

## NAME

pipe - create an interprocess channel

## SYNOPSIS

```
#include <unistd.h>
```

```
int pipe(int fildes[2]);
```

## DESCRIPTION

The pipe() function creates an I/O mechanism called a pipe and returns two file descriptors, `_f_ i_ l_ d_ e_ s[0]` and `_f_ i_ l_ d_ e_ s[1]`. The files associated with `_f_ i_ l_ d_ e_ s[0]` and `_f_ i_ l_ d_ e_ s[1]` are streams and are both opened for reading and writing.

The `O_NDELAY` and `O_NONBLOCK` flags are cleared.

A read from `_f_ i_ l_ d_ e_ s[0]` accesses the data written to `_f_ i_ l_ d_ e_ s[1]` on a first-in-first-out (FIFO) basis and a read from `_f_ i_ l_ d_ e_ s[1]` accesses the data written to `_f_ i_ l_ d_ e_ s[0]` also on a FIFO basis.

The `FD_CLOEXEC` flag will be clear on both file descriptors.

Upon successful completion pipe() marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the pipe.

## RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

## ERRORS

The pipe() function will fail if:

## EMFILE

There are `OPEN_MAX-1` or more file descriptors currently open for this process.

## ENFILE

A file table entry could not be allocated.

## NOTES

Since a pipe is bi-directional, there are two separate flows of data. Therefore, the size (`st_size`) returned by a call to `fstat(2)` with argument `_f_ i_ l_ d_ e_ s[0]` or `_f_ i_ l_ d_ e_ s[1]` is the number of bytes available for reading from `_f_ i_ l_ d_ e_ s[0]` or `_f_ i_ l_ d_ e_ s[1]` respectively. Previously, the size (`st_size`) returned by a call to `fstat()` with argument `_f_ i_ l_ d_ e_ s[1]` (the write-end) was the number of bytes available for reading from `_f_ i_ l_ d_ e_ s[0]`

(the read-end).

System Calls read(2)

## NAME

read, readv, pread - read from file

## SYNOPSIS

```
#include <unistd.h>
```

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

## DESCRIPTION

The read() function attempts to read `_n_b_y_t_e` bytes from the file associated with the open file descriptor, `_f_i_l_d_e_s`, into the buffer pointed to by `_b_u_f`.

If `_n_b_y_t_e` is 0, read() will return 0 and have no other results.

On files that support seeking (for example, a regular file), the read() starts at a position in the file given by the file offset associated with `_f_i_l_d_e_s`. The file offset is incremented by the number of bytes actually read.

Files that do not support seeking (for example, terminals) always read from the current position. The value of a file offset associated with such a file is undefined.

If `_f_i_l_d_e_s` refers to a socket, read() is equivalent to `recv(3SOCKET)` with no flags set.

No data transfer will occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 will be returned. If the file refers to a device special file, the result of subsequent read() requests is implementation-dependent.

If the value of `_n_b_y_t_e` is greater than `SSIZE_MAX`, the result is implementation-dependent.

When attempting to read from a regular file with mandatory file/record locking set (see `chmod(2)`), and there is a write lock owned by another process on the segment of the file to be read:

- + o If `O_NDELAY` or `O_NONBLOCK` is set, read() returns -1 and sets `errno` to `EAGAIN`.
- + o If `O_NDELAY` and `O_NONBLOCK` are clear, read() sleeps until the blocking record lock is removed.

When attempting to read from an empty pipe (or FIFO):

- + o If no process has the pipe open for writing, read() returns 0 to indicate end-of-file.
- + o If some process has the pipe open for writing and O\_NDELAY is set, read() returns 0.
- + o If some process has the pipe open for writing and O\_NONBLOCK is set, read() returns -1 and sets errno to EAGAIN.
- + o If O\_NDELAY and O\_NONBLOCK are clear, read() blocks until data is written to the pipe or the pipe is closed by all processes that had opened the pipe for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- + o If O\_NDELAY is set, read() returns 0.
- + o If O\_NONBLOCK is set, read() returns -1 and sets errno to EAGAIN.
- + o If O\_NDELAY and O\_NONBLOCK are clear, read() blocks until data become available.

When attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO, or a terminal, and the file has no data currently available:

- + o If O\_NDELAY or O\_NONBLOCK is set, read() returns -1 and sets errno to EAGAIN.
- + o If O\_NDELAY and O\_NONBLOCK are clear, read() blocks until data becomes available.

