UNIX® System V Network Programming

Stephen A. Rago

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

UNIX® System V Network Programming

Stephen A. Rago

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts Menlo Park, California New York Don Mills, Ontario
Wokingham, England Amsterdam Bonn Paris Milan Madrid Sydney Tokyo
Seoul Taipei Mexico City San Juan

The programs and applications presented in this book have been included for their instructional value. They have been tested with care, but are not guaranteed for any particular purpose. The publisher does not offer any warranties or representations, nor does it accept any liabilities with respect to the programs or applications.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information please contact:

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company
One Jacob Way
Reading, Massachusetts 01867

Library of Congress Cataloging-in-Publication Data

Rago, Stephen A.
UNIX System V Network Programming/Stephen A. Rago.
p. cm. (Addison-Wesley professional computing series)
Includes index.
ISBN 0-201-56318-5 (hard)
1. Operating systems (Computers). 2. UNIX System V (Computer file). 3. Computer networks. I. Title. II. Series.
QA76.76.063R34 1993
005.7′11—dc20 92-45276
CIP

Copyright © 1993 by Addison-Wesley Publishing Company
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

ISBN 0-201-56318-5
Text printed on recycled and acid-free paper.
1 2 3 4 5 6 7 8 9 10 MU 96959493
First Printing, June 1993

To Patricia

This page intentionally left blank

Contents

Preface

This book is for programmers who are interested in learning how to use the networking interfaces in UNIX System V Release 4 (SVR4). We use real-life examples to demonstrate how interfaces are used and techniques are applied. All too often in the workplace we find ourselves faced with new assignments for which we have little background. In these situations, we must educate ourselves as quickly as possible so that we can competently undertake the task at hand. Although technical manuals usually provide the information necessary to complete a task, they often lack the background, motivation, and explanation that help us to understand more clearly what we're doing and why we're doing it.

Intended as a practical reference, this book contains very little coverage of theory, and details better dealt with through manual pages are omitted, although references are used liberally. It could, however, be used to complement a graduate or advanced undergraduate course in networking.

As a prerequisite to reading this book, you should be familiar with the UNIX environment and the C programming language so that the examples can be understood. Some background in data structures and algorithms would be helpful, but is not required.

References to SVR4 manual pages are in the running text, appearing as the command name or function name, followed by the section of the manual in which the page is found, as in `open(2)`. Here, we are referring to the `open` manual page in Section 2 of the system manuals.

Originally, there was only one manual for the system. With the introduction of each new release of the system, the manual grew in size until it had to be split up into separate manuals. In UNIX System V Release 3, there was one manual for users, one manual for programmers, and one manual for system administrators.

In SVR4, however, the manual pages were redistributed by functional area. The user commands are no longer in a single manual, nor can you find all the programming interfaces in one place. This new organization has proven difficult to navigate by novices and experts alike. The following summary should aid in the process of locating the desired manual pages.

Programmer's Reference Manual
- (1) Commands relating to source code management, compilation, and loading
- (2) System calls
- (3, 3C, 3S, 3E, 3G, 3M, 3X) Most library routines
- (4) File formats
- (5) Miscellany (commonly used constants, data structures, and macros)

Programmer's Guide: Networking Interfaces
- (1, 1M) Networking commands
- (3, 3C, 3N) Network-related library routines
- (4) Network-related file formats
- (5) Miscellany, including network-related environment variables
- (7) Networking drivers and modules

Programmer's Guide: STREAMS
- (1, 1M) STREAMS-related commands
- (2) STREAMS-specific system calls
- (3C) STREAMS-specific library routines
- (7) STREAMS modules and drivers

User's Reference Manual
- (1) Commands any user might want to run

System Administrator's Reference Manual
- (1M) Administrative commands
- (4) Administrative file formats
- (5) Miscellaneous facilities
- (7) Special files (devices)
- (8) Administrative procedures

You might find it helpful if these manuals are close by when you read this book.

Background

The first standard network interface incorporated in the UNIX system was the socket mechanism. This mechanism was provided in the 4.2 release of the Berkeley Software Distribution (BSD) version of the UNIX operating system from the University of California at Berkeley. With it was an implementation of the Internet protocol suite (TCP, UDP, IP, et al.). These became available in 1983.

AT&T did not address standard networking interfaces in System V until 1985, when it ported Dennis Ritchie's Streams mechanism from the Version 8 Research UNIX System to UNIX System V Release 2.0p, the unreleased predecessor to System V Release 3.0 (SVR3). With the release of SVR3 in 1986, STREAMS, the framework for networking in System V, became generally available, along with the Transport Layer Interface (TLI) library. Ironically, SVR3 was released without including any networking protocols.

In 1988, X/OPEN, a consortium dedicated to enhancing application portability through standards endorsements, specified its own transport layer interface library, based on AT&T's TLI library. The X/OPEN specification, called the X/OPEN Transport Interface (XTI), is effectively a superset of TLI. In 1990 the Portable

Operating System Interface (POSIX) committee of the Institute of Electrical and Electronics Engineers (IEEE) created the 1003.12 working group to standardize portable networking interfaces for application programs. As of this writing, the 1003.12 working group's efforts are still underway, but it looks as though both sockets and XTI will be included in the standard.

SVR4 is unique in that it includes support for many standards in one operating system. Unlike other versions of UNIX that support dual-universe environments, SVR4 provides applications with one environment consisting of features from previous versions of the System V, SunOS, BSD, Xenix, SCO, and Research UNIX systems, as well as some new features of its own. Support for POSIX 1003.1 (the system application programming interface) is also provided. The major networking interfaces provided include STREAMS, TLI, sockets, and remote procedure calls.

Organization

The material covered in this book pertains mainly to SVR4, although some features were present in earlier releases of UNIX System V. This book is divided into four sections: background material, user-level network programming, kernel-level network programming, and a design example.

Both user-level and kernel-level networking components are described to present a complete picture of network programming in UNIX System V. Although not everyone will be interested in both environments, knowledge of one environment makes programming in the other easier. Instead of just blindly following the instructions in the manuals, it enables the programmer to understand the effects of his or her actions and make better design decisions.

The first two chapters provide some background that will make the rest of the book more useful to readers with less experience. More experienced readers can skip these introductory chapters without much loss of context. Chapter 1 provides a brief introduction to networking concepts, and Chapter 2 provides an overview of application programming in the UNIX System V environment. In particular, Chapter 2 contains example functions that are used throughout the rest of this text. If you skip Chapter 2, you might want to refer back to individual examples as you come across these functions in later chapters.

Chapter 3 is the first chapter concerned with network programming per se. It covers the STREAMS programming environment. Since the STREAMS mechanism is the basis for most of the communication facilities in System V, understanding its services and system call interface is a prerequisite to discussing any System V networking facility.

Chapter 4 covers the Transport Layer Interface library. This is the interface applications use to access the services provided by the transport layer of a computer network. Emphasis is placed on application design to support network independence.

Chapter 5 describes the network selection and name-to-address translation facilities, which further extend the ability of a programmer to design network-independent applications. Chapter 6 covers the network listener process. Using the listener simplifies the design of server processes. The Service Access Facility (SAF), the administrative framework in which the listener operates, is also discussed.

Chapter 7 gives a brief description of the BSD socket interface and its corresponding implementation in SVR4. The socket and TLI mechanisms are contrasted and compared. Chapter 8 discusses remote procedure calls and the external data representation used to develop distributed applications. This ends the user-level section of the text.

The next four chapters are dedicated to kernel-level network programming. Chapter 9 describes the kernel environment, its utility routines, and the interfaces to the STREAMS environment. Chapter 10 describes how to write STREAMS drivers, centering around the design of a simple Ethernet driver. Chapter 11 describes how to write STREAMS modules, centering around the design of a module that can be used to emulate a terminal over a network connection. Chapter 12 describes how to write STREAMS multiplexing drivers. It uses a simple connection-oriented transport provider as a detailed example.

Finally, the last section of the book, Chapter 13, covers the design of a SLIP package for SVR4, including both the user-level and kernel-level components. It illustrates the application of much from the preceding 12 chapters and, in essence, ties the book together.

Much of the interesting material lies in the examples. You are encouraged to work through each until it is understood. Source code for the examples is available via anonymous FTP from the host `ftp.uu.net` in the file `published/books/rago.netprog.tar.Z`. If you don't have direct access to the Internet, you can use `uucp` to copy the source to your machine as follows:

```
uucp uunet!~/published/books/rago.netprog.tar.Z /tmp
```

(This will place a copy of `rago.netprog.tar.Z` in `/tmp` on your system.) If you have any comments, questions, or bug reports, please send electronic mail to `sar@plc.com`.

Acknowledgements

This book was produced on an Intel i386-based system running UNIX System V Release 4.0, Version 3. The text editor `sam` was used to create and update the text. The pictures were created with `xcip`, a newer version of `cip`, on an AT&T 630MTG terminal. The output for the book was produced with `eqn`, `tbl`, `pic`, `troff`, and `dpost` from the Documenter's WorkBench, Version 3.2.

I would like to thank the following reviewers for their invaluable input: Steve Albert (Unix System Laboratories), Maury Bach (IBM Scientific and Technical Center), George Bittner (Programmed Logic Corporation), Steve Buroff (AT&T Bell Labs), Jeff Gitlin (AT&T), Ron Gomes (Morgan Stanley & Company), Peter Honeyman (University of Michigan), Brian Kernighan (AT&T Bell Labs), Dave Olander (Unix System Laboratories), Dennis Ritchie (AT&T Bell Labs), Michael Scheer (Plexus Systems), Douglas Schmidt (University of California, Irvine), Rich Stevens (independent consultant), and Graham Wheeler (Aztec Information Management). In particular, both Brian Kernighan and Rich Stevens read every chapter and freely

shared their knowledge, experience, and formatting macros and shell scripts. They have greatly increased the quality of the book.

Many people helped by answering questions where written history was vague or incomplete. In addition to the reviewers, this group includes Guy Harris (Auspex Systems), Bob Israel (Epoch Systems), Hari Pulijal (Unix System Laboratories), Usha Pulijal (Unix System Laboratories), Glenn Skinner (SunSoft), Ken Thompson (AT&T Bell Labs), and Larry Wehr (AT&T Bell Labs).

Rich Drechsler (AT&T Bell Labs) provided the PostScript program that increased the width of the constant-width font used throughout this book. Both he and Len Rago (AT&T Bell Labs) helped in debugging problems with the laser printer used during the typesetting of this book. Thanks to them both. Thanks to Dick Hamilton (Unix System Laboratories) for making an early copy of SVR4.2 documentation available. Also, thanks to Gus Amegadzie (Programmed Logic Corporation), who helped test the SLIP software presented in Chapter 13. Special thanks to John Wait (Addison-Wesley) for his advice and encouragement during the last two years.

Finally, I want to thank my family, without whom this book wouldn't have been possible. They have supported me and helped to pull up the slack created by the amount of time I devoted to writing this book. My parents instilled in me the work ethic necessary to get it done (as well as provided their baby-sitting services), and my wife worked harder to give me the time to write it.

This page intentionally left blank

3 STREAMS

The STREAMS mechanism in UNIX System V Release 4 provides the framework on which communication services can be built. These services include communication between terminals and a host computer, between processes on the same computer, and between processes on different computers. This chapter will describe what makes up the STREAMS mechanism and how applications can use it to build communication services.

3.1 STREAMS BACKGROUND

The STREAMS subsystem [not to be confused with the streams returned by `fopen(3C)`] was designed to unify disparate and often ad hoc mechanisms that previously existed in the UNIX operating system to support different kinds of character-based I/O. In particular, it was intended to replace the `clist` mechanism that provided support for terminal I/O in previous releases.

In the `clist`-based terminal subsystem, each terminal line could have one processing element, called a *line discipline*, associated with it. The line discipline handled all special character processing. If a user needed some nonstandard processing of the terminal data stream, he or she could change the line discipline, but only one line discipline could be associated with a terminal at a time.

STREAMS provides a variation on this theme: users can add ("push") and remove ("pop") intermediate processing elements, called *modules*, to and from the data stream at will. The modules can be stacked so that more than one can be used in the data stream at a time. This fundamental change allows independent modules that perform simple tasks to be combined in interesting ways to perform more complex tasks, in much the same way as UNIX commands are connected via shell pipelines.

Data transfer in a stream occurs by passing messages between adjacent processing elements. Only pointers to the messages are passed, avoiding the costly overhead of data copying. Messages are typed and have an associated priority, both indicating how they should be processed. Using message-passing to perform I/O creates another fundamental difference between STREAMS and previous character-based subsystems: data transfer in a stream is data-driven rather than demand-driven.

In previous I/O subsystems, when a user wanted to read data from a device, the driver's `read` routine was invoked. Similarly, when a user wanted to write data, the driver's `write` routine was invoked. In STREAMS, drivers usually do not know when users are reading from or writing to the stream. A read will block until data are available, and a write will result in messages being sent to the driver.

