Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'read.3p' command

$ man read.3p

READ(3P) POSIX Programmer's Manual READ(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

pread, read ? read from a file

SYNOPSIS

#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

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

DESCRIPTION

The read() function shall attempt to read nbyte bytes from the file as? sociated with the open file descriptor, fildes, into the buffer pointed to by buf. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action described below is taken, and if nbyte is zero, the read() function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the read() function shall return zero and have no other results.

On files that support seeking (for example, a regular file), the read() shall start at a position in the file given by the file offset associ?

ated with fildes. The file offset shall be 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.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned.

If the file refers to a device special file, the result of subsequent read() requests is implementation-defined.

If the value of nbyte is greater than {SSIZE_MAX}, the result is imple? mentation-defined.

When attempting to read from an empty pipe or FIFO:

* If no process has the pipe open for writing, read() shall return 0 to indicate end-of-file.
* If some process has the pipe open for writing and O_NONBLOCK is set, read() shall return -1 and set errno to [EAGAIN].
* If some process has the pipe open for writing and O_NONBLOCK is clear, read() shall block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that sup? ports non-blocking reads and has no data currently available:

* If O_NONBLOCK is set, read() shall return -1 and set errno to [EA? GAIN].
* If O_NONBLOCK is clear, read() shall block the calling thread until some data becomes available.
* The use of the O_NONBLOCK flag has no effect if there is some data 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 writ? ten, read() shall return bytes with value 0. For example, lseek() al? lows 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 shall return bytes with value 0 until data is written into the gap.

Upon successful completion, where nbyte is greater than 0, read() shall mark for update the last data access timestamp of the file, and shall return the number of bytes read. This number shall never be greater than nbyte. The value returned may be less than nbyte if the number of bytes left in the file is less than nbyte, 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 nbyte 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 shall return -1 with errno set to [EINTR].

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

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

If fildes refers to a socket, read() shall be equivalent to recv() with no flags set.

If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If the O_SYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

If fildes refers to a shared memory object, the result of the read() function is unspecified.

If fildes refers to a typed memory object, the result of the read() function is unspecified.

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 shall be byte-stream mode. This can be changed using the I_SRDOPT ioctl() request, and can be tested with I_GRDOPT ioctl(). In byte-stream mode, read() shall retrieve 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() shall retrieve data until as many bytes as were requested are transferred, or until a message bound? ary is reached. If read() does not retrieve all the data in a message, the remaining data shall be 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 shall be discarded, and shall not be available for a subsequent read(), getmsg(), or getpmsg() call.

How read() handles zero-byte STREAMS messages is determined by the cur? rent read mode setting. In byte-stream mode, read() shall accept data until it has read nbyte bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The read() function shall then return the number of bytes read, and place the zero-byte message back on the STREAM to be retrieved by the next read(), getmsg(), or getpmsg(). In message-nondiscard mode or message-discard mode, a zero-byte message shall return 0 and the message shall be re? moved from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message shall be removed from the STREAM and 0 shall be returned, regardless of the read mode.

A read() from a STREAMS file shall return 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() shall fail if a message con? taining 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 I_SRDOPT ioctl() command. In control-data mode, read() shall convert any control part to data and pass it to the application before passing any data part originally present in the

same message. In control-discard mode, read() shall discard message control parts but return to the process any data part in the message.

In addition, read() shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno shall not reflect the result of read(), but reflect the prior error. If a hangup occurs on the STREAM being read, read() shall continue to op? erate normally until the STREAM head read queue is empty. Thereafter, it shall return 0.

The pread() function shall be equivalent to read(), except that it shall read from a given position in the file without changing the file offset. The first three arguments to pread() are the same as read() with the addition of a fourth argument offset for the desired position inside the file. An attempt to perform a pread() on a file that is in? capable of seeking shall result in an error.

RETURN VALUE

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

ERRORS

These functions shall fail if:

EAGAIN The file is neither a pipe, nor a FIFO, nor a socket, the O_NON? BLOCK flag is set for the file descriptor, and the thread would be delayed in the read operation.

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

EBADMSG

The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.

EINTR The read operation was terminated due to the receipt of a sig? nal, and no data was transferred.

EINVAL The STREAM or multiplexer referenced by fildes is linked (di? rectly or indirectly) downstream from a multiplexer.

EIO The process is a member of a background process group attempting

to read from its controlling terminal, and either the calling thread is blocking SIGTTIN or the process is ignoring SIGTTIN or the process group of the process is orphaned. This error may also be generated for implementation-defined reasons.

EISDIR The fildes argument refers to a directory and the implementation does not allow the directory to be read using read() or pread(). The readdir() function should be used instead.

EOVERFLOW

The file is a regular file, nbyte is greater than 0, the start? ing position is before the end-of-file, and the starting posi? tion is greater than or equal to the offset maximum established in the open file description associated with fildes.

