NAME

pipe − overview of pipes and FIFOs

DESCRIPTION

Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel. A pipe has a *read end* and a *write end*. Data written to the write end of a pipe can be read from the read end of the pipe.

A pipe is created using **pipe**(2), which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end. Pipes can be used to create a communication channel between related processes; see **pipe**(2) for an example.

A FIFO (short for First In First Out) has a name within the file system (created using **mkfifo**(3)), and is opened using **open**(2). Any process may open a FIFO, assuming the file permissions allow it. The read end is opened using the **O_RDONLY** flag; the write end is opened using the **O_WRONLY** flag. See **fifo**(7) for further details. *Note*: although FIFOs have a pathname in the file system, I/O on FIFOs does not involve operations on the underlying device (if there is one).

I/O on Pipes and FIFOs

The only difference between pipes and FIFOs is the manner in which they are created and opened. Once these tasks have been accomplished, I/O on pipes and FIFOs has exactly the same semantics.

If a process attempts to read from an empty pipe, then **read**(2) will block until data is available. If a process attempts to write to a full pipe (see below), then **write**(2) blocks until sufficient data has been read from the pipe to allow the write to complete. Non-blocking I/O is possible by using the **fcntl**(2) **F_SETFL** operation to enable the **O_NONBLOCK** open file status flag.

The communication channel provided by a pipe is a *byte stream*: there is no concept of message boundaries.

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to **read**(2) from the pipe will see end-of-file (**read**(2) will return 0). If all file descriptors referring to the read end of a pipe have been closed, then a **write**(2) will cause a **SIGPIPE** signal to be generated for the calling process. If the calling process is ignoring this signal, then **write**(2) fails with the error **EPIPE**. An application that uses **pipe**(2) and **fork**(2) should use suitable **close**(2) calls to close unnecessary duplicate file descriptors; this ensures that end-of-file and **SIGPIPE**/**EPIPE** are delivered when appropriate.

It is not possible to apply **lseek**(2) to a pipe.

Pipe Capacity

A pipe has a limited capacity. If the pipe is full, then a **write**(2) will block or fail, depending on whether the **O_NONBLOCK** flag is set (see below). Different implementations have different limits for the pipe capacity. Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available, so that a writing process does not remain blocked.

In Linux versions before 2.6.11, the capacity of a pipe was the same as the system page size (e.g., 4096 bytes on i386). Since Linux 2.6.11, the pipe capacity is 65536 bytes.

PIPE_BUF

POSIX.1-2001 says that **write**(2)s of less than **PIPE_BUF** bytes must be atomic: the output data is written to the pipe as a contiguous sequence. Writes of more than **PIPE_BUF** bytes may be non-atomic: the kernel may interleave the data with data written by other processes. POSIX.1-2001 requires **PIPE_BUF** to be at least 512 bytes. (On Linux, **PIPE_BUF** is 4096 bytes.) The precise semantics depend on whether the file descriptor is non-blocking (**O_NONBLOCK**), whether there are multiple writers to the pipe, and on *n*, the number of bytes to be written:

O_NONBLOCK disabled, *n* <= **PIPE_BUF**
All *n* bytes are written atomically; **write**(2) may block if there is not room for *n* bytes to be written immediately

O_NONBLOCK enabled, *n* <= **PIPE_BUF**
If there is room to write *n* bytes to the pipe, then **write**(2) succeeds immediately, writing all *n* bytes; otherwise **write**(2) fails, with *errno* set to **EAGAIN**.

O_NONBLOCK disabled, *n* > **PIPE_BUF**
The write is non-atomic: the data given to **write**(2) may be interleaved with **write**(2)s by other process; the **write**(2) blocks until *n* bytes have been written.

O_NONBLOCK enabled, *n* > **PIPE_BUF**
If the pipe is full, then **write**(2) fails, with *errno* set to **EAGAIN**. Otherwise, from 1 to *n* bytes may be written (i.e., a "partial write" may occur; the caller should check the return value from **write**(2) to see how many bytes were actually written), and these bytes may be interleaved with writes by other processes.

Open File Status Flags

The only open file status flags that can be meaningfully applied to a pipe or FIFO are **O_NONBLOCK** and **O_ASYNC**.

Setting the **O_ASYNC** flag for the read end of a pipe causes a signal (**SIGIO** by default) to be generated when new input becomes available on the pipe (see **fcntl**(2) for details). On Linux, **O_ASYNC** is supported for pipes and FIFOs only since kernel 2.6.

Portability notes

On some systems (but not Linux), pipes are bidirectional: data can be transmitted in both directions between the pipe ends. According to POSIX.1-2001, pipes only need to be unidirectional. Portable applications should avoid reliance on bidirectional pipe semantics.

SEE ALSO

dup(2), **fcntl**(2), **open**(2), **pipe**(2), **poll**(2), **select**(2), **socketpair**(2), **stat**(2), **mkfifo**(3), **epoll**(7), **fifo**(7)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.