The read() function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, read() returns bytes with value 0. For example, lseek(2) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data will return bytes with value 0 until data is written into the gap.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with `_f_i_l_d_e_s`.

Upon successful completion, where `_n_b_y_t_e` is greater than 0, read() will mark for update the `st_atime` field of the file, and return the number of bytes read. This number will never be greater than `_n_b_y_t_e`. The value returned may be less than `_n_b_y_t_e` if the number of bytes left in the file is less than `_n_b_y_t_e`, if the read() request was interrupted by a signal, or

if the file is a pipe or FIFO or special file and has fewer than `_n_b_y_t_e` bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

If a `read()` is interrupted by a signal before it reads any data, it will return `-1` with `errno` set to `EINTR`.

If a `read()` is interrupted by a signal after it has successfully read some data, it will return the number of bytes read.

A `read()` from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl(2)` request, and can be tested with the `I_GRDOPT` `ioctl()`. In byte-stream mode, `read()` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read()` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read()` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read()` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read()` returns is discarded, and is not available for a subsequent `read()`, `readv()` or `getmsg(2)` call.

How `read()` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read()` accepts data until it has read `_n_b_y_t_e` bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read()` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read()`, `readv()` or `getmsg(2)`. In message-nondiscard mode or message-discard

mode, a zero-byte message returns `0` and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and `0` is returned, regardless of the read mode.

A `read()` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read()` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The

read() fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the L\_SRDOPT ioctl() command. In control-data mode, read() converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, read() discards message control parts but returns to the process any data part in the message.

In addition, read() and readv() will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of read() or readv() but reflects the prior error. If a hangup occurs on the STREAM being read, read() continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns 0.

#### RETURN VALUES

Upon successful completion, read() and readv() return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions return -1 and set errno to indicate the error.

#### ERRORS

The read(), readv(), and pread() functions will fail if:

##### EAGAIN

Mandatory file/record locking was set, O\_NDELAY or O\_NONBLOCK was set, and there was a blocking record lock; total amount of system memory available when reading using raw I/O is temporarily insufficient; no data is waiting to be read on a file associated with a tty device and O\_NONBLOCK was set; or no message is waiting to be read on a stream and O\_NDELAY or O\_NONBLOCK was set.

EBADF The `_f_i_l_d_e_s` argument is not a valid file descriptor open for reading.

##### EBADMSG

Message waiting to be read on a stream is not a data message.

##### EDEADLK

The read was going to go to sleep and cause a deadlock to occur.

##### EFAULT

The `_b_u_f` argument points to an illegal address.

EINTR A signal was caught during the read operation and no data was transferred.

##### EINVAL

An attempt was made to read from a stream linked to a

multiplexor.

EIO A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned.

#### EISDIR

The `_f_i_l_d_e_s` argument refers to a directory on a file system type that does not support read operations on directories.

#### ENOLCK

The system record lock table was full, so the `read()` or `readv()` could not go to sleep until the blocking record lock was removed.

#### ENOLINK

The `_f_i_l_d_e_s` argument is on a remote machine and the link to that machine is no longer active.

ENXIO The device associated with `_f_i_l_d_e_s` is a block special or character special file and the value of the file pointer is out of range.

The `read()` and `readv()` functions will fail if:

#### EOVERFLOW

The file is a regular file, `_n_b_y_t_e` is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with `_f_i_l_d_e_s`.

System Calls `write(2)`

#### NAME

`write`, `pwrite`, `writew` - write on a file

#### SYNOPSIS

```
#include <unistd.h>
```

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

#### DESCRIPTION

The write() function attempts to write `_n_b_y_t_e` bytes from the buffer pointed to by `_b_u_f` to the file associated with the open file descriptor, `_f_i_l_d_e_s`.

If `_n_b_y_t_e` is 0, write() will return 0 and have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file offset associated with `_f_i_l_d_e_s`. Before successful return from write(), the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file will be set to this file offset.

If the O\_SYNC flag of the file status flags is set and `_f_i_l_d_e_s` refers to a regular file, a successful write() does not return until the data is delivered to the underlying hardware.

