**FIFO**

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

**Syntax**

#include <sys/types.h>

#include <sys/stat.h>

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

#include <fcntl.h> /\* Definition of AT\_\* constants \*/

#include <sys/stat.h>

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

**Return**

On success 0

On failure -1, errno is set appropriately

mkfifo() makes a FIFO special file with name pathname. mode specifies the FIFO's permissions.

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 filesystem 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.

A FIFO must be opened either read-only or write-only. It must not be opened for read-write, because a FIFO is half-duplex.

Additional to **mkfifoat**()

1. If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by mkfifo() for a relative pathname).
2. If pathname is relative and dirfd is the special value AT\_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like mkfifo()).
3. If pathname is absolute, then dirfd is ignored.

# Broken Pipe

“Broken Pipe”! What does this mean?

Well, what has happened is that when all readers for a FIFO close and the writer is still open, the writer will receiver the signal SIGPIPE the next time it tries to write(). The default signal handler for this signal prints “Broken Pipe” and exits. Of course, you can handle this more gracefully by catching SIGPIPE through the signal() call.

# Example

// writer.c

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#define FIFO\_NAME "american\_maid"

int main(void) {

char s[300];

int num, fd;

mkfifo(FIFO\_NAME, 0644);

printf("waiting for readers...\n");

fd = open(FIFO\_NAME, O\_WRONLY);

printf("got a reader--type some stuff\n");

while (gets(s), !feof(stdin)) {

if ((num = write(fd, s, strlen(s))) == -1)

perror("write");

else

printf("writer: wrote %d bytes\n", num);

}

return 0;

}

// reader.c

#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

#define FIFO\_NAME "american\_maid"

int main(void) {

char s[300];

int num, fd;

mkfifo(FIFO\_NAME, 0644);

printf("waiting for writers...\n");

fd = open(FIFO\_NAME, O\_RDONLY);

printf("got a writer\n");

do {

if ((num = read(fd, s, 300)) == -1)

perror("read");

else {

s[num] = '\0';

printf("tick: read %d bytes: \"%s\"\n", num, s);

}

} while (num > 0);

return 0;

}

# Additional Properties of Pipes and FlFOs

## Set non-block

If a descriptor is already open, fcntl() can be called to enable the O\_NONBLOCK flag. This technique must be used with a pipe, since open is not called for a pipe, and no way exists to specify the 0-NONBLOCK flag in the call to pipe.

**How to set non-block ?**

**Correct way**

first fetch the current file status flags with the F-GETFL command, bitwise-OR the O\_NONBLOCK flag, and then store the file status flags with the F-SETFL command:

int flags;

if ( (flags = fcntl(fd, F\_GETFL, 0)) < 0)

err-sys ( "F\_GETFL error" ) ;

flags |= O\_NONBLOCK;

if (fcntl(fd, F\_SETFL, flags) < 0)

err-sys("F\_SETFL error");

**Incorrect way**

if ( fcntl(fd, F\_SETFL, O\_NONBLOCK) < 0 )

err-sys("F-SETFL error");

**Problem:** This also clears all the other possible file status flags.

## Atomic write operation

If the number of bytes to write <= PIPE-BUF the write is guaranteed to be atomic. The system does not intermix the data from the two processes.

If the number of bytes to write > PIPE-BUF, there is no guarantee that the write operation is atomic.

## Write in case of O\_NONBLOCK

When a pipe or FIFO is set nonblocking, the return value from write depends on the number of bytes to write and the amount of space currently available in the pipe or FIFO.

1. If the **number of bytes to write <= PIPE-BUF:**

* If there is room in the pipe or FIFO for the requested number of bytes, all the bytes are transferred.
* If there is not enough room in the pipe or FIFO for the requested number of bytes, return is made immediately with an error of EAGAIN. Since the O\_NONBLOCK flag is set, the process does not want to be put to sleep. But the kernel cannot accept part of the data and still guarantee an atomic write, so the kernel must return an error and tell the process to try again later.

1. If the **number of bytes to write > PIPE-BUF:**

* If there is room for at least 1 byte in the pipe or FIFO, the kernel transfers whatever the pipe or FIFO can hold, and that is the return value from write.
* If the pipe or FIFO is full, return is made immediately with an error of EAGAIN.

## Broken Pipe

If we write to a pipe or FIFO that is not open for reading, the SIGPIPE signal is generated:

* If the process does not catch or ignore SIGPIPE, the default action of terminating the process is taken.
* If the process ignores the SIGPIPE signal, or if it catches the signal and returns from its signal handler, then write returns an error of EPIPE.

# Pipe and FIFO Limits

The only system-imposed limits on pipes and FIFOs are

**OPEN\_MAX** the maximum number of descriptors open at any time by a process (Posix requires that this be at least 16)

**PIPE\_BUF** the maximum amount of data that can be written to a pipe or FIFO atomically

One characteristic of pipes and FIFOs is that their data is a byte stream, similar to a TCP connection. Any delineation of this byte stream into records is left to the application.

Message queues provide record boundaries, similar to UDP datagrams.

# END