The original Streams [*sic*] mechanism was invented by Dennis Ritchie at AT&T Bell Laboratories around 1982 to unify and improve the character I/O subsystem, improve performance, and decrease system size. It was included in Version 8 of the Research UNIX System. Between 1984 and 1985, AT&T's development organization "productized" Streams, adding a new message structure and support for multiplexing, and capitalizing the name. STREAMS was first generally available in UNIX System V Release 3.0 in 1986. Ironically, full terminal support did not appear until System V Release 4.0, four years later.

3.2 STREAMS ARCHITECTURE

A simple *stream* provides a bidirectional data path between a process at user level and a device driver in the kernel (see Figure 3.1). Data written by the user process travel *downstream* toward the driver, and data received by the driver from the hardware travel *upstream* to be retrieved by the user. Even though data travel up and down the stream in messages, drivers and modules can treat the data flow as a byte stream.

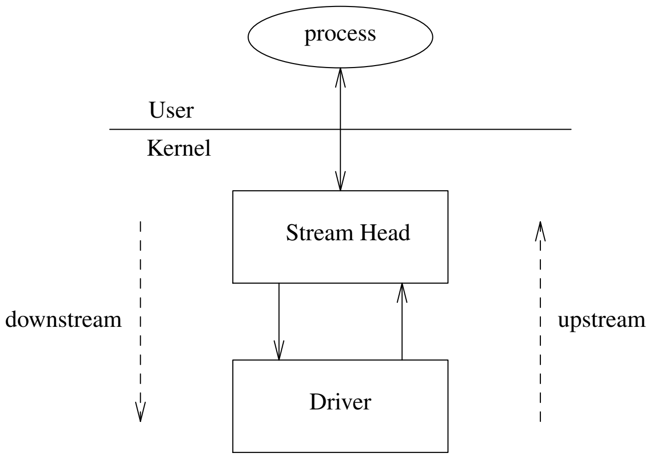

Fig. 3.1. A Simple Stream

A simple stream consists of two processing elements: the *stream head* and a driver. The stream head consists of a set of routines that provide the interface between applications in user space and the rest of the stream in kernel space. When an application makes a system call with a STREAMS file descriptor, the stream head routines are invoked, resulting in data copying, message generation, or control operations being performed. The stream head is the only component in the stream that can copy data between user space and kernel space. All other components effect data transfer solely by passing messages and thus are isolated from direct interaction with users of the stream.

The second processing element is the driver, found at the end, or tail, of the stream. Its job is to control a peripheral device and transfer data between the kernel and the device. Since it interacts with hardware, this kind of driver is called a *hardware driver*. Another kind of driver, called a *software driver*, or *pseudo-driver*, is not associated with any hardware. Instead, it provides a service to applications, such as emulating a terminal-like interface between communicating processes.

The stream head cannot be replaced in the same way that a driver can. Drivers can be added to the kernel simply by linking their object files with the kernel object files. The stream head, on the other hand, is provided with the kernel proper and is fixed. The same stream head processing routines are used with every stream in the system. Each stream head, however, is customizable to a small extent by changing the processing options it supports.

The fundamental building block in a stream is the *queue* (see Figure 3.2). It links one component to the next, thereby forming the stream. Each component in the stream contains at least one pair of queues: one queue for the read side (upstream) and one for the write side (downstream). The queue serves as a location to store messages as they flow up and down the stream, contains status information, and acts as a registry for the routines that will be used to process messages.

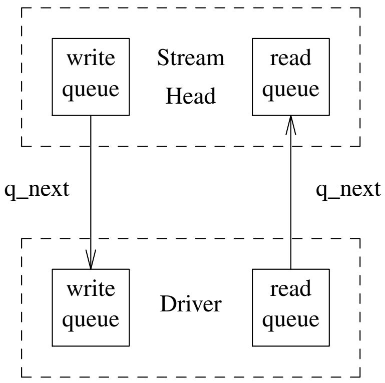

Fig. 3.2. STREAMS Queues

When one component wants to pass a message along in the stream, the queue is used to identify the next component. Then, the next component's queue is used to identify the function to call to pass the message to that component. In this manner, each component's queue provides an interface between the component and the rest of the stream.

A module on a stream is shown in Figure 3.3. A module is an intermediate processing element that can be dynamically added to, or removed from, the stream. Modules are structurally similar to drivers, but usually perform some kind of filter processing on the messages passing between the stream head and the driver. For example, a module might perform data encryption or translation between one interface and another.

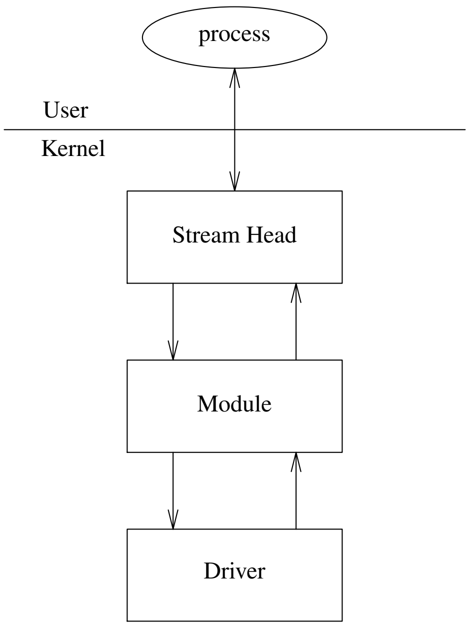

Fig. 3.3. A Module on a Stream

Adding and removing modules are not the only ways a user can customize a stream. A user can also establish and dismantle multiplexing configurations. Multiple streams can be *linked underneath* a special kind of software driver called a *multiplexing driver*, or *multiplexor* (see Figure 3.4). The multiplexing driver will route messages between upper streams opened to access the driver, and lower streams linked underneath the driver. Multiplexing drivers are well suited to implementing windowing systems and networking protocols. Windowing systems multiplex data between multiple windows and the physical terminal. Networking protocols multiplex messages between multiple users and possibly multiple transmission media.

As we have seen, streams can be used to connect processes with devices, but this is not their only use. Streams are also used to connect processes with other

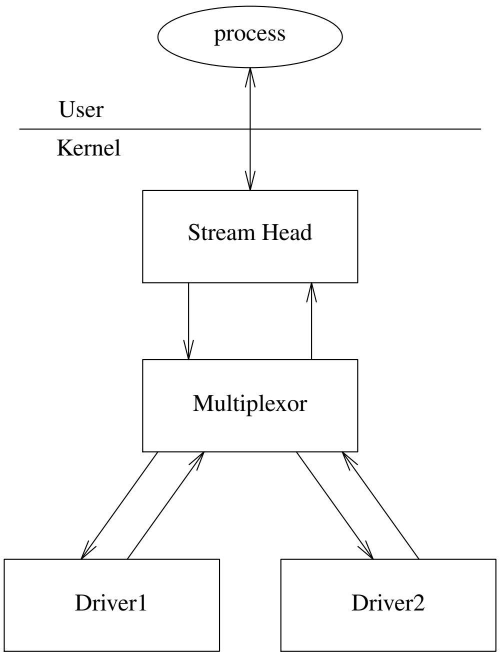

Fig. 3.4. A Multiplexing Driver

processes. Pipes are implemented as streams in UNIX System V Release 4. There are two kinds of pipes: unnamed pipes and named pipes. An unnamed pipe (also called an ‘‘anonymous pipe’’) is so called because it has no entry in the file system namespace. The `pipe` system call creates an unnamed pipe by allocating two stream heads and pointing the write queue of each at the read queue of the other (see Figure 3.5).

Before pipes were implemented using streams, they could only be used for unidirectional data transfer. On successful return, `pipe` would present the user with a file descriptor for one end of the pipe open for reading and a file descriptor for the other end open for writing. In contrast, pipes in SVR4 are full-duplex connections; both pipe ends are open for reading and writing.

A named pipe (also called a ‘‘FIFO’’ because data are retrieved in a first-in–first-out manner) is created via the `mknod` system call. It has a name in the file system and can be accessed with the `open` system call. A named pipe is actually one stream head with its write queue pointing at its read queue (see Figure 3.6). Data written to a named pipe are available for reading from the same ‘‘end’’ of the pipe.

Two processes can use named pipes as rendezvous points, but since communication is unidirectional, their usefulness is limited. They are retained primarily to support applications that still use them. The *mounted streams* facility found in SVR4 makes named pipes obsolete by giving users a way to associate a name with an

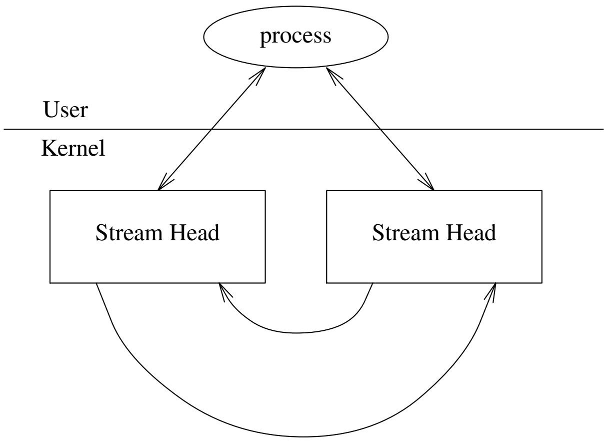

Fig. 3.5. An Anonymous Pipe

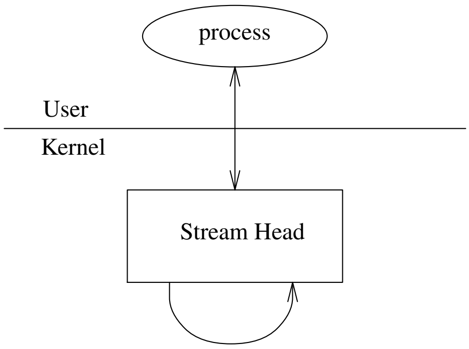

Fig. 3.6. A FIFO

anonymous pipe. Mounted streams are discussed in Section 3.6. Although strictly speaking, an anonymous pipe is also a FIFO, we will follo w current conventions and use the term ‘ ‘FIFO’’ to refer to a named pipe and the term ‘ ‘pipe’’ to refer to an anonymous pipe.

There are se veral advantages to STREAMS-based pipes. First, local inter - process communication (IPC) no w uses the same mechanisms as remote, or net- worked, IPC. This allows applications to treat local IPC connections the same as remote connections. Most operations that can be applied to a stream can no w be applied to a pipe. For example, modules can be pushed onto pipes to obtain more functionality. Second, STREAMS-based pipes are full-duple x, allowing bidirectional communication between two processes with one pipe instead of two.

Now that we have seen the major components that make up a stream, we will briefly look at some of the characteristics of STREAMS messages that are of interest to user-level applications. All communication within a stream occurs by passing pointers to STREAMS messages. The messages are typed, and the type indicates both the purpose of the message and its priority. Based on the type, a message can be either high-priority or normal-priority. The normal-priority messages are further subdivided into *priority bands* for the purposes of flow control and message queueing.

Any data the user wants to transmit to the other end of the stream are packaged in `M_DATA` messages by the stream head. This is the most common message type. If the user needs to send or receive control information, then an `M_PROTO` message is used. Control information is intended for a module or driver in the stream, is interpreted by that component, and is usually not transmitted past the component. A special message type, `M_PCPROTO`, is reserved for high-priority control information, such as interface acknowledgements.

Simple messages are composed of one *message block*. More complex messages can be created by linking multiple message blocks together. These are then treated logically as larger messages. The data in one message block are viewed as being contiguous with the data in the next message block. The message structure is usually transparent to user-level applications. One exception to this is when dealing with a complex message including both control information and user data. In the next section, we shall see how messages like these can be generated and received.

Chapter 9 discusses the STREAMS kernel architecture, including message structure and types, in detail.

3.3 SYSTEM CALLS

Access to a stream is provided via the `open` and `pipe` system calls. In the former case, if a device is not already open, the `open` system call will build the stream. This involves allocating a stream head and allocating two pairs of queues, one pair for the stream head and one pair for the driver. The queues are linked together as shown in Figure 3.2, and the driver's open routine is called. If the driver open succeeds, a file descriptor referring to the stream is returned to the user. If the driver open fails, the structures are freed and the system call fails, returning −1 to the user.

Once the stream is constructed and the first open has completed successfully, another open of the same device will create a new file descriptor referring to the same stream. The driver open routine is called again, but there is no need to allocate the data structures since they are already set up. Pipes, when created, have their streams "opened" internally by the operating system. There is no driver to open.

Multiple processes using the same device also use the same stream. A device is uniquely identified by its *device number*. The device number is split into a *major number* and a *minor number*. The major number identifies the actual device and its associated driver. The minor number identifies a subdevice. For example, a serial ports board would be identified by its major device number, but an individual line on the board is identified by its minor device number.

