STREAMS-Based Pipes and FIFOS 11

Overview of Pipes and FIFOs

A pipe in the SunOS 5.4 system is a mechanism that provides a communication path between multiple processes. Prior to SunOS 5.0, SunOS had *standard* pipes and named pipes (also called FIFOs, or First-In-First-Out). With standard pipes, one end was opened for reading and the other end for writing, thus data flow was unidirectional. FIFOs had only one end and typically one process opened the file for reading and another process opened the file for writing. Data written into the FIFO by the writer could then be read by the reader.

To provide greater support and development flexibility for networked applications, pipes and FIFOs have become STREAMS-based in SunOS 5.4. The basic interface remains the same but the underlying implementation has changed. Pipes now provide a bidirectional mechanism for process communication. When a pipe is created via the `pipe`(2) system call, two Streams are opened and connected side-by-side, thus providing a full-duplex mechanism. Data flow is on First-In-First-Out (FIFO) basis. Previously, pipes were associated with character devices and the creation of a pipe was limited to the capacity and configuration of the device. STREAMS-based pipes and FIFOs are not attached to STREAMS-based character devices. This eliminates configuration constraints and limits the number of opened pipes to the number of file descriptors allowed for each process.

The remainder of this chapter uses the terms *pipe* and *STREAMS-based pipe* interchangeably for a STREAMS-based pipe.

11

Creating and Opening Pipes and FIFOs

FIFOs are created via `mknod`(2) or `mkfifo`(3C). FIFOs behave like regular file system nodes but are distinguished from other file system nodes by the `p` in the first column when the `ls -l` command is executed. Data written to the FIFO or read from the FIFO flow up and down the Stream in STREAMS buffers. Data written by one process can be read by another process.

FIFOs are opened in the same manner as other file system nodes via the `open`(2) system call. Any data written to the FIFO can be read from the same file descriptor in the First-In-First-Out manner (serial, sequentially). Modules can also be pushed on the FIFO. See `open`(2) for the restrictions that apply when opening a FIFO.

A STREAMS-based pipe is created via the `pipe`(2) system call that returns two file descriptors, *fd[0]* and *fd[1]*. Both file descriptors are opened for reading and writing. Data written to *fd[0]* becomes data read from *fd[1]* and vice versa.

Each end of the pipe has knowledge of the other end through internal data structures. Subsequent reads, writes, and closes are aware of if the other end of the pipe is open or closed. When one end of the pipe is closed, the internal data structures provide a way to access the Stream for the other end so that an `M_HANGUP` message can be sent to its Stream head.

After successful creation of a STREAMS-based pipe, 0 is returned. If `pipe`(2) is unable to create and open a STREAMS-based pipe, it will fail with *errno* set as follows:

- `EINTR` - Signal was caught while creating the Stream heads.
- `EMFILE` - Could not allocate more file descriptors for the process.
- `ENFILE` - File table has overflowed.
- `ENOMEM` - Could not allocate two `vnodes`.
- `ENOSR` - Could not allocate resources for both Stream heads.

STREAMS modules can be added to a STREAMS-based pipe with the `ioctl`(2) `I_PUSH`. A module can be pushed onto one or both ends of the pipe (see Figure 11-1). However, a pipe maintains the concept of a midpoint so that if a module is pushed onto one end of the pipe, that module cannot be popped from the other end.