If `_f_i_l_d_e_s` refers to a socket, write() is equivalent to send(3SOCKET) with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the O\_APPEND flag of the file status flags is set, the file offset will be set to the end of the file prior to each write and no intervening file modification operation will occur between changing the file offset and the write operation.

For regular files, no data transfer will occur past the offset maximum established in the open file description with `_f_i_l_d_e_s`.

A write() to a regular file is blocked if mandatory file/record locking is set (see chmod(2)), and there is a record lock owned by another process on the segment of the file to be written:

- + o If O\_NDELAY or O\_NONBLOCK is set, write() returns -1 and sets errno to EAGAIN.
- + o If O\_NDELAY and O\_NONBLOCK are clear, write() sleeps until all blocking locks are removed or the write() is terminated by a signal.

If a write() requests that more bytes be written than there is room for—for example, if the write would exceed the pro-

cess file size limit (see `getrlimit(2)` and `ulimit(2)`), the system file size limit, or the free space on the device-only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A `write()` of 512-bytes returns 20. The next `write()` of a non-zero number of bytes gives a failure return (except as noted for pipes and FIFO below).

If `write()` is interrupted by a signal before it writes any data, it will return -1 with `errno` set to `EINTR`.

If `write()` is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

If the value of `_n_b_y_t_e` is greater than `SSIZE_MAX`, the result is implementation-dependent.

After a `write()` to a regular file has successfully returned:

- + o Any successful `read(2)` from each byte position in the file that was modified by that write will return the data specified by the `write()` for that position until such byte positions are again modified.
- + o Any subsequent successful `write()` to the same byte position in the file will overwrite that file data.

Write requests to a pipe or FIFO are handled the same as a regular file with the following exceptions:

- + o There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
- + o Write requests of `{PIPE_BUF}` bytes or less are guaranteed not to be interleaved with data from other processes doing writes on the same pipe. Writes of greater than `{PIPE_BUF}` bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the `O_NONBLOCK` or `O_NDELAY` flags are set.
- + o If `O_NONBLOCK` and `O_NDELAY` are clear, a write request may cause the process to block, but on normal completion it returns `_n_b_y_t_e`.
- + o If `O_NONBLOCK` and `O_NDELAY` are set, `write()` does not block the process. If a `write()` request for `PIPE_BUF` or fewer bytes succeeds completely `write()` returns `_n_b_y_t_e`. Otherwise, if `O_NONBLOCK` is set, it returns -1 and sets `errno` to `EAGAIN` or if `O_NDELAY` is set, it returns 0. A `write()` request for greater than `{PIPE_BUF}` bytes transfers what it can and returns the number of bytes written or it transfers no data and, if `O_NONBLOCK` is set, returns -1 with `errno` set to `EAGAIN` or if `O_NDELAY` is set, it returns 0.

Finally, if a request is greater than PIPE\_BUF bytes and all data previously written to the pipe has been read, write() transfers at least PIPE\_BUF bytes.

When attempting to write to a file descriptor (other than a pipe, a FIFO, a socket, or a STREAM) that supports nonblocking writes and cannot accept the data immediately:

- + o If O\_NONBLOCK and O\_NDELAY are clear, write() blocks until the data can be accepted.
- + o If O\_NONBLOCK or O\_NDELAY is set, write() does not block the process. If some data can be written without blocking the process, write() writes what it can and returns the number of bytes written. Otherwise, if O\_NONBLOCK is set, it returns -1 and sets errno to EAGAIN or if O\_NDELAY is set, it returns 0.

Upon successful completion, where `count` is greater than 0, write() will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the `S_ISUID` and `S_ISGID` bits of the file mode may be cleared.

For STREAMS files (see `intro(3)` and `streamio(7I)`), the operation of write() is determined by the values of the minimum and maximum `count` range ("packet size") accepted by the STREAM. These values are contained in the topmost STREAM module, and can not be set or tested from user level. If `count` falls within the packet size range, `count` bytes are written. If `count` does not fall within the range and the minimum packet size value is zero, write() breaks the buffer into maximum packet size segments prior to sending the data downstream (the last segment may be smaller than the maximum packet size). If `count` does not fall within the range and the minimum value is non-zero, write() fails and sets errno to ERANGE. Writing a zero-length buffer (`count` is zero) to a STREAMS device sends a zero length message with zero returned. However, writing a zero-length buffer to a pipe or FIFO sends no message and zero is returned. The user program may issue the `I_SWROPT` ioctl(2) to enable zero-length messages to be sent across the pipe or FIFO (see `streamio(7I)`).