For a network, minor devices are usually virtual entities since they are multiplexed over one communication line. Instead of being limited by the number of lines, the minor devices are usually limited by tunable configuration parameters that correspond to the maximum number of simultaneous conversations.

Often, applications do not care what particular minor device they use. They just want one that is not already in use. To relieve the applications of the burden of searching for unused minor devices, drivers can be written to support special minors called *clones*. When an application opens the clone minor device of a particular driver, the driver selects a different, unused minor device to be used by the application. Clones are particularly well suited for network drivers and pseudo-drivers.

After a stream is opened, the user may apply to it almost any system call that takes a valid file descriptor. In addition, four system calls work only on file descriptors that refer to streams. These are `getmsg(2)`, `getpmsg(2)`, `putmsg(2)`, and `putpmsg(2)`. They deal with information that is separated into two classes: user data and control information.

The few system calls that will not work with streams are those that make restrictions on the file type, such as `getdents(2)`, which only works with directories, or those that support a conflicting paradigm, like `mmap(2)`. `mmap` and streams do not work together, because mapping a STREAMS driver into the address space of a process would enable direct I/O to the device through loads from and stores into the mapped address range, and the entire stream would be bypassed. No messages would be created, and modules would not get a chance to process the data.

Data can be written to the stream using either `write` or `putmsg`. With `write`, the stream head will copy the user's data into (possibly multiple) STREAMS messages of type `M_DATA` and send them downstream. Data are fragmented according to the *maximum packet size* of the topmost module or driver in the stream. The maximum packet size is a parameter specified by each module and driver that determines the size of the largest STREAMS data message that the component can accept.

If there are too many bytes of data in the messages on the write queues downstream, a stream is said to be *flow-controlled* on the write side. When this condition occurs, `writes` to the stream will block until flow-control restrictions are lifted. If, however, the file descriptor is in nonblocking mode, `write` will return −1 with `errno` set to `EAGAIN` instead of blocking. Flow control protects the system from any one stream using too much memory for messages.

Example 3.3.1. Assume the module on the top of the stream has a maximum packet size of 4096 bytes. Then the line

```
    write(fd, buf, 4097);
```

will send two STREAMS messages downstream. The first message will have 4096 bytes of data in it, and the second message will contain the last byte of data. If, on the other hand, the maximum packet size is larger than 4097 bytes, the `write` will generate only one message.

Actually, a global tunable parameter, STRMSGSZ, can be set by a system administrator to limit the largest STREAMS message created. By default, STRMSGSZ is set to 0 to indicate that the limit is infinite. In this case, write behaves as described. If STRMSGSZ is set to a nonzero value, however, the size of messages created by write is limited by the smaller of the maximum packet size of the module on top of the stream and the value of STRMSGSZ. □

With putmsg, the stream head will try to create exactly one message from the user's buffers. This system call can be used to send control information, data, or both. An M_DATA, M_PROTO, or M_PCPROTO message can be generated, depending on whether the user supplies a control buffer and what flags the user specifies, as summarized in Table 3.1.

Table 3.1. putmsg Argument Combinations

Control Buffer	Data Buffer	Flag	Message Type
No	Yes	0	M_DATA
Yes	Don't care	0	M_PROTO
Yes	Don't care	RS_HIPRI	M_PCPROTO

The synopsis for putmsg is

```
#include <stropts.h>

int putmsg(int fd, const struct strbuf *ctlp,
     const struct strbuf *datp, int flag);
```

fd is a file descriptor referring to a stream, ctlp is a pointer to a structure describing an optional control buffer to be transmitted, and datp is a pointer to an optional data buffer to be transmitted. If a control buffer is provided, flag will determine whether the resulting message is a normal protocol message (M_PROTO) or a high-priority protocol message (M_PCPROTO; "PC" stands for Priority Control). The only valid values for flag are 0 for a normal protocol message and RS_HIPRI for a high-priority protocol message. If no control buffer is provided, then flag must be set to 0, or an error will result.

If a data buffer is provided, then there will be one or more M_DATA blocks linked to the protocol message block. If a data buffer is provided, but no control buffer is provided, then a single M_DATA message block is generated.

To describe the control and data portions of the generated message, the strbuf structure is used, defined in <sys/stropts.h> as:

```
struct strbuf {
     int      maxlen;
     int      len;
     char     *buf;
};
```

maxlen is ignored by putmsg. It is used to specify the size of the user's buffer in calls to getmsg. len indicates the amount of control information or data to be transmitted. buf contains the address of the b uffer containing the control information or data.

On success, putmsg returns 0; on error, it returns −1. If the size of the data is either greater than the maximum packet size or less than the minimum pack et size of the topmost module or driver in the stream, then the system call will fail with errno set to ERANGE.

Example 3.3.2. Assume you ha ve to communicate with a netw ork driver that expects user data to be presented to it with control information describing the identity of the recipient of the data. The recipient is known by its network address. The control information is stored in an M_PROTO message block, and the user data is stored in M_DATA blocks linked to the M_PROTO block. The driver expects the M_PROTO message to contain the following structure:

```
struct data_req {
     long        primitive;       /* identifies message */
     ushort_t    addr_len;        /* destination address */
     ushort_t    addr_offset;     /* location in message */
};

#define DATA_REQUEST 1        /* data request primitive */
```

The data_req structure and the recipient's address are both stored as control information. The address location in the buffer is given by addr_offset. To use the least amount of space, we will choose the address to follo w immediately after the data_req structure.

We can use the following function to request that the driver transmit a message:

```
#include <sys/types.h>
#include <stropts.h>
#include <stdlib.h>
#include <memory.h>

int
senddata(int fd, char *buf, uint_t blen, char *addr,
     ushort_t alen)
{
     struct data_req *reqp;
     struct strbuf ctl, dat;
     char *bp;
     int size, ret;

     /*
      * Allocate a memory buffer large enough to hold
      * the control information.
      */
     size = sizeof(struct data_req) + alen;
     if ((bp = malloc(size)) == NULL)
         return(-1);

     /*
      * Initialize the data_req structure.
```

```
         */
        reqp = (struct data_req *)bp;
        reqp->primitive = DATA_REQUEST;
        reqp->addr_len = alen;
        reqp->addr_offset = sizeof(struct data_req);

        /*
         * Copy the address to the buffer.
         */
        memcpy(bp + reqp->addr_offset, addr, alen);
        ctl.buf = bp;
        ctl.len = size;
        dat.buf = buf;
        dat.len = blen;

        /*
         * Send the message downstream, free the memory
         * allocated for the control buffer, and return.
         */
        ret = putmsg(fd, &ctl, &dat, 0);
        free(bp);
        return(ret);
    }
```

The arguments to `senddata` are a file descriptor referring to the stream, the address of a data buffer, the amount of data in the buffer, the destination address, and the address length. We allocate enough memory for the control buffer to hold the `data_req` structure plus the destination address. We then populate the `data_req` structure with the necessary information and initialize the `strbuf` structures describing the control and data information. After we call `putmsg` to create the message and send it downstream, we free the memory we allocated and return the value returned by `putmsg`. □

Early in the implementation of System V STREAMS, `putmsg` was actually called `send`. Similarly, `getmsg` was called `recv`. Before released, the names were changed to their present ones to avoid conflicting with the 4BSD system calls used for data transfer over sockets. Somehow, the definition of the flag for `getmsg` and `putmsg` was never changed, hence it retains its original name, `RS_HIPRI`. The "R" stands for "receive," and the "S" stands for "send."

Data can be obtained from the stream using either the `read` or `getmsg` system call. `read` treats the data flow as a byte stream and, by default, only operates on `M_DATA` messages. This means the data returned by `read` may span message boundaries. If a `read` is attempted from a stream with an `M_PROTO` or `M_PCPROTO` message at the head of its read queue, the `read` will fail with `errno` set to `EBADMSG`. There are options to change the default behavior of `read`. They will be discussed later in this chapter.

Since `read` is byte-stream-oriented, applications have to do something extra to determine when all the data have been received. Three common methods are

1. Use fixed-size messages. Both the writer and the reader agree in advance on the size of each message passed.
2. Always start each message with a field describing the size of the message.
3. Always end each message with a special character or sequence of characters.

The application determines which method is appropriate.

Example 3.3.3. This example illustrates a function that reads e xactly the amount asked. It can be used to implement method (1) discussed previously.

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

int
mread(int fd, char *buf, int len)
{
     int n;

     while (len > 0) {
         n = read(fd, buf, len);
         if (n <= 0) {
             if (n == 0) /* unexpected EOF */
                 errno = EPROTO;
             return(-1);
         }
         len -= n;
         buf += n;
     }
     return(0);
}
```

Since we have to read exactly `len` bytes, if we receive less than that, we treat it as an error. If `read` returns 0, we treat it as an end-of-f ile condition and return an error. On success, we return 0 instead of the number of bytes read since the caller knows we have read as much as we were asked. □

`getmsg`, like `putmsg`, deals with only one message at a time. It can process both user data and control information, retrie ving an `M_DATA`, `M_PROTO`, or `M_PCPROTO` message from the front of the stream head read queue.

```
#include <stropts.h>

int getmsg(int fd, struct strbuf *ctlp,
     struct strbuf *datp, int *flagp);
```

`fd` is a file descriptor referring to a stream, `ctlp` is a pointer to a structure describing an optional control b uffer to be recei ved, and `datp` is a pointer to an optional data buffer to be recei ved. Information in the `M_PROTO` or `M_PCPROTO` portion of the message is stored in the control buffer described by an `strbuf` structure, shown earlier. The `maxlen` field indicates the size of the b uffer. On return, the `len` field indicates the amount of information received and placed in the buffer. Information in the `M_DATA` portion of the message is processed in a similar manner.

The `flagp` field is a *pointer* to an integer, unlike the `flag` field in `putmsg`. A common mistake is to pass a flag in this field, resulting in `getmsg` failing with `errno` set to `EFAULT` (although the stronger type-checking done by ANSI C compilers has reduced the likelihood of this error).

If the flag pointed to by `flagp` is set to 0, the first message on the stream head read queue will be retrieved. If the flag is set to `RS_HIPRI`, then `getmsg` will wait until an `M_PCPROTO` message arrives at the stream head and will retrieve it instead. On return, the flag will be set to `RS_HIPRI` if an `M_PCPROTO` message has been received, and 0 otherwise.

On success, if an entire message is retrieved, `getmsg` returns 0. If only part of the message is retrieved (because the caller's buffer was too small), then `getmsg` will return nonnegative values. If there is more control information, `MORECTL` is returned. If there are more data, `MOREDATA` is returned. If both remain, then `(MORECTL|MOREDATA)` is returned. On error, −1 is returned.

Example 3.3.4. Assume the same driver used in Example 3.3.2 responds with an acknowledgement every time it receives a request to transmit a message. The acknowledgement does not contain user data, but it does contain control information. It can be implemented as an `M_PCPROTO` message containing the following structure in its data buffer:

```
struct data_ack {
     long    primitive;   /* identifies message */
     long    status;      /* success or failure */
};
#define DATA_ACK 2        /* data request acknowledgement */
```

The `primitive` field identifies the message as a `DATA_ACK`. The `status` field contains an error number if the data request failed, or 0 if it succeeded.

The following routine retrieves the `M_PCPROTO` message from the front of the stream head read queue. It returns 0 if an acknowledgement was received and indicates success. If either the acknowledgement cannot be received or the acknowledgement indicates the request failed, it returns −1.

