15 Pipes and FIFOs

A *pipe* is a mechanism for interprocess communication; data written to the pipe by one process can be read by another process. The data is handled in a first-in, first-out (FIFO) order. The pipe has no name; it is created for one use and both ends must be inherited from the single process which created the pipe.

A *FIFO special file* is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.

A pipe or FIFO has to be open at both ends simultaneously. If you read from a pipe or FIFO file that doesn't have any processes writing to it (perhaps because they have all closed the file, or exited), the read returns end-of-file. Writing to a pipe or FIFO that doesn't have a reading process is treated as an error condition; it generates a `SIGPIPE` signal, and fails with error code `EPIPE` if the signal is handled or blocked.

Neither pipes nor FIFO special files allow file positioning. Both reading and writing operations happen sequentially; reading from the beginning of the file and writing at the end.

15.1 Creating a Pipe

The primitive for creating a pipe is the `pipe` function. This creates both the reading and writing ends of the pipe. It is not very useful for a single process to use a pipe to talk to itself. In typical use, a process creates a pipe just before it forks one or more child processes (see Section 26.4 [Creating a Process], page 749). The pipe is then used for communication either between the parent or child processes, or between two sibling processes.

The `pipe` function is declared in the header file `unistd.h`.

`int pipe` (*int* ***filedes***`[2]`) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe fd | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `pipe` function creates a pipe and puts the file descriptors for the reading and writing ends of the pipe (respectively) into ***filedes***`[0]` and ***filedes***`[1]`.

An easy way to remember that the input end comes first is that file descriptor `0` is standard input, and file descriptor `1` is standard output.

If successful, `pipe` returns a value of `0`. On failure, `-1` is returned. The following `errno` error conditions are defined for this function:

`EMFILE` The process has too many files open.

`ENFILE` There are too many open files in the entire system. See Section 2.2 [Error Codes], page 23, for more information about `ENFILE`. This error never occurs on GNU/Hurd systems.

Here is an example of a simple program that creates a pipe. This program uses the `fork` function (see Section 26.4 [Creating a Process], page 749) to create a child process. The parent process writes data to the pipe, which is read by the child process.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

/* Read characters from the pipe and echo them to stdout. */

void
read_from_pipe (int file)
{
  FILE *stream;
  int c;
  stream = fdopen (file, "r");
  while ((c = fgetc (stream)) != EOF)
    putchar (c);
  fclose (stream);
}

/* Write some random text to the pipe. */

void
write_to_pipe (int file)
{
  FILE *stream;
  stream = fdopen (file, "w");
  fprintf (stream, "hello, world!\n");
  fprintf (stream, "goodbye, world!\n");
  fclose (stream);
}

int
main (void)
{
  pid_t pid;
  int mypipe[2];

  /* Create the pipe. */
  if (pipe (mypipe))
    {
      fprintf (stderr, "Pipe failed.\n");
      return EXIT_FAILURE;
    }

  /* Create the child process. */
  pid = fork ();
  if (pid == (pid_t) 0)
    {
      /* This is the child process.
      Close other end first. */
      close (mypipe[1]);
      read_from_pipe (mypipe[0]);
      return EXIT_SUCCESS;
    }
  else if (pid < (pid_t) 0)
    {
      /* The fork failed. */
      fprintf (stderr, "Fork failed.\n");
```