The pread() function shall fail if:

EINVAL The file is a regular file or block special file, and the offset argument is negative. The file offset shall remain unchanged.

ESPIPE The file is incapable of seeking.

The read() function shall fail if:

EAGAIN The file is a pipe or FIFO, the O_NONBLOCK flag is set for the file descriptor, and the thread would be delayed in the read op? eration.

EAGAIN or EWOULDBLOCK

The file is a socket, the O_NONBLOCK flag is set for the file descriptor, and the thread would be delayed in the read opera? tion.

ECONNRESET

A read was attempted on a socket and the connection was forcibly closed by its peer.

ENOTCONN

A read was attempted on a socket that is not connected.

ETIMEDOUT

A read was attempted on a socket and a transmission timeout oc? curred.

These functions may fail if:

EIO A physical I/O error has occurred.

ENOBUFS

Insufficient resources were available in the system to perform the operation.

ENOMEM Insufficient memory was available to fulfill the request.

ENXIO A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The following sections are informative.

EXAMPLES

Reading Data into a Buffer

The following example reads data from the file associated with the file descriptor fd into the buffer pointed to by buf.

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

APPLICATION USAGE

None.

RATIONALE

This volume of POSIX.1?2017 does not specify the value of the file off? set after an error is returned; there are too many cases. For program? ming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EA? GAIN], clearly the offset should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

Note that a read() of zero bytes does not modify the last data access timestamp. A read() that requests more than zero bytes, but returns zero, is required to modify the last data access timestamp.

Implementations are allowed, but not required, to perform error check? ing for read() requests of zero bytes.

Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as lread() and lwrite() (using and returning longs) were considered at one time. The current solution is to use abstract types on the ISO C standard function to read() and write(). The abstract types can be declared so that existing functions work, but can also be declared so that larger types can be represented in future implementa? tions. It is presumed that whatever constraints limit the maximum range of size_t also limit portable I/O requests to the same range. This vol? ume of POSIX.1?2017 also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an int can hold.

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of {PIPE_BUF} or any other size that would be an aid to applications portability.

This volume of POSIX.1?2017 requires that no action be taken for read() or write() when nbyte is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file de? scriptors). This is consistent with the rest of this volume of POSIX.1?2017, but the phrasing here could be misread to require detec? tion of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O oper? ations. It is a known attribute of terminals that this is not honored,

and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

There were recommendations to add format parameters to read() and write() in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was de? termined that this should correspond with similar C-language facili? ties, and that is beyond the scope of this volume of POSIX.1?2017. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

In 4.3 BSD, a read() or write() that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth Edition, there is an additional function, select(), whose purpose is to pause until speci? fied activity (data to read, space to write, and so on) is detected on specified file descriptors. It is common in applications written for those systems for select() to be used before read() in situations (such as keyboard input) where interruption of I/O due to a signal is de? sired.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an ``unrecoverable error''. It is considered beyond the scope of this volume of POSIX.1?2017 to describe what happens in the case of hardware errors.

Earlier versions of this standard allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the re? sulting confusion, it was decided that POSIX.1?2008 should support only one of these behaviors. Historical practice on AT&T-derived systems was to have read() and write() return -1 and set errno to [EINTR] when in? terrupted after some, but not all, of the data requested had been

transferred. However, the US Department of Commerce FIPS 151?1 and FIPS 151?2 require the historical BSD behavior, in which read() and write() return the number of bytes actually transferred before the interrupt. If -1 is returned when any data is transferred, it is difficult to re? cover from the error on a seekable device and impossible on a non-seek? able device. Most new implementations support this behavior. The be? havior required by POSIX.1?2008 is to return the number of bytes trans? ferred.

POSIX.1?2008 does not specify when an implementation that buffers read()s actually moves the data into the user-supplied buffer, so an implementation may choose to do this at the latest possible moment. Therefore, an interrupt arriving earlier may not cause read() to return a partial byte count, but rather to return -1 and set errno to [EINTR].

Consideration was also given to combining the two previous options, and setting errno to [EINTR] while returning a short count. However, not only is there no existing practice that implements this, it is also contradictory to the idea that when errno is set, the function respon? sible shall return -1.

This volume of POSIX.1?2017 intentionally does not specify any pread() errors related to pipes, FIFOs, and sockets other than [ESPIPE].

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl(), ioctl(), lseek(), open(), pipe(), readv()

The Base Definitions volume of POSIX.1?2017, Chapter 11, General Termi? nal Interface, <stropts.h>, <sys_uio.h>, <unistd.h>

COPYRIGHT

The Open Group Standard, the original IEEE and The Open Group Standard is the referee document. The original Standard can be obtained online at http://www.opengroup.org/unix/online.html .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.ker?nel.org/doc/man-pages/reporting_bugs.html .