When writing to a STREAM, data messages are created with a priority band of zero. When writing to a socket or to a STREAM that is not a pipe or a FIFO:

- + o If O\_NDELAY and O\_NONBLOCK are not set, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), write() blocks until data can be accepted.
- + o If O\_NDELAY or O\_NONBLOCK is set and the STREAM cannot accept data, write() returns -1 and sets errno to EAGAIN.

- + o If O\_NDELAY or O\_NONBLOCK is set and part of the buffer has already been written when a condition occurs in which the STREAM cannot accept additional data, write() terminates and returns the number of bytes written.

## RETURN VALUES

Upon successful completion, write() returns the number of bytes actually written to the file associated with `_f_i_l_d_e_s`. This number is never greater than `_n_b_y_t_e`. Otherwise, `-1` is returned, the file-pointer remains unchanged, and `errno` is set to indicate the error.

Upon successful completion, writev() returns the number of bytes actually written. Otherwise, it returns `-1`, the file-pointer remains unchanged, and `errno` is set to indicate an error.

## ERRORS

The write(), pwrite(), and writev() functions will fail if:

### EAGAIN

Mandatory file/record locking is set, O\_NDELAY or O\_NONBLOCK is set, and there is a blocking record lock; an attempt is made to write to a STREAM that can not accept data with the O\_NDELAY or O\_NONBLOCK flag set; or a write to a pipe or FIFO of PIPE\_BUF bytes or less is requested and less than `_n_b_y_t_e_s` of free space is available.

EBADF The `_f_i_l_d_e_s` argument is not a valid file descriptor open for writing.

### EDEADLK

The write was going to go to sleep and cause a deadlock situation to occur.

### EDQUOT

The user's quota of disk blocks on the file system containing the file has been exhausted.

## FAULT

The `_b_u_f` argument points to an illegal address.

EFBIG An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see `getrlimit(2)` and `ulimit(2)`).

EFBIG The file is a regular file, `_n_b_y_t_e` is greater than 0, and the starting position is greater than or equal to the offset maximum established in the file description associated with `_f_i_l_d_e_s`.

EINTR A signal was caught during the write operation and no data was transferred.

**EIO** The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag is set, or the process is neither ignoring nor blocking SIGTTOU signals and the process group of the process is orphaned.

#### **ENOLCK**

Enforced record locking was enabled and {LOCK\_MAX} regions are already locked in the system, or the system record lock table was full and the write could not go to sleep until the blocking record lock was removed.

#### **ENOLINK**

The `_f_i_l_d_e_s` argument is on a remote machine and the link to that machine is no longer active.

#### **ENOSPC**

During a write to an ordinary file, there is no free space left on the device.

**ENOSR** An attempt is made to write to a STREAMS with insufficient STREAMS memory resources available in the system.

**ENXIO** A hangup occurred on the STREAM being written to.

**EPIPE** An attempt is made to write to a pipe or a FIFO that is not open for reading by any process, or that has only one end open (or to a file descriptor created by `socket(3SOCKET)`, using type `SOCK_STREAM` that is no longer connected to a peer endpoint). A `SIGPIPE` signal will also be sent to the process. The process dies unless special provisions were taken to catch or ignore the signal.

#### **ERANGE**

### **NAME**

`dup` - duplicate an open file descriptor

### **SYNOPSIS**

```
#include <unistd.h>
```

```
int dup(int fildes);
```

### **DESCRIPTION**

The `dup()` function returns a new file descriptor having the following in common with the original open file descriptor

`_f_i_l_d_e_s`:

- + o same open file (or pipe)

- + o same file pointer (that is, both file descriptors share one file pointer)
- + o same access mode (read, write or read/write).

The new file descriptor is set to remain open across `_e_x_e_c` functions (see `fcntl(2)`).

The file descriptor returned is the lowest one available.

The `dup(_f_ i_ l_ d_ e_ s)` function call is equivalent to:

```
fcntl(_f_ i_ l_ d_ e_ s, F_DUPFD, 0)
```

#### RETURN VALUES

Upon successful completion, a non-negative integer representing the file descriptor is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

#### ERRORS

The `dup()` function will fail if:

**EBADF** The `_f_ i_ l_ d_ e_ s` argument is not a valid open file descriptor.

**EINTR** A signal was caught during the execution of the `dup()` function.

#### EMFILE

The process has too many open files (see `getrlimit(2)`).

