Named Pipes

- Unnamed pipes have several drawbacks
  - Shared by processes with a command ancestor
  - Cease to exist as soon as the processes that using them terminate
- Named pipes make up for these shortcomings
- A named pipe, or FIFO, is very much like an unnamed pipe in how you use it.
  - Read from it and write to it in the same way
  - It behaves the same way with respect to the consequences of opeing and closing it when various processes are either reading or writing or doing neither
- Differences:
  - They exist as directory entries in the file system and therefore have associated permissions and ownership
  - They can be use by processes that are not related to each other
  - They can be created and deleted at the shell level or at the programming level

Named Pipes at the Command Level

- There are two commands to create a FIFO: the older one is mknod, the new one is mkfifo.

$mknod PIPE p

Creates a FIFO named "PIPE", where 'p' indicates to mknod that PIPE should be a FIFO

$ls –l PIPE

prw-r--r-- 1 wanghd wanghd 0 2014-11-03 22:25 PIPE|

Try the following command sequence:

$ cat < PIPE &

$ ls –l > PIPE; wait

The cat command is trying to read from PIPE and so it will not return and we will not get the shell prompt back without backgrounding it.

The cat command will terminate as soon as it receives the return value 0 from its read() call

In this example, the writer is the process that executes "ls –l"

The wait command's only purpose is to delay the shell's prompt until after cat exits.

Programming with Named Pipes

- A named pipe can be created either by using the mknod() system call, or the mkfifo() library function.
  - Recommended to use mkfifo()
  - Easier to use and does not require super user privileges

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

  - The call mkfifo("MY_PIPE", 0666) creates a FIFO named MY_PIPE with permission 0666 & ~umask.
  - The convention is to use UPPERCASE letters for the names of FIFOs.
  - ***public*** and ***private*** FIFOs:
    - A ***public FIFO*** is one that is known to all clients. It is not that there is a specific function that makes a FIFO public; it is just that it is given a name that is easy to remember and that its location is advertised so that client programs know where to find it.
    - A ***private FIFO***, in contrast, is given a name that is not known to anyone except the process that creates it and the processes to which it chooses to divulge it. In our first example, we will use only a single public FIFO.

A Server-Client Example

```
#include "fifo1.h"

int main( int argc, char *argv[])
{
    int         nbytes;         // number of bytes read from popen()
    int         n = 0;
    int         dummyfifo;      // file descriptor to write-end of PUBLIC
    int         publicfifo;     // file descriptor to read-end of PUBLIC
    static char buffer[PIPE_BUF];// buffer to store output of command

    // Create public FIFO
    if ( mkfifo(PUBLIC, 0666) < 0 )
        if (errno != EEXIST ) {
            perror(PUBLIC);
            exit(1);
        }
    if ( (publicfifo = open(PUBLIC, O_RDONLY) ) == -1 ||
         ( dummyfifo = open(PUBLIC, O_WRONLY | O_NDELAY )) == -1  ) {
        perror(PUBLIC);
        exit(1);
    }
    while ( 1 ) {
        memset(buffer, 0, PIPE_BUF);
        if ( ( nbytes = read( publicfifo, buffer, PIPE_BUF)) > 0 ) {
            buffer[nbytes] = '\0';
            printf("Message %d received by server: %s", ++n, buffer);
            fflush(stdout);
        }
        else
            break;
    }
    return 0;
}
```

A Server-Client Example

```
// sendfifo1.c
#include "fifo1.h"
#define  QUIT     "quit"

int main( int argc, char *argv[])
{
    int    nbytes;         // num bytes read
    int    publicfifo;     // file descriptor to write-end of PUBLIC
    char   text[PIPE_BUF];

    // Open the public FIFO for writing
    if ( (publicfifo = open(PUBLIC, O_WRONLY) ) == -1) {
        perror(PUBLIC);
        exit(1);
    }
    // Repeatedly prompt user for command, read it, and send to server
    while (1) {
        memset(text, 0x0, PIPE_BUF);    // zero string
        nbytes = read(fileno(stdin), text, PIPE_BUF);
        if ( !strncmp(QUIT, text, nbytes-1))   // is it quit?
            break;
        write(publicfifo, text, nbytes);
    }
    // User quit; close write-end of public FIFO
    close(publicfifo);
}
```


A Server-Client Example

- The server creates a public FIFO.
- The server and client know the FIFO name because they share a header file.
- The server creates the FIFO and opens it for both reading and writing even if it only needs to read incoming messages on the pipe.
  - because the FIFO needs to have at least one process that has it open for writing, otherwise the server will immediately receive an end-of-file on the FIFO and close its reading loop.
- the server will simply block on the read() to it, waiting for a client to send it data.
- The client opens the public FIFO for writing and then enters a loop where it repeatedly reads from standard input and writes into the write-end of the public FIFO.
- The server reads from the public FIFO and displays the message it receives on its standard output, even though it may be put in the background; it is not detached from the terminal. The best way to run it is to leave it in the foreground and open a few clients in other terminal windows.
- The server increments a counter and displays each received message with the value of the counter, so that you can see the order in which the messages were received. It flushes standard output just in case there is no newline in the message.