```
#include <sys/types.h>
#include <stropts.h>
#include <unistd.h>
#include <errno.h>

int
getack(int fd)
{
     struct data_ack ack;
     struct strbuf ctl;
     int fl = RS_HIPRI;
     int ret;

     /*
      * Initialize the control buffer and retrieve the
      * acknowledgement message.
```

```
         */
        ctl.buf = (caddr_t)&ack;
        ctl.maxlen = sizeof(struct data_ack);
        ret = getmsg(fd, &ctl, NULL, &fl);
        if (ret != 0) {
            /*
             * ret shouldn't be greater than 0, but if it
             * is, then the message was improperly formed.
             */
            if (ret > 0)
                errno = EPROTO;
            return(-1);
        }
        if (ack.primitive != DATA_ACK) {
            /*
             * The message we just obtained was not the
             * acknowledgement we expected.
             */
            errno = EPROTO;
            return(-1);
        }

        /*
         * The status field of the message contains an error
         * number if the request failed, or 0 otherwise.
         */
        errno = ack.status;
        return(errno ? -1 : 0);
    }
```

We start out by setting up the control buffer. Using getmsg with the RS_HIPRI flag, we block until an M_PCPROTO message is received. If we get a message with more control information than we asked for, MORECTL will be returned. If the message had data in it (i.e., was linked to an M_DATA message), MOREDATA will be returned since we do not specify a buffer to be used for data. Either of these cases is an error in this example, so we set errno to EPROTO and return failure notification.

If the primitive is not a data acknowledgement, then there has been a protocol error, so we again set errno to EPROTO and return −1. If the message is a DATA_ACK, we set errno to indicate the status of the previous data request and return 0 on success or −1 on failure. □

Why use getmsg and putmsg when read and write will do? The fact is most people probably will not have to use getmsg or putmsg, at least not directly. getmsg and putmsg were implemented to enable user-level applications to communicate with networking drivers and modules that export message-based interfaces. These interfaces (called *service interfaces*) use M_PROTO and M_PCPROTO messages to implement their service primitives and events.

With the read and write system calls, applications would have to work harder to distinguish between service parameters and user data because read and `write` provide a byte-stream interface and only one buffer is involved. This means applications might have to make multiple system calls to send or receive a single message. In addition, `read` and `write` provide only one band of data flow, so high-priority primitives, such as interface acknowledgements and out-of-band data, which ideally would take precedence over other primitives, will be queued behind existing data.

The `getmsg` and `putmsg` system calls solve these problems. They provide a message-oriented interface with separate buffers for control information and user data. For more information on service interfaces, see Section 3.5.

The `ioctl` system call is used to perform I/O control operations on the stream.

```
int ioctl(int fd, int command, ... /* arg */);
```

The particular control operation is identified by `command`. An optional third argument whose type and semantics vary based on the command is usually included. Almost any file-based operation can be implemented as an `ioctl` command. For this reason, `ioctl` has often been described as the "garbage can" system call.

There are two classes of commands that can be used. One class is a command directed at a module or driver in the stream. The other class is directed at the stream head. This latter class is the set of "generic" stream head `ioctl` commands described in `streamio(7)`.

Example 3.3.5. A module can be "pushed" onto the stream with the `I_PUSH` `ioctl` command. Even though it appears to the user as if the module is on the top of the stream, the module is actually inserted between the stream head and the topmost module or driver in the stream. Even so, the conventions are to say, "the module has been pushed on the stream," and "the module is on top of the stream."

```
ioctl(fd, I_PUSH, "module_a");
ioctl(fd, I_PUSH, "module_b");
```

After this sequence of calls, the module named `module_b` is on the top of the stream. After each module is pushed onto the stream, its open routine is called so that it can allocate any necessary data structures. Each push of a module on a stream invokes a different instance of the module, analogous to the way each minor device provides access to a different instance of a driver.

The topmost module on the stream can be popped off with the `I_POP` `ioctl` command:

```
ioctl(fd, I_POP, 0);
```

If this follows the previous two calls, then the module named `module_a` will be left on the top of the stream. When a module is popped off the stream, its close routine is called so that it may deallocate any data structures associated with that instance of the module. □

The stream head `ioctl` commands are summarized in Table 3.2. The class of module and driver `ioctl` commands is further subdivided into two categories: `I_STR` and *transparent*. The `I_STR` type derives its name from the

Table 3.2. Stream Head `ioctl` Commands

Command	Description
`I_NREAD`	Get the number of messages and the size of the f irst message on the stream head read queue.
`I_PUSH`	Push a module on a stream.
`I_POP`	Remove the top module from a stream.
`I_LOOK`	Get the name of the top module on a stream.
`I_FLUSH`	Flush (discard) data on queues.
`I_SRDOPT`	Set read options.
`I_GRDOPT`	Get read options.
`I_STR`	Driver/module `ioctl` commands.
`I_SETSIG`	Enable `SIGPOLL` generation.
`I_GETSIG`	Get events that generate `SIGPOLL`.
`I_FIND`	Verify if a module is in a stream.
`I_LINK`	Create a multiplexor link.
`I_UNLINK`	Remove a multiplexor link.
`I_PEEK`	Peek at data in the first message on the stream head read queue.
`I_FDINSERT`	Send information about another stream.
`I_SENDFD`	Pass a file descriptor.
`I_RECVFD`	Receive a file descriptor.
`I_SWROPT`	Set write options.
`I_GWROPT`	Get write options.
`I_LIST`	List the modules and driver in a stream.
`I_PLINK`	Create a persistent multiplexor link.
`I_PUNLINK`	Remove a persistent multiplexor link.
`I_FLUSHBAND`	Flush banded data on queues.
`I_CKBAND`	Check if a message with the gi ven band is on the stream head read queue.
`I_GETBAND`	Get the band of the first message on the stream head read queue.
`I_ATMARK`	Check if the f irst message on the stream head read queue is "marked."
`I_SETCLTIME`	Set the close delay time.
`I_GETCLTIME`	Get the close delay time.
`I_CANPUT`	Check if the given band is writable.

command used to implement it. The caller packages the real `ioctl` command and argument in an `strioctl` structure and passes the address of the structure as the third argument to `ioctl`, as in:

```
    struct strioctl {           /* defined in <sys/stropts.h> */
         int     ic_cmd;        /* command */
         int     ic_timout;     /* timeout value */
         int     ic_len;        /* length of data */
         char    *ic_dp;        /* pointer to data */
    };
    struct strioctl str;
    :
    .
    ioctl(fd, I_STR, &str);
```

The `strioctl` structure allows the user to specify one optional buffer to contain data to be sent along with the command. On success, data may be returned to the buffer.

The stream head translates the `strioctl` structure into a message sent downstream. If the command is recognized by a module or driver, the request is performed and an acknowledgement message is sent upstream to complete the system call. If the command is unrecognized by all processing elements in the stream, the driver responds by sending a negative acknowledgement message upstream, which causes the system call to fail.

This mechanism did not allow existing binary applications to use `ioctl` with STREAMS-based drivers or modules since the command and data had to be massaged into the `strioctl` structure. Nor did it support the use of more than one data buffer during `ioctl` processing. To solve these problems, transparent `ioctls` were added to SVR3.2.

With transparent `ioctls`, the stream head does not expect an `I_STR` command, nor does it know anything about the format of the data referenced by the third argument to `ioctl`. All unrecognized commands are treated as transparent by sending a specially tagged `ioctl` message downstream. If a module or driver recognizes the command, it will respond with the proper messages to complete the request. Otherwise, the driver will generate a negative acknowledgement, as with the `I_STR` type.

Note that users can specify a timeout with an `I_STR` `ioctl`. The `ic_timout` field contains the number of seconds to wait for the `ioctl` to complete before giving up. The special symbol `INFTIM` is used to wait indefinitely. Transparent `ioctls` have no way to control the timeout. They will wait indefinitely.

Because modules and drivers stack in a stream, the first component to recognize an `ioctl` command will act on it. Modules pass along `ioctl` messages containing unrecognized commands. Drivers have the responsibility of failing unrecognized `ioctl` commands. More information on the details of `ioctl` processing can be found in Chapters 9 through 12.

Access to a stream is relinquished by calling the `close` system call with the file descriptor referring to the stream. The driver's close routine is only called on the last close of the stream. So if more than one process has opened the same stream, or if a STREAMS file descriptor has been duplicated [see `dup(2)`], the driver will not be notified that a close is occurring until the last file descriptor referring to the stream has been closed.

On last close, a stream is dismantled. Starting at the top of the stream, the system calls the close routine of each module before removing the module from the stream. When no modules are left, the system will call the driver's close routine and deallocate the data structures representing the stream.

Flushing Data

Since messages can be queued within a stream, applications have the ability to flush the data in it with the `I_FLUSH` `ioctl` command. By flush, we do not mean the ability to force that data to the tail of the stream. In this context, "flush" means to discard the data by freeing the messages.

The third parameter to the system call is a flag indicating which side of the stream to flush: `FLUSHR` for the read side, `FLUSHW` for the write side, and `FLUSHRW` for both sides. [`FLUSHRW` is equivalent to `(FLUSHR|FLUSHW)`.]

When the `I_FLUSH` `ioctl` is used, the stream head sends a special message (of type `M_FLUSH`) containing the flags downstream that informs the modules and driver to flush their queues. As each module receives the message, it flushes its queues and passes the message on to the next component. When the driver receives the message and flushes its queues, if `FLUSHR` is set, the driver shuts off `FLUSHW` and sends the message back upstream. When it reaches the head of the stream, the stream head flushes its read queue and frees the message. If `FLUSHR` is not set, the driver frees the message instead.

Flushing can also occur from within a stream. As the result of an external event, a module or driver can generate an `M_FLUSH` message to flush the stream. In this case, the user is unaware that the flushing has occurred. The stream head takes care of the `M_FLUSH` message much in the same way the driver did, but the sense of the flags is reversed. If `FLUSHW` is set, the stream head shuts off `FLUSHR` and sends the message downstream. Otherwise, it frees the message. The shutting off of the `FLUSHR` flag by the stream head and the `FLUSHW` flag by the driver prevents the message from circulating in the stream indefinitely.

Error Handling

The separation of module and driver processing from the user's I/O requests presents problems for error reporting. This is partially because of the decoupling effect of message-based interfaces, and partially because of the ability of modules and drivers to defer processing messages by queueing them. By the time the module or driver detects an error, the application may no longer be performing the system call that caused the error. Errors can also result from the tail of a stream, unrelated to any specific action by the user.

This has led to the use of error semantics that, in most cases, make a stream unusable when an error occurs. Usually when a module or driver detects an unrecoverable error, the action taken is to inform the stream head of the error by sending a message upstream, placing the stream in *error mode*. Then, from that point on, all system calls except `close` and `poll` will fail with the error code specified in the message. The only way for a user to clear the error is to close the stream and reopen the device.

One exception to this type of error handling is `ioctl` failures. The processing of an `ioctl` in a stream is synchronous; the user waits until a message arrives acknowledging the completion of the `ioctl` command. Drivers and modules indicate success or failure in the completion message. There is no need to place the stream in error mode, although this can be done in extreme cases. Drivers and modules usually try to avoid placing a stream in error mode unless absolutely necessary, because of the severe consequences.

Drivers and modules have the option of placing just one side of a stream in error mode. If only the read side is in error mode, then only read-like system calls will fail. If only the write side is in error mode, then only write-like system calls will fail. If a system call is neither read-like nor write-like, then it will fail if either side is in error mode.

Drivers and modules can also put the stream in *hangup mode*. This might occur when the driver detects a problem with the communication line, for example. In hangup mode, `read`s will succeed, retrieving any data on the stream head read queue, until no more data are left. Then `read` will return 0. However, `write` will fail in hangup mode, setting `errno` to `EIO`.

3.4 NONBLOCKING I/O AND POLLING

Normally, a read from a stream will block until data are available. A write will block if the stream is flow-controlled. An alternative to this form of I/O is called *nonblocking I/O*. If a stream is opened with the `O_NDELAY` flag or the `O_NONBLOCK` flag, then `read` and `getmsg` will fail with `errno` set to `EAGAIN` if there are no data immediately available, and `write` and `putmsg` will fail with `errno` set to `EAGAIN` if the stream is flow-controlled. For `write`, if part of the data has been written before the stream is flow-controlled, then `write` will return the number of bytes written.

These semantics are useful to applications that do not want to wait for data to either arrive or drain. This might be the case if the application has a lower-priority task it can perform until I/O can be continued. In some cases, an application might be able to use nonblocking I/O to improve its response time.

Nonblocking I/O alone would be tedious to use: an application would have to check the file descriptors periodically to see if the state had changed. With the use of the `I_SETSIG` `ioctl` command, this is not necessary. `I_SETSIG` provides a way for the application to be signaled when data arrive or flow-control restrictions are removed. The application specifies a bitmask of events it is interested in, and the stream head sends the `SIGPOLL` signal to the process when any of the events occurs. Then the application can either handle the event right away, or note that it occurred and handle it at its convenience.

Table 3.3 summarizes the events of interest that can be registered with the stream head. (Out-of-band data are discussed in Section 3.7.)

Table 3.3. Stream Head SIGPOLL Events

Event	Description
S_INPUT	Data (other than high-priority) can be read.
S_HIPRI	High-priority data can be read.
S_OUTPUT	Write side is no longer flow-controlled for normal data.
S_MSG	Signal message is at head of stream head read queue.
S_ERROR	Stream is in error (M_ERROR message received).
S_HANGUP	Device has hung up stream.
S_RDNORM	Normal data (band 0) can be read.
S_WRNORM	Same as S_OUTPUT.
S_RDBAND	Out-of-band data can be read.
S_WRBAND	Write side is no longer flow-controlled for out-of-band data.
S_BANDURG	Modifier to S_RDBAND to generate SIGURG instead of SIG-POLL.

Normal data are packaged in M_DATA messages and have a priority band of zero. S_INPUT is equivalent to (S_RDNORM|S_RDBAND) because read and getmsg remove the first message from the stream head read queue, regardless of the message's band. S_WRNORM is the same as S_OUTPUT because write and putmsg (without the RS_HIPRI flag) generate messages in band zero. S_WRNORM was added only to maintain a consistent naming scheme. S_BANDURG is used by the socket library described in Chapter 7.

The default action for the SIGPOLL signal is to kill the process. This may seem severe, but the stream head will never generate SIGPOLL unless the process explicitly requests it by invoking the I_SETSIG ioctl command. Therefore, applications must be careful to install the signal handler for SIGPOLL before using the I_SETSIG ioctl.

In a way, the default disposition of SIGPOLL encourages proper use of I_SETSIG. If you ask for signals to be generated without being prepared to handle them, then you pay the price. There is one unfortunate side effect, however. If a program has arranged for SIGPOLL generation and then execs, the signal dispositions for signals with handlers are restored to their default values. If the process that calls exec is a descendant of the process that called I_SETSIG, then everything is fine because SIGPOLL is only sent to the process that requested it. On the other hand, if the process calling exec is the same one that requested SIGPOLL be generated, then the new program will be killed if it receives SIGPOLL. Thus, before execing, processes using SIGPOLL should either disable its generation, ignore the signal, or set the close-on-exec flag for the file descriptors associated with the streams that might generate the signal.

Example 3.4.1. Data transfer to a stream can be illustrated with the cat(1) command. Since terminals are streams, we can use the standard output as the example of a STREAMS file descriptor to which data will be written.

Consider the simplified implementation of `cat` from Example 2.4.6. It is fairly straightforward, reading from the file and writing to the standard output until either there is an error, or the end of the file has been reached.

But what would happen if the standard output flow-controlled? The application would block in the `write` until the flow control subsided and the rest of the data could be written. We could make better use of this time if we were able to read more of the file until the standard output could accept more data. This version of the program employs nonblocking I/O to do just that:

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stropts.h>
#include <signal.h>

#define  BUFSIZE (64*1024)
#define  RDSIZE (BUFSIZE/8)

char buf[BUFSIZE];
int widx, ridx;        /* write and read indices */
int totwr, totrd;      /* total amounts read and written */
int flowctl;           /* 1 if flow-controlled, 0 if not */
int nfc;               /* number of times flow-controlled */

void catreg(int), cattostream(int);
int doread(int);
void dowrite(int), finwrite(void);
void setblock(int), setnonblock(int);

#ifdef FCBUG
void nop(int);
#endif

extern void error(const char *fmt, ...);
extern void fatal(const char *fmt, ...);

void
main(int argc, char *argv[])
{
     int i, fd, isoutstr;
#ifdef FCBUG
     struct sigaction sa;

     /*
      * If system contains flow-control bug,
      * install a signal handler for SIGALRM.
      */
     sa.sa_handler = nop;
     sigemptyset(&sa.sa_mask);
     sa.sa_flags = 0;
```

```
        if (sigaction(SIGALRM, &sa, NULL) < 0)
            fatal("cat: sigaction failed");
#endif

        /*
         * See if the standard output is a stream.  If
         * isastream fails, assume stdout is not a stream.
         */
        isoutstr = isastream(1);
        if (isoutstr == -1)
            isoutstr = 0;

        /*
         * Process each file named on the command line.
         */
        for (i = 1; i < argc; i++) {
            if ((fd = open(argv[i], O_RDONLY)) < 0) {
                error("cat: cannot open %s", argv[i]);
                continue;
            }

            /*
             * If the standard output is a stream, call
             * cattostream to print the file.  Otherwise
             * call catreg (see Example 2.4.6) to do it.
             */
            if (isoutstr)
                cattostream(fd);
            else
                catreg(fd);
            close(fd);
        }
#ifdef DEBUG
        printf("cat: number of flow controls = %d\n", nfc);
#endif
        exit(0);
}

#ifdef FCBUG
void
nop(int sig)
{
}
#endif
```