#### ENOLINK

The `_f_ i_ l_ d_ e_ s` argument is on a remote machine and the link to that machine is no longer active.

Standard C Library Functions dup2(3C)

#### NAME

`dup2` - duplicate an open file descriptor

#### SYNOPSIS

```
#include <unistd.h>
```

```
int dup2(int fildes, int fildes2);
```

#### DESCRIPTION

The `dup2()` function causes the file descriptor `_f_i_l_d_e_s_2` to refer to the same file as `_f_i_l_d_e_s`. The `_f_i_l_d_e_s` argument is a file descriptor referring to an open file, and `_f_i_l_d_e_s_2` is a non-negative integer less than the current value for the maximum number of open file descriptors allowed the calling process. See `getrlimit(2)`. If `_f_i_l_d_e_s_2` already refers to an open file, not `_f_i_l_d_e_s`, it is closed first. If `_f_i_l_d_e_s_2` refers to `_f_i_l_d_e_s`, or if `_f_i_l_d_e_s` is not a valid open file descriptor, `_f_i_l_d_e_s_2` will not be closed first.

The `dup2()` function is equivalent to `fcntl(_f_i_l_d_e_s, F_DUP2FD, _f_i_l_d_e_s_2)`.

#### RETURN VALUES

Upon successful completion a non-negative integer representing the file descriptor is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

#### ERRORS

The `dup2()` function will fail if:

**EBADF** The `_f_i_l_d_e_s` argument is not a valid open file descriptor.

**EBADF** The `_f_i_l_d_e_s_2` argument is negative or is not less than the current resource limit returned by `getrlimit(RLIMIT_NOFILE, ...)`.

**EINTR** A signal was caught during the `dup2()` call.

#### EMFILE

The process has too many open files. See `fcntl(2)`.

Standard C Library Functions

`popen(3C)`

#### NAME

`popen`, `pclose` - initiate a pipe to or from a process

#### SYNOPSIS

```
#include <stdio.h>
```

```
FILE *popen(const char *command, const char *mode);
```

```
int pclose(FILE *stream);
```

#### DESCRIPTION

The `popen()` function creates a pipe between the calling program and the command to be executed. The arguments to `popen()` are pointers to null-terminated strings.

The `_c_o_m_m_a_n_d` argument consists of a shell command line. The `_m_o_d_e` argument is an I/O mode, either `r` for reading or `w` for

writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is `_w`, by writing to the file `_s_t_r_e_a_m` (see `intro(3)`); and one can read from the standard output of the command, if the I/O mode is `r`, by reading from the file `_s_t_r_e_a_m`. Because open files are shared, a type `r` command may be used as an input filter and a type `_w` as an output filter.

The environment of the executed command will be as if a child process were created within the `popen()` call using `fork(2)`. If the application is standard-conforming (see `standards(5)`), the child is invoked with the call:

```
execl("/usr/bin/ksh", "ksh", "-c", _c_o_m_m_a_n_d, (char *)0);
```

otherwise, the child is invoked with the call:

```
execl("/usr/bin/sh", "sh", "-c", _c_o_m_m_a_n_d, (char *)0);
```

A stream opened by `popen()` should be closed by `pclose()`, which closes the pipe, and waits for the associated process to terminate and returns the termination status of the process running the command language interpreter. This is the value returned by `waitpid(2)`. See `wstat(3XFN)` for more information on termination status.

## RETURN VALUES

The `popen()` function returns a null pointer if files or processes cannot be created.

The `pclose()` function returns the termination status of the command. It returns `-1` if `_s_t_r_e_a_m` is not associated with a `popen()` command and sets `errno` to indicate the error.

## ERRORS

The `popen()` function may fail if:

### EMFILE

There are currently `FOPEN_MAX` or `STREAM_MAX` streams open in the calling process.

### EINVAL

The `_m_o_d_e` argument is invalid.

The `pclose()` function will fail if:

### ECHILD

The status of the child process could not be obtained, as described above.

The `popen()` function may also set `errno` values as described by `fork(2)` or `pipe(2)`.