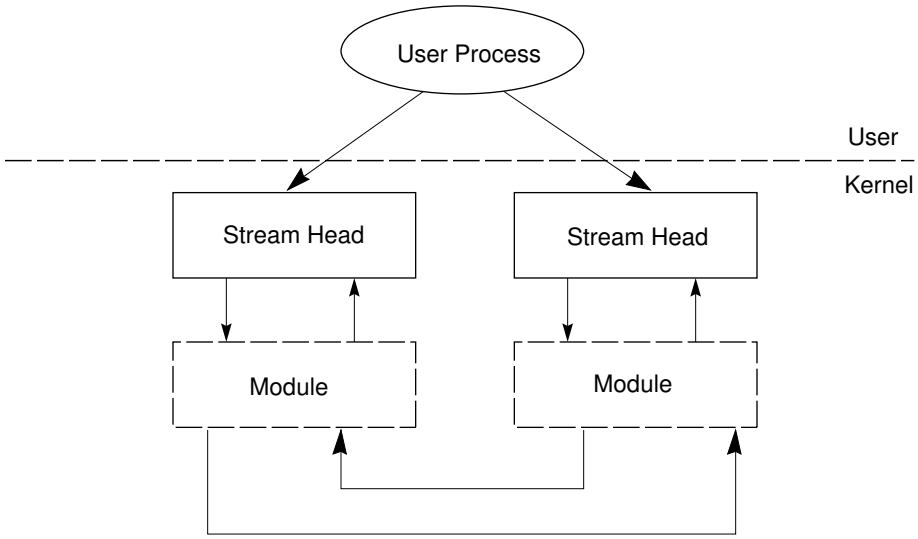

Figure 11-1 Pushing Modules on a STREAMS-based Pipe

Accessing Pipes and FIFOs

STREAMS-based pipes and FIFOs can be accessed through the operating system routines `read`(2), `write`(2), `ioctl`(2), `close`(2), `putmsg`(2), `getmsg`(2), and `poll`(2). In the case of FIFOs, `open`(2) is also used.

Reading from a Pipe or FIFO

The `read`(2) (or `getmsg`(2)) system call is used to read from a pipe or FIFO. Data can be read from either end of a pipe.

On success, the `read` returns the number of bytes read and placed in the buffer. When the end of the data is reached, the `read` returns 0.

When a user process attempts to read from an empty pipe (or FIFO), the following will happen:

- If one end of the pipe is closed, 0 is returned indicating the end of the file.
- If no process has the FIFO open for writing, `read`(2) returns 0 to indicate the end of the file.
- If some process has the FIFO open for writing, or both ends of the pipe are open, and `O_NDELAY` is set, `read`(2) returns 0.
- If some process has the FIFO open for writing, or both ends of the pipe are open, and `O_NONBLOCK` is set, `read`(2) returns -1 and set *errno* to `EAGAIN`.
- If `O_NDELAY` and `O_NONBLOCK` are not set, the `read` call will block until data is written to the pipe, until one end of the pipe is closed, or the FIFO is no longer open for writing.

Writing to a Pipe or FIFO

When a user process calls the `write`(2) system call, data is sent down the associated Stream. If the pipe or FIFO is empty (no modules pushed), data written is placed on the read queue of the other Stream for STREAMS-based pipes, and on the read queue of the same Stream for FIFOs. Since the size of a pipe is the number of unread data bytes, the written data is reflected in the size of the other end of the pipe.

Zero-Length Writes

If a user process issues `write`(2) with 0 as the number of bytes to send a STREAMS-based pipe or FIFO, 0 is returned, and by default no message is sent down the Stream. However, if a user must send a zero-length message downstream, an `ioctl` call may be used to change this default behavior. The flag `SNDZERO` supports this. If `SNDZERO` is set in the Stream head, `write`(2) requests of 0 bytes will generate a zero-length message and send the message down the Stream. If `SNDZERO` is not set, no message is generated and 0 is returned to the user.

The SNDZERO bit may be manipulated via the ioctl I_SWROPT. If the arg in the ioctl call has SNDZERO set, the bit is turned on. If the arg is set to 0, the SNDZERO bit is turned off.

The `ioctl I_GWROPT` is used to return the current write settings.

Atomic Writes

If multiple processes simultaneously write to the same pipe, data from one process can be interleaved with data from another process, if modules are pushed on the pipe or the write is greater than `PIPE_BUF`. The order of data written is not necessarily the order of data read. To ensure that writes of less than `PIPE_BUF` bytes will not be interleaved with data written from other processes, any modules pushed on the pipe should have a maximum packet size of at least `PIPE_BUF`.

Note – `PIPE_BUF` is an implementation-specific constant that specifies the maximum number of bytes that are atomic when writing to a pipe. When writing to a pipe, write requests of `PIPE_BUF` or less bytes will not be interleaved with data from other processes doing writes on the same pipe. However, write requests greater than `PIPE_BUF` bytes may have data interleaved on arbitrary byte boundaries with writes by other processes whether or not the `O_NONBLOCK` or `O_NDELAY` flag is set.

If the module packet size is at least the size of `PIPE_BUF`, the Stream head packages the data in such a way that the first message is at least `PIPE_BUF` bytes. The remaining data may be packaged into smaller or larger blocks depending on buffer availability. If the first module on the Stream cannot support a packet of `PIPE_BUF`, atomic writes on the pipe cannot be guaranteed.

Closing a Pipe or FIFO

The `close`(2) system call closes a pipe or FIFO and dismantles its associated Streams. On the last close of one end of a pipe, an `M_HANGUP` message is sent upstream to the other end of the pipe. Subsequent `read`(2) or `getmsg`(2) calls on that Stream head will return the number of bytes read and zero when there are no more data. Subsequent `write`(2) or `putmsg`(2) requests will fail with *errno* set to `EPIPE`. If the other end of the pipe is mounted, the last close of the pipe will force it to be unmounted.