There are several differences between this version of `cat` and the one presented in Example 2.4.6. First, we have added a dummy signal handler, `nop`, for `SIGALRM`. It is only used if the symbol `FCBUG` is defined. `nop` is part of a workaround for a bug in STREAMS flow control found in versions of SVR4. (The bug has been fixed in SVR4.1 and SVR4.2.) The bug creates a window where the event that triggers the generation of `SIGPOLL` can be lost. The same bug can result in missing the event when using `poll`, too.

The second difference is the call to `isastream(3C)` to determine if the standard output is a stream. The synopsis for `isastream` is

```
    int isastream(int fd);
```

`isastream` returns 1 if `fd` is a file descriptor associated with a stream, 0 if not, and −1 on error. If an error occurs, we assume the standard output is not a stream. We use `catreg`, from Example 2.4.6, if file descriptor 1 is not a stream. Otherwise, we call `cattostream` to copy the contents of the file to the standard output.

Notice that the buffer is larger, but the read size (`RDSIZE`) will still be the same as the previous version. This is because we will use the b uffer in a circular f ashion: as we are copying data from one part of it, we will cop y data into another part, until we reach the end of the buffer, where we will jump back to the beginning of the buffer again, until no more data are left (see Figure 3.7).

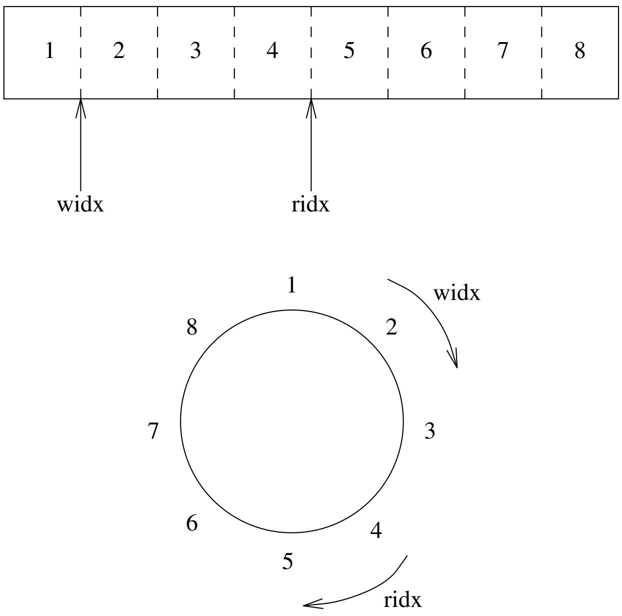

Fig. 3.7. Circular Buffer

Before writing to the stream, `cattostream` installs a signal handler for `SIGPOLL` and registers with the stream to be sent `SIGPOLL` whenever flow-control restrictions are removed. Then it places the stream in nonblocking mode by calling `setnonblock` (described shortly).

The basic flow is to read some data from the file, write the data to the terminal, and if flow control is asserted, continue reading from the f ile into the circular buffer until either there is no room left in the buffer or the `read` is interrupted by the delivery of `SIGPOLL`. We use `sigprocmask` to manage the critical sections of code where an error might occur if the e xecution is interrupted by the deli very of

SIGPOLL. If there is no room left in the b uffer, we have nothing to do but wait for flow-control restrictions to be lifted, so we use sigsuspend.

```
void
cattostream(int fd)
{
     int n;
     struct sigaction sa;
     sigset_t s, os;

     sigemptyset(&s);
     sigaddset(&s, SIGPOLL);

     /*
      * Install a signal handler for SIGPOLL.
      */
     sa.sa_handler = dowrite;
     sigemptyset(&sa.sa_mask);
     sa.sa_flags = 0;
     if (sigaction(SIGPOLL, &sa, NULL) < 0)
         fatal("cat: sigaction failed");

     /*
      * Arrange to be notified when the standard output
      * is no longer flow-controlled.  Then place the file
      * descriptor for stdout in nonblocking mode.
      */
     if (ioctl(1, I_SETSIG, S_OUTPUT) < 0)
         fatal("cat: I_SETSIG ioctl failed");
     setnonblock(1);
     totrd = totwr = 0;
     ridx = widx = 0;
     flowctl = 0;
     for (;;) {
         if ((n = doread(fd)) == 0) {
             /*
              * End of file; finish writing.
              */
             finwrite();
             break;
         } else if (n < 0) {
             /*
              * Read was interrupted by SIGPOLL.
              */
             continue;
         } else {
             /*
              * Successfully read something.
              */
             totrd += n;
         }

         /*
          * Start critical section.  Block SIGPOLL.
```

```
             * Then try to write what we've just read.
             */
            sigprocmask(SIG_BLOCK, &s, &os);
            dowrite(0);
            while (flowctl) {
                if (ridx != widx) {
                    /*
                     * Allow read to be interrupted.
                     */
                    sigprocmask(SIG_UNBLOCK, &s, NULL);
                    if ((n = doread(fd)) == 0) { /* EOF */
                        finwrite();
                        return;
                    } else if (n > 0) { /* read data */
                        totrd += n;
                    }
                    sigprocmask(SIG_BLOCK, &s, NULL);
                } else {
#ifdef FCBUG
                    /*
                     * Flow control bug -- might miss event.
                     */
                    alarm(1);
#endif
                    /*
                     * Atomically unblock SIGPOLL and
                     * wait to be interrupted.  On return,
                     * SIGPOLL is still blocked.
                     */
                    sigsuspend(&os);
#ifdef FCBUG
                    alarm(0);
                    if (ioctl(1, I_CANPUT, 0) != 0) {
                        /*
                         * Flow control lifted;
                         * continue writing.
                         */
                        flowctl = 0;
                        dowrite(0);
                        break;
                    }
#endif
                }
            }

            /*
             * End critical section.  Unblock SIGPOLL.
             */
            sigprocmask(SIG_UNBLOCK, &s, NULL);
        }
}
```

We use `alarm` so that we do not block indef initely if `FCBUG` is defined. To work around the flow-control bug, we use the `I_CANPUT` `ioctl` to check if we can write to the stream. If not, we just continue in the loop. If so, we clear the `flowctl` flag and call `dowrite` to continue writing to the stream. On systems that have fixed the flow-control bug, `dowrite` is called as the signal handler for `SIGPOLL`.

We use `setnonblock` to place a stream in nonblocking mode and `setblock` to restore it back to the default blocking behavior. Both functions get the current copy of the file flags for the stream and then turn the `O_NONBLOCK` flag either off or on. Modifying a copy of the current flags ensures that we do not change any of the flags already set; we just want to change the status of the `O_NONBLOCK` flag.

```
void
setnonblock(int fd)
{
     int fl;

     /*
      * Get the current file flags and turn on
      * nonblocking mode.
      */
     if ((fl = fcntl(fd, F_GETFL, 0)) < 0)
          fatal("cat: fcntl F_GETFL failed");
     if (fcntl(fd, F_SETFL, fl|O_NONBLOCK) < 0)
          fatal("cat: fcntl F_SETFL failed");
}

void
setblock(int fd)
{
     int fl;

     /*
      * Get the current file flags and turn off
      * nonblocking mode.
      */
     if ((fl = fcntl(fd, F_GETFL, 0)) < 0)
          fatal("cat: fcntl F_GETFL failed");
     if (fcntl(fd, F_SETFL, (fl&~O_NONBLOCK)) < 0)
          fatal("cat: fcntl F_SETFL failed");
}
```

When we have reached the end of the input file, `read` will return 0 and we call `finwrite` to finish writing the data to the standard output. In it, we attempt to cancel the `SIGPOLL` generation. If that fails, we just ignore `SIGPOLL`. The `I_SETSIG` should not fail, but we program defensively where we can. Then we disable nonblocking mode for the standard output and finish writing the data to the terminal.

```
void
finwrite(void)
{
     /*
      * Cancel SIGPOLL generation for stdout.
      */
     if (ioctl(1, I_SETSIG, 0) < 0) {
         struct sigaction sa;

         /*
          * I_SETSIG shouldn't have failed, but
          * it did, so the next best thing is to
          * ignore SIGPOLL.
          */
         sa.sa_handler = SIG_IGN;
         sigemptyset(&sa.sa_mask);
         sa.sa_flags = 0;
         if (sigaction(SIGPOLL, &sa, NULL) < 0)
             fatal("sigaction failed");
     }

     /*
      * Disable nonblocking mode and write last
      * portion to the standard output.
      */
     setblock(1);
     dowrite(0);
}
```

In the circular buffer, `ridx` is the index of the next location into which we read data, and `widx` is the index of the next location from which we write data. The maximum data read at once is given by `RDSIZE`. If we are less than `ridx` bytes from the end of the buffer, then we read only as many bytes as can fit in the buffer.

