**Pipe**

pipe, pipe2 - create pipe

**Syntax**

#include <unistd.h>

int pipe(int pipefd[2]);

#define \_GNU\_SOURCE /\* See feature\_test\_macros(7) \*/

#include <fcntl.h> /\* Obtain O\_\* constant definitions \*/

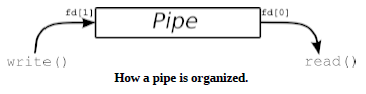
#include <unistd.h>

int pipe2(int pipefd[2], int flags);

**Return**

On success, zero

On error, -1, and errno is set appropriately.



pipe() creates a pipe, a unidirectional data channel that can be used for IPC.

The array pipefd is used to return two file descriptors referring to the ends of the pipe. A call to the pipe() function returns a pair of file descriptors.

pipefd[0] read end

pipefd[1] write end

Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

On many systems, pipes will fill up after you write about 10K to them without reading anything out.

# Flags

Following values can be bitwise ORed in flags to obtain different behavior:

**O\_CLOEXEC**

Set the close-on-exec (FD\_CLOEXEC) flag on the two new file descriptors.

**O\_DIRECT** (since Linux 3.4)

Create a pipe that performs I/O in "packet" mode. Each write(2) to the pipe is dealt with as a separate packet, and read(2)s from the pipe will read one packet at a time.

1. Writes of greater than PIPE\_BUF bytes (see pipe(7)) will be split into multiple packets. The constant PIPE\_BUF is defined in <limits.h>.
2. If a read(2) specifies a buffer size that is smaller than the next packet, then the requested number of bytes are read, and the excess bytes in the packet are discarded. Specifying a buffer size of PIPE\_BUF will be sufficient to read the largest possible packets (see the previous point).
3. Zero-length packets are not supported.

**O\_NONBLOCK**

Set the O\_NONBLOCK file status flag on the two new open file descriptions.

# Example

fork() and pipe()

Implement "ls | wc -l" in C

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(void) {

int pfds[2];

pipe(pfds);

if (!fork()) {

close(1); /\* close normal stdout \*/

dup(pfds[1]); /\* make stdout same as pfds[1] \*/

close(pfds[0]); /\* we don't need this \*/

execlp("ls", "ls", NULL);

}

else {

close(0); /\* close normal stdin \*/

dup(pfds[0]); /\* make stdin same as pfds[0] \*/

close(pfds[1]); /\* we don't need this \*/

execlp("wc", "wc", "-l", NULL);

}

return 0;

}

close(1) frees up file descriptor 1 (standard output). dup(pfds[1]) makes a copy of the write-end of the pipe in the first available file descriptor, which is “1”, since we just closed that. In this way, anything that ls writes to standard output (file descriptor 1) will instead go to pfds[1] (the write end of the pipe).

The wc section of code works the same way, except in reverse.

# Limitation

They have no name, and can therefore be used only by related processes.

Solution: FIFOs (named pipes)

Note: Technically, pipes can be used between unrelated processes, given the ability to pass descriptors between processes.

# popen and pclose Functions

As another example of pipes, the standard I/O library provides the popen function that creates a pipe and initiates another process that either reads from the pipe or writes to the pipe.

#include <stdio.h>

FILE \*popen (const char \*command, const char \*type) ;

Returns: file pointer if OK, NULL on error

int pclose (FILE \*stream);

Returns: termination status of shell or -1 on error

command: shell command line

It is processed by the sh program (normally a Bourne shell), so the PATH environment variable is used to locate the command.

A pipe is created between the calling process and the specified command. The value returned by popen is a standard I/O FILE pointer that is used for either input or output, depending on the character string type.

* If type is r, the calling process reads the standard output of the command.
* If type is w, the calling process writes to the standard input of the command.

The pclose function closes a standard I/O stream that was created by popen, waits for the command to terminate, and then returns the termination status of the shell.

# Example

int main(int argc, char \*\*argv) {

size-t n;

char buff [MAXLINE] , command [MAXLINE] ;

FILE \*fp;

/\* read pathname \*/

fgets (buff, MAXLINE, stdin) ;

n = strlen(buff); /\* fgets() guarantees null byte at end \*/

if (buff[n - 11 == '\nl)

n-- ; /\* delete newline from fgets() \*/

snprintf(command, sizeof(command), "cat %s", buff);

fp = popen(command, "r");

/\* copy from pipe to standard output \*/

while (fgets(buff, MAXLINE, fp) != NULL)

fputs (buff , stdout);

pclose (fp) ;

exit (0) ;

}

# END