## USAGE

If the original and `popen()` processes concurrently read or write a common file, neither should use buffered I/O. Prob-

lems with an output filter may be forestalled by careful buffer flushing, for example, with fflush() (see fclose(3C)). A security hole exists through the IFS and PATH environment variables. Full pathnames should be used (or PATH reset) and IFS should be set to space and tab (" \t").

## EXAMPLES

Example 1: popen() example

The following program will print on the standard output (see stdio(3C)) the names of files in the current directory with a .c suffix.

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    char *cmd = "/usr/bin/ls *.c";
    char buf[BUFSIZ];
    FILE *ptr;

    if ((ptr = popen(cmd, "r")) != NULL)
        while (fgets(buf, BUFSIZ, ptr) != NULL)
            (void) printf("%s", buf);
    return 0;
}
```

Maintenance Commands

mknod(1M)

## NAME

mknod - make a special file

## SYNOPSIS

```
mknod _n_a_m_e b _m_a_j_o_r
_m_ i_n_o_r
```

```
mknod _n_a_m_e c _m_a_j_o_r
_m_ i_n_o_r
```

```
mknod _n_a_m_e p
```

## DESCRIPTION

mknod makes a directory entry for a special file.

## OPTIONS

The following options are supported:

b Create a block-type special file.

- c Create a character-type special file.
- p Create a FIFO (named pipe).

#### OPERANDS

The following operands are supported:

- \_ m\_ a\_ j\_ o\_ r The \_ m\_ a\_ j\_ o\_ r device number.
- \_ m\_ i\_ n\_ o\_ r The \_ m\_ i\_ n\_ o\_ r device number; can be either decimal or octal. The assignment of major device numbers is specific to each system. You must be the super-user to use this form of the command.
- \_ n\_ a\_ m\_ e A special file to be created.

#### USAGE

See `largefile(5)` for the description of the behavior of `mknod` when encountering files greater than or equal to 2 Gbyte (  $2^{31}$  bytes).

#### NOTES

If `mknod(2)` is used to create a device, the major and minor device numbers are always interpreted by the kernel running on that machine.

With the advent of physical device naming, it would be preferable to create a symbolic link to the physical name of the device (in the `/devices` subtree) rather than using `mknod`.

Maintenance Commands

`mkfifo(1M)`

#### NAME

`mkfifo` - make FIFO special file

#### SYNOPSIS

```
/usr/bin/mkfifo [ -m _ m_ o_ d_ e ] _ p_ a_ t_ h ...
```

#### DESCRIPTION

The `mkfifo` command creates the FIFO special files named by its argument list. The arguments are taken sequentially, in the order specified; and each FIFO special file is either created completely or, in the case of an error or signal, not created at all.

If errors are encountered in creating one of the special files, `mkfifo` writes a diagnostic message to the standard error and continues with the remaining arguments, if any.

The `mkfifo` command calls the library routine `mkfifo(3C)`, with the `_p_a_t_h` argument is passed as the `_p_a_t_h` argument from the command line, and `_m_o_d_e` is set to the equivalent of `a=rw`, modified by the current value of the file mode creation mask `umask(1)`.

## OPTIONS

The following option is supported:

`-m _m_o_d_e`

Set the file permission bits of the newly-created FIFO to the specified `_m_o_d_e` value. The `_m_o_d_e` option-argument will be the same as the `_m_o_d_e` operand defined for the `chmod(1)` command. In `<_s_y_m_b_o_l_ i_ cmode>` strings, the `_o_p` characters `+` and `-` will be interpreted relative to an assumed initial mode of `a=rw`.

## OPERANDS

The following operand is supported:

`file` A path name of the FIFO special file to be created.

## USAGE

See `largefile(5)` for the description of the behavior of `mkfifo` when encountering files greater than or equal to 2 Gbyte (  $2^{31}$  bytes).

## ENVIRONMENT VARIABLES

See `environ(5)` for descriptions of the following environment variables that affect the execution of `mkfifo`: `LC_CTYPE`, `LC_MESSAGES`, and `NLSPATH`.

## EXIT STATUS

The following exit values are returned:

System Calls `mknod(2)`

## NAME

`mknod` - make a directory, or a special or ordinary file

## SYNOPSIS