After reading, we increment `ridx` by the number of bytes read and if it reaches the end of the buffer, we reset it to 0. If `SIGPOLL` comes in during the middle of the `read`, then we can get one of two results: either we transferred some data to the buffer and `read` will return the number of bytes transferred, or we did not transfer anything and `read` fails with `errno` set to `EINTR`. Any other failure is a real error.

```
    int
    doread(int fd)
    {
        int n, rcnt;

        /*
         * Calculate the space left to read.
         * Read at most RDSIZE bytes.
         */
        rcnt = widx - ridx;
        if (rcnt <= 0) {
            /*
             * The writer is behind the reader
             * in the buffer.
             */
            rcnt = BUFSIZE - ridx;
            if (rcnt > RDSIZE)
```

```
                rcnt = RDSIZE;
        }

        /*
         * Read as much as we can.
         */
        n = read(fd, &buf[ridx], rcnt);
        if (n >= 0) {
            ridx += n;

            /*
             * If we've reached the end of the buffer,
             * reset the read index to the beginning.
             */
            if (ridx == BUFSIZE)
                ridx = 0;
        } else if (errno != EINTR) {
            fatal("cat: read failed");
        }
        return(n);
}
```

The routine that does the writing to the terminal, dowrite, is always called with SIGPOLL blocked. This is because the write might cause the stream to become flow-controlled, and we do not want the signal to be generated while we are still in dowrite or we could lose track of whether or not the stream is flow-controlled.

Consider what would happen if we did not block SIGPOLL and it came in after write failed because of flow control, but before flowctl was set to 1. dowrite would again be called, but this time as the handler for SIGPOLL. If it is able to write to the stream, it will set flowctl to 0. Then, when it returns to the original instance of dowrite, we continue by setting flowctl to 1, which is incorrect. If cattostream is able to read the remaining part of the file, then the error will be of no consequence, because cattostream will just call finwrite to write the contents of the buffer to the stream. If, however, the buffer fills without having read the entire file, then cattostream will call sigsuspend and never return because the stream is not really flow-controlled.

```
void
dowrite(int sig)
{
        int n, wcnt;

        while (widx >= ridx) {

            /*
             * Stop when the writer has caught up with
             * the reader.
             */
            if ((widx == ridx) && (totrd == totwr))
                break;
            /*
             * The writer is ahead of the reader in the
             * buffer.  Calculate the amount left to write,
             * and write it.
             */
            wcnt = BUFSIZE - widx;
            n = write(1, &buf[widx], wcnt);
            if (n < 0) {
                if (errno == EAGAIN) {
                    /*
                     * The stream is flow-controlled.
                     */
                    nfc++;
                    flowctl = 1;
                    return;
                } else {
                    fatal("cat: write failed");
                }
            } else {
                totwr += n;
            }
            widx += n;

            /*
             * If the write index has reached the end
             * of the buffer, reset it to 0.
             */
            if (widx == BUFSIZE)
                widx = 0;
        }
        while (widx < ridx) {
            /*
             * The writer is behind the reader in the buffer.
             */
            wcnt = ridx - widx;
            n = write(1, &buf[widx], wcnt);
            if (n < 0) {
                if (errno == EAGAIN) {
                    /*
                     * The stream is flow-controlled.
                     */
                    nfc++;
                    flowctl = 1;
                    return;
                } else {
                    fatal("cat: write failed");
                }
            } else {
                totwr += n;
            }
            widx += n;

            /*
             * If the write index has reached the end
             * of the buffer, reset it to 0.
             */
            if (widx == BUFSIZE)
                widx = 0;
        }

        /*
         * If we reach this point, write was able to
         * transmit everything, so we probably aren't
         * flow-controlled.
         */
        flowctl = 0;
    }
```

If `widx` is greater than `ridx`, then the most we can write is from `widx` to the end of the buffer. Otherwise, the most we can write is from `widx` to the next place to read, `ridx`. If `write` fails with `errno` set to `EAGAIN`, we set the `flowctl` flag to indicate flow control has been asserted. If it fails for any other reason, it is a real error, and we exit. If `write` succeeds, we increment `widx` by the number of bytes written. If `widx` goes past the end of the buffer, we set it back to 0. Note that if `widx` equals `ridx`, nothing is written (i.e., the ‘‘writer’’ has caught up with the “reader”).

Although more complex, this version of the program can execute several times faster than the first. This is because the program does other useful work while the output is flow-controlled. However, if the output device is fast enough such that it never flow-controls, there will be no increase in speed.

On a 33 MHz i386, I timed different versions of `cat` over three terminal devices: the console (whose screen contents are essentially mapped into the kernel address space), the asynchronous serial line connected to an ASCII terminal running at 19,200 bps (using the ASY driver), and the same serial line with an AT&T 630MTG windowing terminal running `layers` (using the XT driver, with ASY linked underneath it). The results from printing a 100,000-byte file are summarized in Table 3.4. All times are in seconds.

Table 3.4. Timing Results Writing to Different Devices

Device	`cat` from `/usr/bin`	`cat` from Example 2.4.6	`cat` from This Example
Console	36	36	36
ASY	285	273	122
XT	381	369	247

During the tests, the console stream never flow-controlled, the ASY stream flow-controlled twice, and the XT stream flow-controlled three times. Note that no increase of speed is seen with the console. That is because the console driver merely has to copy the message contents to the memory location representing the console screen to display text. This is a good example of the output device being fast enough to prevent the stream from flow-controlling.

Although this example completed faster than the other versions when the serial driver was used, the amount of time actually needed to display the text on the screen was almost equivalent. This is because this version of `cat` simply wrote its data and exited. It did not make the display rate any higher. The stream buffered as much data as the drivers and modules would allow. The drivers and modules continued to process data even though the command had exited because the shell still held the stream open. □

Polling

Applications that need to handle multiple file descriptors can use the `poll` system call to check pending conditions on multiple file descriptors at once. Rather than blocking in a system call on one of the file descriptors while events are occurring on others, applications can block waiting for events on any of a number of file descriptors. The synopsis for `poll` is

```
#include <sys/types.h>
#include <poll.h>
#include <stropts.h>

int poll(struct pollfd *parray, ulong_t nfds, int timeout);
```

An application supplies an array of `pollfd` structures, the number of entries in the array, and a timeout value. If nothing happens within the number of milliseconds specified by the timeout, the system call returns 0. The special timeout value of `INFTIM` (−1) causes the system call to block indefinitely until an event occurs on one of the file descriptors. The special timeout value of 0 prevents the poll from waiting (i.e., it peeks at the state of things and returns). The `pollfd` structure is defined as:

```
struct pollfd {
     int      fd;          /* file descriptor */
     short    events;      /* requested events */
     short    revents;     /* returned events */
};
```

An application sets the `fd` field to the file descriptor to be polled and the `events` field to the bitmask of events to be checked. On return, the `revents` field contains the subset of `events` that occurred, plus some that the system can set. The events about which the caller can request notification are listed in Table 3.5.

In addition to those events that the caller can request, three other events can be reported (see Table 3.6). The `POLLERR`, `POLLHUP`, and `POLLNVAL` events only apply to the `revents` field. An application cannot explicitly poll for these events by setting them in the `events` field.

Example 3.4.2. Most commands like `cu(1)` and `rlogin(1)` are implemented by forking, using one process to read from the terminal and write to the network, and the other process to read from the network and write to the terminal. This architecture

Table 3.5. Requestable `poll` Events

Event	Description
POLLIN	Data (other than high-priority) can be read.
POLLPRI	High-priority data can be read.
POLLOUT	Write side is no longer flow-controlled for normal data.
POLLRDNORM	Normal data (band 0) can be read.
POLLWRNORM	Same as POLLOUT.
POLLRDBAND	Out-of-band data can be read.
POLLWRBAND	Write side is no longer flow-controlled for out-of-band data.

Table 3.6. Nonrequestable `poll` Events

Event	Description
POLLERR	Stream is in error (M_ERROR message received).
POLLHUP	Device has hung up stream.
POLLNVAL	The file descriptor is invalid.

can be reimplemented using only one process and using `poll` to avoid blocking on either file descriptor. The following function illustrates how this might be done.

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

extern void error(const char *fmt, ...);

void
comm(int tfd, int nfd)
{
     int n, i;
     struct pollfd pfd[2];
     char buf[256];

     pfd[0].fd = tfd;    /* terminal */
     pfd[0].events = POLLIN;
     pfd[1].fd = nfd;    /* network */
     pfd[1].events = POLLIN;
     for (;;) {

         /*
          * Wait for events to occur.
          */
         if (poll(pfd, 2, -1) < 0) {
             error("poll failed");
             break;
         }
```

```
            /*
             * Check each file descriptor.
             */
            for (i = 0; i < 2; i++) {
                /*
                 * If an error occurred, just return.
                 */
                if (pfd[i].revents&(POLLERR|POLLHUP|POLLNVAL))
                    return;

                /*
                 * If there are data present, read them from
                 * one file descriptor and write them to the
                 * other one.
                 */
                if (pfd[i].revents&POLLIN) {
                    n = read(pfd[i].fd, buf, sizeof(buf));
                    if (n > 0) {
                        write(pfd[1-i].fd, buf, n);
                    } else {
                        if (n < 0)
                            error("read failed");
                        return;
                    }
                }
            }
        }
    }
```

The `comm` function sets up the `pollfd` array and performs a poll that will block until data arrive on either file descriptor. The variable `i` indicates the file descriptor to check. If there was an error or hangup on the stream, we return. If there are data to be read, the `POLLIN` flag will be set and we read the data from the stream and write them to the other stream. Since we only have two indices, 0 and 1, the idiom `1-i` gives us the other index. Then we perform the same task with the other stream. We continue this until there is an error.

Do not be misled by the example. Both `cu` and `rlogin` are a lot more complex than this example might imply. They are both complicated by signal handling and local command (“escapes”) processing. `rlogin` uses out-of-band data to implement end-to-end flow control and to propagate interrupts across the network. `cu`, on the other hand, transmits all characters in a single band, sometimes resulting in awkward delays between the time that special characters (such as interrupt and stop) are typed and the time that the remote computer reacts to them. This is mainly because other characters that were transmitted before the special ones might still be buffered, delaying the remote machine from interpreting the special ones. This behavior is because `cu` was not designed from the start to work over networks. It was designed to work primarily with asynchronous serial communication devices and was not converted to use the transport-level network interface until SVR3. Even now, it does not attempt to use out-of-band transmission services if the underlying communication facility supports them. □

3.5 SERVICE INTERFACES

A *service interface* is the boundary between a user (called the ‘ ‘consumer’’) and the provider of a service. A service interface consists of the primiti ves that can be passed across the interf ace and the rules specifying the state transitions that occur when the primitives are generated and received. The state transitions imply an ordering of primiti ves; for e xample, an ackno wledgement being sent in response to a request. The consumer can generate request primiti ves to obtain service from the provider. The provider, in turn, may answer a request with a response primiti ve. External events might cause the provider to notify the consumer with an e vent primitive (see Figure 3.8).

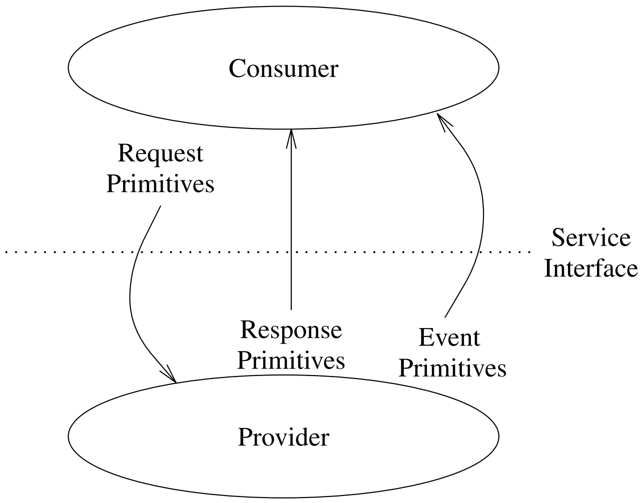

Fig. 3.8. Service Interface

Service interfaces are closely related to the OSI reference model discussed in Chapter 1. At each layer in the model, an interf ace exists between the upper and lower layers. Service interfaces are modeled after these interf aces. The layer beneath the interface provides a service to the layer above the interface.

The purpose of specifying a service interf ace is to allo w consumers of services to be written independent of the pro viders of the service. This enhances the possibility for different consumers to be matched up with dif ferent providers, as long as they share a common service interf ace. The Transport Provider Interface (TPI) is one example of a service interface. It is discussed in detail in Chapter 12.

Consider an application (the consumer) that lets users remotely log in to dif ferent computers connected to a netw ork. If the application conforms to a published service interface, then it is possible to ha ve a v ariety of netw ork protocols (the providers) that support remote login just by conforming to the same interf ace. Figure 3.9 illustrates this point. In it, TIMOD is a module that helps the transport interface library routines conform to the proper service interface, the TPI. Two transport providers are shown, each implementing a different protocol. TCP is the Internet's Transmission Control Protocol, and TP4 is ISO's class 4 transport protocol.

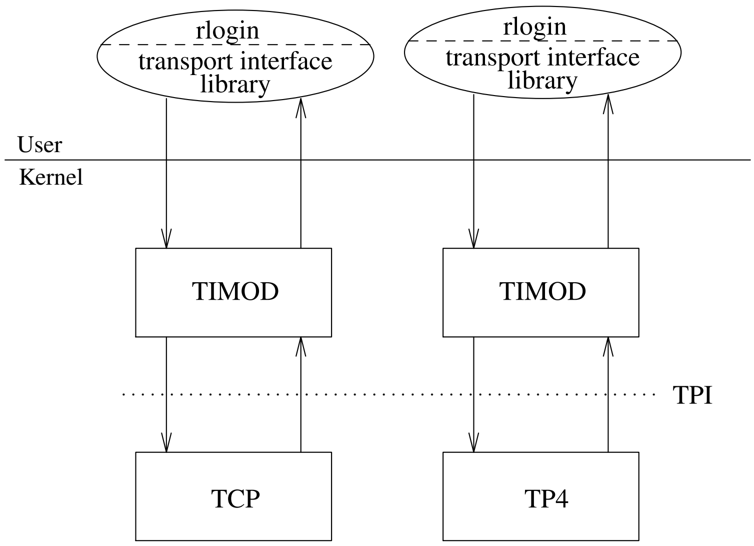

Fig. 3.9. Service Interfaces Promoting Software Reuse

Service interfaces can exist at the boundary between any two processing elements in a stream. The service interfaces provide the separation and isolation needed to implement layered protocols.

In System V, service interface primitives are implemented as STREAMS messages. Usually, primitives are constructed by creating `M_PROTO` message blocks containing control information, such as the primitive type and addressing information, and linking these to `M_DATA` message blocks containing the associated user data. By separating user data from control parameters, protocol-independent filtering modules can be used in the stream (they process only the `M_DATA` portions of messages). For example, you could write a module that encrypts data in `M_DATA` messages and push it on a network connection's stream. Both sides of the connection would have to push the same module and then inform the module of the correct key to use to encrypt and decrypt the data. The module could then operate by processing all `M_DATA` messages it receives, including the ones linked to `M_PROTO` and `M_PCPROTO` messages. The module would not have to know anything about the protocol used, because the module operates on user data only. Thus, the module could be used with different protocols.

The reason that the `getmsg` and `putmsg` system calls exist is to provide a means for user-level applications to communicate across service interfaces. The message-based service interface used in the kernel between processing elements in a stream is extended to applications running at user level by the ability of `getmsg` and `putmsg` to process both control information and user data.

A less obvious benefit of service interfaces is the ability to migrate protocols from the kernel to intelligent peripheral devices. Since applications deal with a fixed service interface, all that need be done is to provide a driver that presents the same service interface to users while communicating the primitives to the peripheral device where the real work is done. The system calls are unaffected by changes like this because they know nothing about the communication involved; they merely act as a message-transfer mechanism. Protocol processing can even migrate between user level and kernel level as long as the same service interfaces are provided for existing applications.

An example of a service primitive is a request to send data. The primitive type and addressing information would be contained in the control buffer, and the user data would be contained in the data buffer.

Example 3.5.1. A hypothetical (and inefficient) protocol might require that a positive or negative acknowledgement be sent to the consumer for every data request generated. Combining the routines from two previous examples (3.3.2 and 3.3.4), we can write a routine that implements this service interface.

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stropts.h>

int
send(int fd, char *buf, uint_t blen, char *addr,
     uint_t alen)
{
     sigset_t set, oset;

     /*
      * Block SIGPOLL.
      */
     sigemptyset(&set);
     sigaddset(&set, SIGPOLL);
     sigprocmask(SIG_BLOCK, &set, &oset);

     /*
      * Send the message.
      */
     if (senddata(fd, buf, blen, addr, alen) < 0) {
          sigprocmask(SIG_SETMASK, &oset, NULL);
          return(-1);
     }

     /*
      * Receive the acknowledgement.
      */
```

```
        if (getack(fd) < 0) {
            sigprocmask(SIG_SETMASK, &oset, NULL);
            return(-1);
        }

        /*
         * Restore the original signal mask.
         */
        sigprocmask(SIG_SETMASK, &oset, NULL);
        return(0);
    }
```