11

Flushing Pipes and FIFOs

When the flush request is initiated from a user `ioctl` or from a `flushq`() routine, the `FLUSHR` and/or the FLUSHW bits of an `M_FLUSH` message must be switched. The point of switching the bits is the point where the `M_FLUSH` message is passed from a write queue to a read queue. This point is also known as the mid-point of the pipe.

The mid-point of a pipe is not always easily detectable, especially if there are numerous modules pushed on either end of the pipe. In that case, there needs to be a mechanism to intercept all messages passing through the Stream. If the message is an `M_FLUSH` message and it is at the Streams mid-point, the flush bits need to switched.

This bit switching is handled by the `pipemod` module. `pipemod` should be pushed onto a pipe or FIFO where flushing of any kind will take place. The `pipemod` module can be pushed on either end of the pipe. The only requirement is that it is pushed onto an end that previously did not have modules on it. That is, `pipemod` must be the first module pushed onto a pipe so that it is at the mid-point of the pipe itself.

The `pipemod` module handles only `M_FLUSH` messages. All other messages are passed to the next module via the `putnext`() utility routine. If an `M_FLUSH` message is passed to `pipemod` and the `FLUSHR` and `FLUSHW` bits are set, the message is not processed but is passed to the next module via the `putnext`() routine. If only the `FLUSHR` bit is set, the `FLUSHR` bit is turned off and the `FLUSHW` bit is set. The message is then passed to the next module via `putnext`. Similarly, if the `FLUSHW` bit was the only bit set in the `M_FLUSH` message, the `FLUSHW` bit is turned off and the `FLUSHR` bit is turned on. The message is then passed to the next module on the Stream.

The `pipemod` module can be pushed on any Stream if it requires the bit switching.

Named Streams

It may be necessary for some applications to associate a Stream or STREAMS-based pipe with an existing node in the file system name space. For example, a server process may create a pipe, name one end of the pipe, and allow unrelated processes to communicate with it over that named end.

fattach

A STREAMS file descriptor can be named by attaching that file descriptor to a node in the file system name space. The routine `fattach`() (see also `fattach`(3C)) is used to name a STREAMS file descriptor. Its format is:

```
int fattach (int fildes, char *path)
```

where *fildes* is an open file descriptor that refers to either a STREAMS-based pipe or a STREAMS device driver (or a pseudo device driver), and *path* is an existing node in the file system name space (for example, a regular file, directory, character special file, etc).

The *path* cannot have a Stream already attached to it. It cannot be a mount point for a file system nor the root of a file system. A user must be an owner of the *path* with write permission or a user with the appropriate privileges in order to attach the file descriptor.

If the *path* is in use when the routine `fattach`() is executed, those processes accessing the *path* will not be interrupted and any data associated with the *path* before the call to the `fattach`() routine will continue to be accessible by those processes.

After a Stream is named, all subsequent operations (for example, `open`(2)) on the *path* will operate on the named Stream. Thus, it is possible that a user process has one file descriptor pointing to the data originally associated with the *path* and another file descriptor pointing to a named Stream.

Once the Stream has been named, the `stat`(2) system call on *path* will show information for the Stream. If the named Stream is a pipe, the `stat`(2) information will show that *path* is a pipe. If the Stream is a device driver or a pseudo device driver, *path* appears as a device. The initial modes, permissions, and ownership of the named Stream are taken from the attributes of the *path*. The user can issue the system calls `chmod`(2) and `chown`(2) to alter the attributes of the named Stream and not affect the original attributes of the *path* nor the original attributes of the STREAMS file.

The size represented in the `stat`(2) information will reflect the number of unread bytes of data currently at the Stream head. This size is not necessarily the number of bytes written to the Stream.

A STREAMS-based file descriptor can be attached to many different *path*s at the same time (that is, a Stream can have many names attached to it). The modes, ownership, and permissions of these *path*s may vary, but operations on any of these *path*s will access the same Stream.

Named Streams can have modules pushed on them, be polled, be passed as file descriptors, and be used for any other STREAMS operation.

fdetach

A named Stream can be disassociated from a filename with the `fdetach`() routine (see also `fdetach`(3C)) that has the following format:

```
int fdetach (char *path)
```

where *path* is the name of the previously named Stream. Only the owner of *path* or the user with the appropriate privileges may disassociate the Stream from its name. The Stream may be disassociated from its name while processes are accessing it. If these processes have the named Stream open at the time of the `fdetach`() call, the processes will not get an error, and will continue to access the Stream. However, after the disassociation, subsequent operations on *path* access the underlying file rather than the named Stream.