```
          return EXIT_FAILURE;
        }
      else
        {
          /* This is the parent process.
          Close other end first. */
          close (mypipe[0]);
          write_to_pipe (mypipe[1]);
          return EXIT_SUCCESS;
        }
    }
```

15.2 Pipe to a Subprocess

A common use of pipes is to send data to or receive data from a program being run as a subprocess. One way of doing this is by using a combination of `pipe` (to create the pipe), `fork` (to create the subprocess), `dup2` (to force the subprocess to use the pipe as its standard input or output channel), and `exec` (to execute the new program). Or, you can use `popen` and `pclose`.

The advantage of using `popen` and `pclose` is that the interface is much simpler and easier to use. But it doesn't offer as much flexibility as using the low-level functions directly.

`FILE * popen` (*const char *command, const char *mode*) [Function]

Preliminary: | MT-Safe | AS-Unsafe heap corrupt | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `popen` function is closely related to the `system` function; see Section 26.1 [Running a Command], page 747. It executes the shell command *command* as a subprocess. However, instead of waiting for the command to complete, it creates a pipe to the subprocess and returns a stream that corresponds to that pipe.

If you specify a *mode* argument of `"r"`, you can read from the stream to retrieve data from the standard output channel of the subprocess. The subprocess inherits its standard input channel from the parent process.

Similarly, if you specify a *mode* argument of `"w"`, you can write to the stream to send data to the standard input channel of the subprocess. The subprocess inherits its standard output channel from the parent process.

In the event of an error `popen` returns a null pointer. This might happen if the pipe or stream cannot be created, if the subprocess cannot be forked, or if the program cannot be executed.

`int pclose` (*FILE *stream*) [Function]

Preliminary: | MT-Safe | AS-Unsafe heap plugin corrupt lock | AC-Unsafe corrupt lock fd mem | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `pclose` function is used to close a stream created by `popen`. It waits for the child process to terminate and returns its status value, as for the `system` function.

Here is an example showing how to use `popen` and `pclose` to filter output through another program, in this case the paging program `more`.

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

```
#include <stdlib.h>

void
write_data (FILE * stream)
{
  int i;
  for (i = 0; i < 100; i++)
    fprintf (stream, "%d\n", i);
  if (ferror (stream))
    {
      fprintf (stderr, "Output to stream failed.\n");
      exit (EXIT_FAILURE);
    }
}

int
main (void)
{
  FILE *output;

  output = popen ("more", "w");
  if (!output)
    {
      fprintf (stderr,
               "incorrect parameters or too many files.\n");
      return EXIT_FAILURE;
    }
  write_data (output);
  if (pclose (output) != 0)
    {
      fprintf (stderr,
               "Could not run more or other error.\n");
    }
  return EXIT_SUCCESS;
}
```

15.3 FIFO Special Files

A FIFO special file is similar to a pipe, except that it is created in a different way. Instead of being an anonymous communications channel, a FIFO special file is entered into the file system by calling `mkfifo`.

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it. Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

The `mkfifo` function is declared in the header file `sys/stat.h`.

`int mkfifo` (*const char *`filename`, mode_t `mode`*) [Function]

Preliminary: | MT-Safe | AS-Safe | AC-Safe | See Section 1.2.2.1 [POSIX Safety Concepts], page 2.

The `mkfifo` function makes a FIFO special file with name *filename*. The *mode* argument is used to set the file's permissions; see Section 14.9.7 [Assigning File Permissions], page 409.

The normal, successful return value from `mkfifo` is `0`. In the case of an error, `-1` is returned. In addition to the usual file name errors (see Section 11.2.3 [File Name Errors], page 245), the following `errno` error conditions are defined for this function:

`EEXIST` The named file already exists.

`ENOSPC` The directory or file system cannot be extended.

`EROFS` The directory that would contain the file resides on a read-only file system.

15.4 Atomicity of Pipe I/O

Reading or writing pipe data is *atomic* if the size of data written is not greater than `PIPE_BUF`. This means that the data transfer seems to be an instantaneous unit, in that nothing else in the system can observe a state in which it is partially complete. Atomic I/O may not begin right away (it may need to wait for buffer space or for data), but once it does begin it finishes immediately.

Reading or writing a larger amount of data may not be atomic; for example, output data from other processes sharing the descriptor may be interspersed. Also, once `PIPE_BUF` characters have been written, further writes will block until some characters are read.

See Section 32.6 [Limits on File System Capacity], page 848, for information about the `PIPE_BUF` parameter.