This is a simple example, but it illustrates a problem with library routines used to implement service interface primitives that require a response. The response message might generate SIGPOLL if the caller had previously called I_SETSIG and the message type of the response corresponds to an input event in which the caller was interested. If we ignore SIGPOLL in the library routine, the caller might miss the arrival of data. If we block (hold) SIGPOLL, then we might generate an event because of the arrival of the response message when there is really nothing for the caller to read.

Luckily, response primitives are usually M_PCPROTO messages, and applications are usually only interested in the arrival of normal and out-of-band data. This is because only one high-priority message is enqueued at one time on a stream head's read queue, so it is unlikely that M_PCPROTO messages are used for anything other than interface acknowledgements. (Otherwise, primitives would be lost when additional M_PCPROTO messages arrived.) In the event that the caller wants to be signaled when high-priority messages arrive, it is probably better that we generate a false event than to remain silent, so we choose to hold SIGPOLL instead of ignoring it. □

Examples of more realistic service interfaces are discussed in later chapters.

3.6 IPC WITH STREAMS PIPES

We briefly introduced STREAMS-based pipes in Section 3.2. In this section we will explore the architecture of pipes, along with some of the side effects of this architecture. Then we will see how STREAMS pipes are used for interprocess communication.

A pipe is created with the pipe system call:

```
#include <unistd.h>

int pfd[2];

if (pipe(pfd) < 0) {
     perror("pipe failed");
     exit(1);
}
```

Figure 3.10 shows the structure of a pipe. Each file descriptor has its own stream head, with the write queue pointing at the other's read queue. Data written using one file descriptor will be packaged into messages by the stream head and placed on the read queue of the stream head at the other end of the pipe. These data are then available to be read using the other file descriptor.

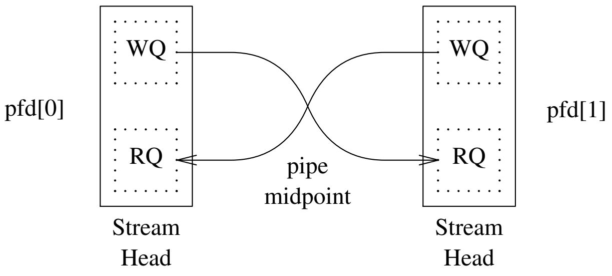

Fig. 3.10. Queue Linkage in a Pipe

If a module is pushed on one end of the pipe, it cannot be popped from the other end (see Figure 3.11). To be removed, it must be popped from the same end on which it was pushed. This is because the pipe is really two separate streams linked together. The point where the streams "twist" (where the write queues point to the read queues) is the midpoint of the pipe. Operations that are local to a stream will end here, such as searching for modules to pop, or listing the modules in the stream.

Recall that multiple processes that open the same device share the same stream. This is also true for pipes. Since a pipe is a stream, when more than one process is writing to the same end of a pipe at the same time, the data from one process will be interleaved with the data from the other processes.

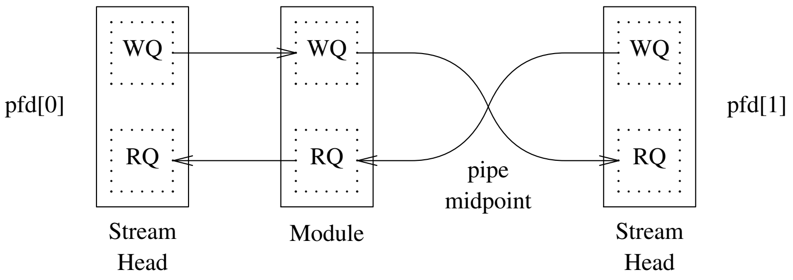

Fig. 3.11. A Module Pushed on a Pipe

Historically, pipe semantics have guaranteed that if a process writes no more than `PIPE_BUF` bytes, the data will not be fragmented and interleaved with data from other processes. In other words, if a process writes at most PIPE_BUF bytes in one system call, the process reading from the other end of the pipe (assuming it reads the same amount) will receive all the data written by the writer, without the data being intermixed with data from other writers. If, however, the writer sends the data with more than one write system call, data from other writers might be interleaved with the data from the separate system calls.

Different UNIX systems support different values for PIPE_BUF. Two common values are 4096 and 5120 bytes. POSIX_PIPE_BUF defines the minimum value that any POSIX-conforming system must support. Both constants are defined in <limits.h>. Portable applications should not assume that the value for PIPE_BUF is greater than POSIX_PIPE_BUF. A portable way does exist, however, to find the value of PIPE_BUF for a given system: use pathconf(2) or fpathconf(2).

For example, assume two processes, A and B, are writing to the same end of a pipe, and that 100 < PIPE_BUF < 10,000. If process A writes 100 bytes and process B writes 100 bytes, then the reader will receive 100 bytes from one process, followed by 100 bytes from the other process, depending on who wrote first. Now, if process A decides to write 10,000 bytes, but process B only writes 100 bytes, the reader could receive the first PIPE_BUF from process A, then the 100 bytes from process B, followed by the remaining data (10,000 − PIPE_BUF) from process A. The order depends on a number of factors, such as process scheduling priority and availability of buffers for STREAMS messages.

If a module is pushed on one end of the pipe, these semantics are maintained as long as the module's maximum packet size is at least PIPE_BUF bytes. The maximum packet size determines how the data are fragmented when written. Of course, with modules on the pipe, the order of arrival might change depending on how the modules process the data.

Even though pipes are streams, they exhibit slightly different behavior during nonblocking I/O, because of older pipe semantics. For compatibility, if a pipe's stream is placed in nonblocking mode with the O_NDELAY flag, read will return 0 when no data are available, and write will block when the stream is flow-controlled. If the stream is placed in nonblocking mode with the O_NONBLOCK flag, read will return −1 with errno set to EAGAIN when no data are available, and write will return −1 with errno set to EAGAIN when the stream is flow-controlled.

If one end of a pipe is closed, then the other end is placed in hangup mode. Thus, processes cannot write to the open end, but they can still read from it. Processes will be sent SIGPIPE when they try to write to a pipe when the other end is not in use. The default action for SIGPIPE is to kill the process, so processes that do not want to be killed in this manner when using pipes should either catch or ignore SIGPIPE if there is a chance that one end will go away abruptly.

Flushing Data in a Pipe

An interesting problem occurs with flushing data in a pipe. Recall that flushing occurs by passing an `M_FLUSH` message through the stream. The message contains flags that specify the queues to flush (read-side, write-side, or both). At the point where the two streams forming the pipe meet, the read side of one stream becomes the write side of the other, and vice versa. So if only one side of the stream is flushed, the flag will refer to the opposite side after passing the midpoint of the pipe.

Referring back to Figure 3.10, let's look at what happens when we try to flush the stream using

```
    ioctl(pfd[0], I_FLUSH, flag);
```

If `flag` is `FLUSHR`, then the stream head read queue referred to by `pfd[0]` will be flushed. If `flag` is `FLUSHW`, then the stream head read queue referred to by `pfd[1]` will be flushed, since data written to `pfd[0]` are placed on the read queue of `pfd[1]`. If `flag` is `FLUSHRW`, then both read queues are flushed. The stream head never enqueues messages on its write queue, so there is no need to flush it. Note that no message is generated to flush the pipe when no modules are present.

If there are modules on the stream pipe (see Figure 3.11), however, a message is used instead, in the same way as if there were a driver at the end of the stream. Now let's see what happens in this case. If only `FLUSHR` is set in the `M_FLUSH` message, the module would flush its read queue and pass the message to the stream head of `pfd[1]`. Then that stream head would flush its read queue, which is not what we intended. Since the read queue of `pfd[1]` holds messages we write to `pfd[0]`, flushing the read queue of `pfd[1]` would only make sense if we had specified `FLUSHW`.

If only `FLUSHW` is set, the module would flush its write queue and pass the message to the stream head of `pfd[1]`. The stream head would then route the message down its write side, passing the message back to the module. The module would again flush its write queue and pass the message back to where it originated, the stream head of `pfd[0]`. This time, however, the stream head will not route the message back down the write side of the stream. It is smart enough to prevent the flush message from circulating more than once in the pipe.

The problem of flushing the wrong queues in a pipe exists because, at the midpoint, the side of the stream where processing is taking place changes, but the sense of the flags in the `M_FLUSH` message does not. To flush one side of a pipe correctly, we need to flip the sense of the flags. For example, to flush the write side, we should flush all the write queues from the stream head down to the midpoint, then switch and flush all the read queues on the other side from the midpoint up to the other stream head.