If only one end of the pipe is named, the last close of the other end will cause the named end to be automatically detached. If the named Stream is a device and not a pipe, the last close will not cause the Stream to be detached.

If there is no named Stream or the user does not have access permissions on *path* or on the named Stream, `fdetach`() returns -1 with *errno* set to `EINVAL`. Otherwise, `fdetach`() returns 0 for success.

A Stream will remain attached with or without an active server process. If a server aborted, the only way a named Stream is cleaned up is if the server executed a `clean up` routine that explicitly detached and closed down the Stream.

If the named Stream is that of a pipe with only one end attached, clean up will occur automatically. The named end of the pipe is forced to be detached when the other end closes down. If there are no other references after the pipe is detached, the Stream is deallocated and cleaned up. Thus, a forced detach of a pipe end will occur when the server is aborted.

If the both ends of the pipe are named, the pipe remains attached even after all processes have exited. In order for the pipe to become detached, a server process would have to explicitly call a program that executed the `fdetach`() routine.

To eliminate the need for the server process to invoke the program, the `fdetach`(1M) command can be used. This command accepts a path name that is a path to a named Stream. When the command is invoked, the Stream is detached from the path. If the name was the only reference to the Stream, the Stream is also deallocated.

A user invoking the `fdetach`(1M) command must be an owner of the named Stream or a user with the appropriate permissions.

isastream

The function `isastream`() (see `isastream`(3C)) may be used to determine if a file descriptor is associated with a STREAM. Its format is:

```
int isastream (int fildes);
```

where *fildes* refers to an open file. `isastream`() returns 1 if *fildes* represents a STREAMS file, and 0 if not. On failure, `isastream`() returns -1 with *errno* set to `EBADF`.

This function is useful for client processes communicating with a server process over a named Stream to check whether the file has been overlaid by a Stream before sending any data over the file.

Passing File Descriptors

Named Streams are useful for passing file descriptors between unrelated processes on the same machine. A user process can send a file descriptor to another process by invoking the `ioctl`(2) `I_SENDFD` on one end of a named Stream. This sends a message containing a file pointer to the Stream head at the other end of the pipe. Another process can retrieve that message containing the file pointer by invoking the `ioctl`(2) `I_RECVFD` on the other end of the pipe.

Named Streams in A Remote Environment

If a user on the server machine creates a pipe and mounts it over a file that is part of an RFS (Remote File System) advertised resource, a user on the client machine (that has remotely named the resource) may access the remote named Stream. A user on the client machine is not allowed to pass file descriptors across the named Stream and will get an error when the `ioctl` request is attempted. If a user on the client machine creates a pipe and attempts to attach it to a file that is a remotely named resource, the system call will fail.

The following three examples are given as illustrations:

- Suppose the server advertised a resource */dev/foo*, created a STREAMS-based pipe, and attached one end of the pipe onto */dev/foo/spipe*. All processes on the server machine will be able to access the pipe when they open */dev/foo/spipe*. Now suppose that client XYZ mounts the advertised resource */dev/foo* onto its */mnt* directory. All processes on client XYZ will be able to access the STREAMS-based pipe when they open */mnt/spipe*.
- If the server advertised another resource */dev/fog* and client XYZ mounts that resource onto its */install* directory and then attaches a STREAMS-based pipe onto */install*, the mount would fail with *errno* set to `EBUSY`, because */install* is already a mount point. If client XYZ attached a pipe onto */install/spipe*, the mount would also fail with *errno* set to `EREMOTE`, because the mount would require crossing an RFS mount point.
- Suppose the server advertised its */usr/control* directory and client XYZ mounts that resource onto its */tmp* directory. The server now creates a STREAMS-based pipe and attaches one end over its */usr* directory. When the server opens */usr* it will access the pipe. On the other hand, when the client opens */tmp* it will access what is in the server's */usr/control* directory.

Unique Connections

With named pipes, client processes may communicate with a server process via a module called `connld` that enables a client process to gain a unique, non-multiplexed connection to a server. The `connld` module can be pushed onto the named end of the pipe. If `connld` is pushed on the named end of the pipe and that end is opened by a client, a new pipe will be created. One file descriptor for the new pipe is passed back to a client (named Stream) as the file

descriptor from the `open`(2) system call and the other file descriptor is passed to the server via `ioctl` I_RECUFD. The server and the client may now communicate through a new pipe.

