO. It is needed since ***mknod*** is also used to create the special files that represent devices.
- The FIFO can be read and written using standard UNIX commands, for example:

  $ cat < channel

- If this command were executed just after channel is created it would 'hang'. This is because by default, a process opening a FIFO for reading will block until another process attempts to open the FIFO for writing.
- Similarly, a process attempting to open a FIFO for writing will block until a process attempts to open it for reading.
- This is entirely sensible, since it saves system resources and makes program coordination simpler.

- As a consequence, if we wanted to create a reader and writer for our last example we would have to execute one of the processes in the background, for example:

```
$ cat < channel&
14942
$ ls -l > channel; wait
-rw-r--r--    1  root root        15 Mon Jan 27 09:36 bar
prw-r--r--    1  root root         0 Mon Jan 27 09:37 channel
-rw-r--r--    1  root root        15 Mon Jan 27 09:35 foo
```

- The listing of the directory in initially produced by ***ls*** and then written down the FIFO. The waiting ***cat*** command then reads data from the FIFO and displays it onto the screen.

- The process running **cat** now exits. This because, when a FIFO is no longer open for writing, a read on it will return 0 just like a normal pipe, which cat takes to mean end of file.

- The **wait** command causes the shell to wait until cat exits before redisplaying the prompt.

Programming with FIFOs

- Instead of using pipe, a FIFO is instead created with ***mknod***. A value of octal 010000 must be added to the mode value to signify a FIFO. So for example, the code fragment

```
if (mknod ("fifo", 010600, 0) < 0)
        perror ("mknod (fifo) call");
```

  creates a FIFO called **fifo** which has permissions of 0600 and is therefore readable and writable by the FIFO owner.

- A more detailed discussion of ***mknod()*** can be found in the man pages. The following is another way of creating a FIFO:

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

- The requested permissions are "**0666**", although they are affected by the default ***umask*** setting as follows:

final_umask = requested_permissions & ~original_umask

- One way of getting around that is to use the ***umask()*** system call to temporarily disable 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.

- Once created, a FIFO must be opened using open. So, for example, the call

```
#include <fcntl.h>
.
.
fd = open ("fifo", O_WRONLY);
```

  opens a FIFO for writing. As we saw earlier, this call will block until another process open the FIFO for reading.

- **Non-blocking** open calls on a FIFO are possible. To achieve this the open call must be made with the **O_NDELAY** flag (defined in **fcntl.h**) bitwise OR'ed with one of **O_RDONLY**, **O_WRONLY** and **O_RDWR**. For example:

```
if ((fd = open ("fifo", O_RDONLY|O_NDELAY)) < 0)
        perror ("open on fifo");
```

- If no process has the FIFO open for writing, then this open will return -1 instead of blocking. If, on the other hand, the open was successful, future read calls on the FIFO will also be non-blocking.

- Time for examples. The first, sendmsg sends individual messages to a FIFO named fifo. It is called as follows:

```
$ sendmsg "message text 1" "message text 2"
```

- Messages are sent as 64-character chunks, via a non-blocking write call. The actual message text is restricted to 63 characters to allow for a trailing null character.

- Notice how a return value of zero from write, indicating a full pipe, is treated; **errno** is forced to take the value **EAGAIN** before fatal is called.

- The next program, ***rcvmsg***, receives the messages by reading the FIFO is called ***rcvmsg***. It does nothing useful, and is intended just to serve as a basic framework.

- The following dialogue shows how these programs can be used. Since neither program creates the FIFO, the first step in the dialogue is to do just this using ***mknod***. ***rcvmsg*** is then placed into the background to receive messages from different invocations of ***sendmsg***.

```
$ mknod fifo p
$ rcvmsg&
14988
$ sendmsg "message 1" "message 2"
message received:message 1
message received:message 2
$ sendmsg "message number 3"
message received:message number 3
```