Rocky Enterprise Linux 9.2 Manual Pages on command 'pipe.7'

$ man pipe.7

PIPE(7) Linux Programmer's Manual PIPE(7)

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 commu? nication channel between related processes; see pipe(2) for an example.

A FIFO (short for First In First Out) has a name within the filesystem (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 filesystem, I/O on FIFOs does not involve opera?

tions 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. Nonblocking 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 re? main 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 16 pages (i.e., 65,536 bytes in a system with a page size of 4096 bytes). Since Linux 2.6.35, the default pipe capac? ity is 16 pages, but the capacity can be queried and set using the fc? ntl(2) F_GETPIPE_SZ and F_SETPIPE_SZ operations. See fcntl(2) for more information.

The following ioctl(2) operation, which can be applied to a file de? scriptor that refers to either end of a pipe, places a count of the number of unread bytes in the pipe in the int buffer pointed to by the final argument of the call:

```
ioctl(fd, FIONREAD, &nbytes);
```

The FIONREAD operation is not specified in any standard, but is pro? vided on many implementations.

/proc files

On Linux, the following files control how much memory can be used for pipes:

/proc/sys/fs/pipe-max-pages (only in Linux 2.6.34)

An upper limit, in pages, on the capacity that an unprivileged user (one without the CAP_SYS_RESOURCE capability) can set for a pipe.

The default value for this limit is 16 times the default pipe capacity (see above); the lower limit is two pages.

This interface was removed in Linux 2.6.35, in favor of /proc/sys/fs/pipe-max-size.

/proc/sys/fs/pipe-max-size (since Linux 2.6.35)

The maximum size (in bytes) of individual pipes that can be set by users without the CAP_SYS_RESOURCE capability. The value as? signed to this file may be rounded upward, to reflect the value actually employed for a convenient implementation. To determine the rounded-up value, display the contents of this file after assigning a value to it.

The default value for this file is 1048576 (1 MiB). The minimum value that can be assigned to this file is the system page size. Attempts to set a limit less than the page size cause write(2) to fail with the error EINVAL.

Since Linux 4.9, the value on this file also acts as a ceiling on the default capacity of a new pipe or newly opened FIFO.

/proc/sys/fs/pipe-user-pages-hard (since Linux 4.5)

The hard limit on the total size (in pages) of all pipes created or set by a single unprivileged user (i.e., one with neither the CAP_SYS_RESOURCE nor the CAP_SYS_ADMIN capability). So long as the total number of pages allocated to pipe buffers for this user is at this limit, attempts to create new pipes will be de? nied, and attempts to increase a pipe's capacity will be denied.

When the value of this limit is zero (which is the default), no hard limit is applied.

/proc/sys/fs/pipe-user-pages-soft (since Linux 4.5)

The soft limit on the total size (in pages) of all pipes created or set by a single unprivileged user (i.e., one with neither the CAP_SYS_RESOURCE nor the CAP_SYS_ADMIN capability). So long as the total number of pages allocated to pipe buffers for this user is at this limit, individual pipes created by a user will be limited to one page, and attempts to increase a pipe's capac? ity will be denied.

When the value of this limit is zero, no soft limit is applied.

The default value for this file is 16384, which permits creating up to 1024 pipes with the default capacity.

Before Linux 4.9, some bugs affected the handling of the pipe-user-pages-soft and pipe-user-pages-hard limits; see BUGS.

PIPE_BUF

POSIX.1 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 nonatomic: the kernel may in? terleave the data with data written by other processes. POSIX.1 re? quires 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 nonblocking (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 nonatomic: the data given to write(2) may be inter? leaved with write(2)s by other process; the write(2) blocks un? til n bytes have been written.

O_NONBLOCK enabled, n > PIPE_BUF

If the pipe is full, then write(2) fails, with errno set to EA? GAIN. 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 writ? ten), 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. The target for delivery of signals must be set using the fc? ntl(2) F_SETOWN command. 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. POSIX.1 requires only unidirectional pipes. Portable applications should avoid reliance on bidirectional pipe semantics.

BUGS

Before Linux 4.9, some bugs affected the handling of the pipe-user-pages-soft and pipe-user-pages-hard limits when using the fcntl(2) F_SETPIPE_SZ operation to change a pipe's capacity:

(1) When increasing the pipe capacity, the checks against the soft and hard limits were made against existing consumption, and excluded the memory required for the increased pipe capacity. The new in? crease in pipe capacity could then push the total memory used by the user for pipes (possibly far) over a limit. (This could also trigger the problem described next.)

Starting with Linux 4.9, the limit checking includes the memory required for the new pipe capacity.

(2) The limit checks were performed even when the new pipe capacity was less than the existing pipe capacity. This could lead to problems if a user set a large pipe capacity, and then the limits were lowered, with the result that the user could no longer de? crease the pipe capacity.

Starting with Linux 4.9, checks against the limits are performed only when increasing a pipe's capacity; an unprivileged user can always decrease a pipe's capacity.

(3) The accounting and checking against the limits were done as fol? lows:

(a) Test whether the user has exceeded the limit.

(b) Make the new pipe buffer allocation.

(c) Account new allocation against the limits.

This was racey. Multiple processes could pass point (a) simulta? neously, and then allocate pipe buffers that were accounted for only in step (c), with the result that the user's pipe buffer al? location could be pushed over the limit.

Starting with Linux 4.9, the accounting step is performed before doing the allocation, and the operation fails if the limit would be exceeded.

Before Linux 4.9, bugs similar to points (1) and (3) could also occur when the kernel allocated memory for a new pipe buffer; that is, when calling pipe(2) and when opening a previously unopened FIFO.

SEE ALSO

mkfifo(1), dup(2), fcntl(2), open(2), pipe(2), poll(2), select(2), socketpair(2), splice(2), stat(2), tee(2), vmsplice(2), mkfifo(3), epoll(7), fifo(7)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.