Figure 11-2 illustrates a server process that has created a pipe and pushed the `connld` module on the other end. The server then invokes the `fattach`() routine to name the other end */usr/toserv*.

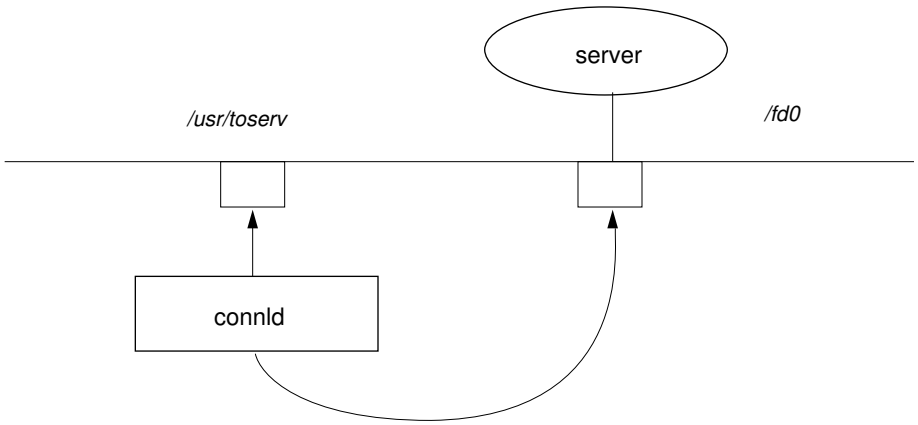

Figure 11-2 Server Sets Up a Pipe

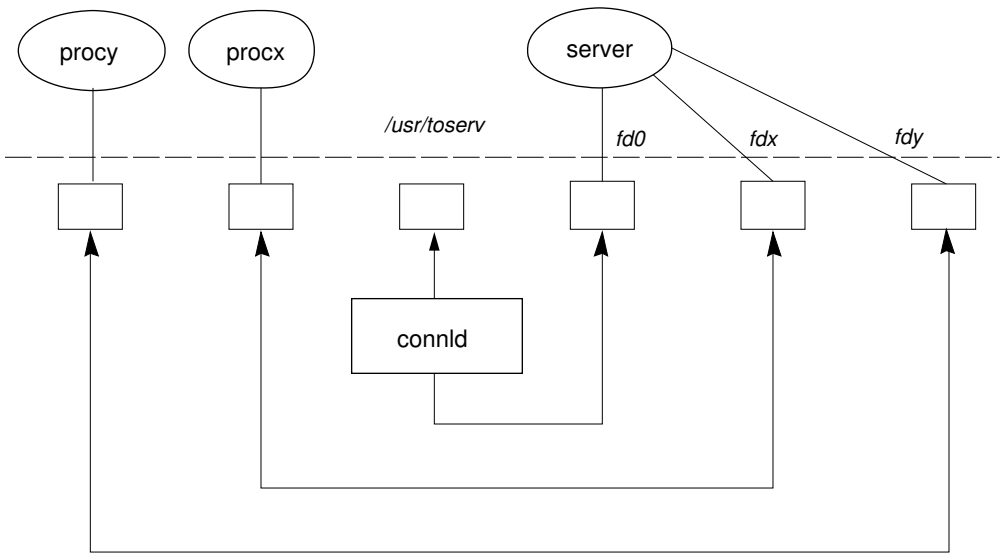

Figure 11-3 Processes X and Y Open */usr/toserv*

When process X (procx) opens */usr/toserv*, it gains a unique connection to the server process that was at one end of the original STREAMS-based pipe. When process Y (procy) does the same, it also gains a unique connection to the server. As shown in Figure 11-3, the server process has access to three separate STREAMS-based pipes via three file descriptors.

`connld` is a STREAMS-based module that has an open, close, and `put` procedure.

When the named Stream is opened, the open routine of `connld` is called. The `connld` open will fail if:

- The pipe ends can not be created.
- A file pointer and file descriptor can not be allocated.
- The Stream head can not stream the two pipe ends.
- `strioctl`() fails while sending the file descriptor to the server.

The open is not complete until the server process has received the file descriptor using the `ioctl I_RECVFD`. The setting of the `O_NDELAY` or `O_NONBLOCK` flag has no impact on the open.

The `connld` module does not process messages. All messages are passed to the next object in the Stream. The `read` and `write put` routines call `putnext`() to send the message up or down the Stream.

11