```
#include <sys/stat.h>
```

```
int mknod(const char *path, mode_t mode, dev_t dev);
```

## DESCRIPTION

The `mknod()` function creates a new file named by the `path` name pointed to by `_p_a_t_h`. The file type and permissions of the new file are initialized from `_m_o_d_e`.

The file type is specified in `_m_o_d_e` by the `S_IFMT` bits, which must be set to one of the following values:

S\_IFIFO  
fifo special

S\_IFCHR  
character special

S\_IFDIR  
directory

S\_IFBLK  
block special

S\_IFREG  
ordinary file

The file access permissions are specified in `_m_o_d_e` by the 0007777 bits, and may be constructed by a bitwise OR operation of the following values:

|              |       |   |
|--------------|-------|---|
| S_ISUID      | 04000 | Set user ID on execution.   |
| S_ISGID      | 020#0 | Set group ID on execution if # is 7, 5, 3, or 1. Enable mandatory file/record locking if # is 6, 4, 2, or 0 |
| S_ISVTX      | 01000 | On directories, restricted deletion flag; on regular files on a UFS file system, do not cache flag.         |
| S_IRWXU      | 00700 | Read, write, execute by owner.  |
| S_IRUSR      | 00400 | Read by owner.  |
| S_IWUSR      | 00200 | Write by owner.   |
| S_IXUSR      | 00100 | Execute (search if a directory) by owner.   |
| S_IRWXG      | 00070 | Read, write, execute by group.  |
| S_IRGRP      | 00040 | Read by group.  |
| S_IWGRP      | 00020 | Write by group.   |
| S_IXGRP      | 00010 | Execute by group.   |
| System Calls |       | <code>mknod(2)</code>   |

|         |       |  |
|---------|-------|--|
| S_IRWXO | 00007 | Read, write, execute (search) by others. |
| S_IROTH | 00004 | Read by others.                          |
| S_IWOTH | 00002 | Write by others                          |
| S_IXOTH | 00001 | Execute by others.                       |

The owner ID of the file is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process. However, if the `S_ISGID` bit is set in the parent directory, then the group ID of the file is inherited from the parent. If the group ID of the new file does not match the effective group ID or one of the supplementary group IDs, the `S_ISGID` bit is cleared.

The access permission bits of `_m_o_d_e` are modified by the

process's file mode creation mask: all bits set in the process's file mode creation mask are cleared (see `umask(2)`). If `_m_o_d_e` indicates a block or character special file, `_d_e_v` is a configuration-dependent specification of a character or block I/O device. If `_m_o_d_e` does not indicate a block special or character special device, `_d_e_v` is ignored. See `makedev(3C)`.

If `_p_a_t_h` is a symbolic link, it is not followed.

## RETURN VALUES

Upon successful completion, `mknod()` returns 0. Otherwise, it returns -1, the new file is not created, and `errno` is set to indicate the error.

## ERRORS

The `mknod()` function will fail if:

### EACCES

A component of the path prefix denies search permission, or write permission is denied on the parent directory.

### EDQUOT

The directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of inodes on the file system where the file is being created has been exhausted.

### EEXIST

The named file exists.

### EFAULT

The `_p_a_t_h` argument points to an illegal address.

**EINTR** A signal was caught during the execution of the `mknod()` function.

### EINVAL

An invalid argument exists.

**EIO** An I/O error occurred while accessing the file system.

**ELOOP** Too many symbolic links were encountered in translating `_p_a_t_h`.

### ENAMETOOLONG

The length of the `_p_a_t_h` argument exceeds `PATH_MAX`, or the length of a `_p_a_t_h` component exceeds `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect.

### ENOENT

A component of the path prefix specified by `_p_a_t_h` does not name an existing directory or `_p_a_t_h` is an empty string.

#### ENOLINK

The `_p_a_t_h` argument points to a remote machine and the link to that machine is no longer active.

#### ENOSPC

The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.

#### ENOTDIR

A component of the path prefix is not a directory.

**EPERM** The effective user of the calling process is not super-user.

**EROFS** The directory in which the file is to be created is located on a read-only file system.

The `mknod()` function may fail if:

#### ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

#### USAGE

Normally, applications should use the `mkdir(2)` routine to make a directory, since the function `mknod()` may not establish directory entries for the directory itself (`.`) and the parent directory (`..`), and appropriate permissions are not required. Similarly, `mkfifo(3C)` should be used in place of `mknod()` in order to create FIFOs.

The `mknod()` function may be invoked only by a privileged user for file types other than FIFO special.