To solve this problem, a module (called PIPEMOD) is available to switch the sense of the flush flags. PIPEMOD performs no other function. If an application requires modules on the pipe and the ability to flush data, then PIPEMOD should be the first module pushed on the stream, closest to the midpoint. (It only needs to be on one side of the pipe.) Otherwise, it can be left off the stream. In addition to

PIPEMOD, other modules intended for use in pipes may incorporate the necessary logic to switch the sense of the flush flags.

Mounted Streams

When a process needs to communicate with one of its children, it usually does so through a pipe. The parent enables communication by creating a pipe before forking off the child. After the child is created, the parent will write to and read from one of the two file descriptors returned from the `pipe` system call. The child will use the other end of the pipe.

More interesting, however, is the use of pipes for communication between unrelated processes. By giving one end of a pipe a name in the file system, a process offers a way for other processes to communicate with it. FIFOs provide this capability, but communication is unidirectional. Mounted streams overcome this problem. Here's how: a process creates a pipe and mounts one end of it over an existing file using `fattach(3)`. Then, when other processes open the file, they gain access to the mounted end of the pipe instead of the preexisting file. Communication can occur as in the parent/child case, with the original process reading from and writing to the unmounted end of the pipe, and the other processes reading from and writing to the mounted end of the pipe.

The `fattach` library routine uses the `mount` system call and a special file system called NAMEFS to support mounted streams (also called ''named streams''). The synopsis for `fattach` is

```
int fattach(int fd, const char *path);
```

The process must own and have write permission on the file represented by `path`, or have super-user privileges. After the stream is attached, processes opening the file will gain access to the pipe. Any processes that have the file open before the call to `fattach` will still access the original file after the attach is made, but if they open the file again, they will access the pipe instead.

Example 3.6.1. One problem that arises with window systems is that they may render your login terminal device useless. Others cannot write to your terminal unless the windowing software has replaced the login terminal's entry in `/var/adm/utmp` with the name of the new device to which messages can be sent. Even though library routines exist to make this change easier [see `getut(3C)`], they can cause problems because they do not serialize access to the data files being changed.

Instead of changing the `utmp` entry, a windowing system can elect to display messages written to your terminal in the current window or in a reserved window by monitoring the end of the pipe returned by this function:

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

#define TTYMODE  (S_IRUSR|S_IWUSR|S_IWGRP)
```

```
int
chgterm()
{
     int pfd[2];
     char *tty;

     /*
      * Get the name of the controlling terminal.
      */
     if ((tty = ttyname(0)) == NULL)
         return(-1);

     /*
      * Create a pipe and mount one end on top of
      * the terminal's device node.  Then change
      * the mode of the pipe to give it the same
      * permissions as terminals.
      */
     if (pipe(pfd) < 0)
         return(-1);
     if ((fattach(pfd[1], tty) < 0) ||
       (chmod(tty, TTYMODE) < 0)) {
         close(pfd[0]);
         close(pfd[1]);
         return(-1);
     }

     /*
      * Close the end of the pipe just mounted and
      * return the other end to the caller.
      */
     close(pfd[1]);
     return(pfd[0]);
}
```

On success, the function returns a file descriptor to be monitored, or −1 if it fails. It obtains the name of the terminal that was used to log in, creates a pipe, and mounts one end of the pipe over the name. Then it makes the name publicly writable. This does not change the mode of the on-disk file representing the terminal name, as you might expect. Instead, it changes the mode of the pipe while it is attached to the name, leaving the mode of the on-disk file unaltered. From the output of `ls -l /dev/ttyXX`, it will appear as if the terminal name is a pipe with permission 0620, but as soon as the pipe is unmounted, things will revert to the actual device mode and permissions.

One catch with using this method is that the terminal device will not act as a terminal when opened by other processes. Since many programs behave differently when they are writing to a terminal, it might be prudent for the windowing system to push a special module on the pipe to make the pipe appear as a terminal. Such a module is easy to write, as we shall see in Chapter 11. □

A process can unmount a stream that has been attached to the file system namespace with `fdetach(3)`:

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

Unless both ends of a pipe are mounted, if one end of a pipe is closed while the other end is mounted, then the mounted end is automatically unmounted. This is why we do not have to call `fdetach` if `chmod` fails in the previous example.

Sharing and Passing File Descriptors

When two processes want to share file descriptors, they usually have to be related in some way. When a process forks, the child process has access to all file descriptors in use by the parent before the fork. Any files opened after the fork are private to the process that opened them. This is useful when the parent needs to fork off a child to perform some task.

Passing file descriptors between unrelated processes can be accomplished in this manner by having the child `exec` the program that is to receive the file descriptors. This approach is inconvenient because it restricts the types of programs that can make use of it. For example, processes already running would not be able to receive file descriptors since the passing can only be done at the creation of the process. Additionally, the file descriptors to be shared are limited to only those file descriptors in use before the `fork`.

The `I_SENDFD` and `I_RECVFD` `ioctl` commands improve this situation by allowing unrelated processes to pass file descriptors to each other. When a process wants to send a file descriptor to another process, it can use the `I_SENDFD` `ioctl` command in conjunction with a mounted pipe:

```
#include <stropts.h>

if (ioctl(pipefd, I_SENDFD, otherfd) < 0)
     perror("Cannot send file descriptor");
```

The first parameter, `pipefd`, must be a file descriptor of a stream pipe or a loop-back driver. The third parameter, `otherfd`, is the file descriptor to be sent. The only requirement is that it be a valid open file descriptor.

After sending `otherfd`, the process can close it if the file descriptor is not needed. The system will keep it open until the receiving process closes it. To pass a file descriptor, the stream head creates an `M_PASSFP` message, obtains a reference to the file pointer, places the file pointer and identification of the sender in the message, and enqueues the message on the stream head read queue at the other end of the pipe.

A process can then receive the file descriptor with the `I_RECVFD` `ioctl` command:

```
#include <stropts.h>

struct strrecvfd recv;

if (ioctl(pipefd, I_RECVFD, &recv) < 0)
     perror("Cannot receive file descriptor");
```

The third argument is a pointer to a `strrecvfd` structure that describes the new file descriptor:

```
    struct strrecvfd {
         int      fd;
         uid_t    uid;
         gid_t    gid;
    };
```

The `fd` field contains the new file descriptor. The `uid` field contains the effective user ID of the sending process. Similarly, the `gid` field contains the effective group ID of the sending process.

If an `M_PASSFP` message is at the front of the stream head read queue, any attempts to use `read` or `getmsg` on that stream will fail with `errno` set to `EBADMSG`.

Example 3.6.2. Suppose client applications must authenticate their network connections so they can obtain service. Instead of having every client application call a function to do the authentication, we can delegate a separate process on each machine to handle the authentication. Separating the authentication from the clients has the advantage that we can change the authentication scheme at any time without having to change all of the applications; we just replace the authenticator process instead. (A similar technique can be applied to server processes as well.)

One way to implement the interface to the authenticator process is to pass the network connection to the process, let it do the work, and then have it pass the file descriptor back to the client. For example, the authenticator might use the user ID found in the `strrecvfd` structure as an index into a database containing public and private keys (passwords) to be used in authentication. A function similar to the following can be used to exchange file descriptors with the authenticator:

```
    #include <fcntl.h>
    #include <unistd.h>
    #include <stropts.h>
    #include <errno.h>

    int
    auth(int netfd)
    {
         int afd;
         struct strrecvfd recv;

         /*
          * Open a mounted stream pipe to the authenticator.
          */
         if ((afd = open("/var/.authpipe", O_RDWR)) < 0) {
             close(netfd);
             return(-1);
         }

         /*
          * Send the network connection to be authenticated.
          */
         if (ioctl(afd, I_SENDFD, netfd) < 0) {
             close(afd);
             close(netfd);
             return(-1);
         }

         /*
          * We don't need the network file descriptor.
          * The authenticator will pass us back the
          * file descriptor to use.
          */
         close(netfd);
         if (ioctl(afd, I_RECVFD, &recv) < 0) {
             recv.fd = -1;
         } else if (recv.uid != 0) { /* impostor */
             close(recv.fd);
             errno = EACCES;
             recv.fd = -1;
         }
         close(afd);
         return(recv.fd);
    }
```

The first step is to open the mounted pipe (/var/.authpipe) that the authentication server is using. It is usually a good idea to start pipe names with a period so people grepping around will not indefinitely block reading from the pipe. If the open fails, we close the network file descriptor since we cannot authenticate the connection, and we return failure.

If we can open the named pipe, we send the file descriptor to the authentication process. Then we close the network file descriptor since we do not need it anymore and receive the authenticated file descriptor from the authentication process. If the authenticator is not running with super-user privileges, we discard the file descriptor and set `errno` to `EACCES` to indicate a permissions problem. This limits the ability of someone being able to spoof the authentication facility.

If everything checks out, we close the pipe's file descriptor and return the file descriptor associated with the authenticated network connection. □

Unique Connections

There is a bug in the previous example. If more than one process is trying to authenticate a connection at the same time, then we might end up with the authenticated file descriptors going to the wrong processes. With more than one process trying to receive a file descriptor from the same end of a pipe, the first process to run will get the file descriptor. This is the same problem that exists with both FIFOs and mounted streams: the reader cannot distinguish data from multiple writers, and multiple readers cannot contend for data from the same end of a pipe without some sort of synchronization.

This problem can be solved by using a special module called CONNLD. The server process can push CONNLD on the mounted end of the pipe. Then, whenever a process opens the pipe, it will get a unique connection to the serv er. The way this works is that CONNLD creates a second pipe and sends one end to the serv er process. Then it arranges to ha ve the other end returned to the client process as the returned file descriptor from `open`. The client process will block until the serv er receives the file descriptor with the `I_RECVFD` `ioctl` command. Then both processes can communicate o ver the unique connection without interference from other processes.

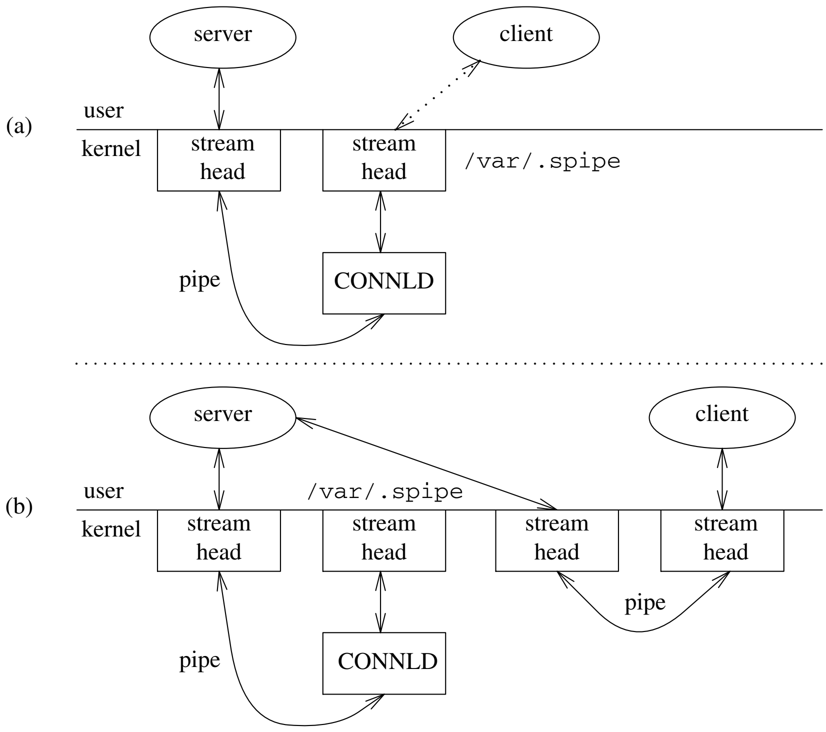

Fig. 3.12. Unique Connections with CONNLD

Figure 3.12 shows a process opening a pipe (with CONNLD on it) mounted on the file named `/var/.spipe`. In part (a), the client process attempts to open the pipe and goes to sleep. After CONNLD creates a ne w pipe and sends one end to the server process, the serv er receives it, waking up the client. Part (b) of the f igure shows the end result. The server process has a ne w file descriptor that refers to one end of a pipe that goes directly to the client process. The client process does *not* gain access to the mounted pipe. Instead, the mounted pipe is replaced by the ne wly created pipe before `open` returns.

Example 3.6.3. The following two routines can be used by a serv er process to provide unique connections on a mounted pipe. The first routine creates the pipe, pushes CONNLD on one end, and then mounts that end in the file system. This is all the initialization needed to enable unique connections. The second routine is a stub for the authentication process discussed in the previous example.

```
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stropts.h>

#define PIPEPATH "/var/.authpipe"
#define ALLRD    (S_IRUSR|S_IRGRP|S_IROTH)
#define ALLWR    (S_IWUSR|S_IWGRP|S_IWOTH)
#define PIPEMODE (ALLRD|ALLWR)

int pfd[2];

extern void fatal(const char *fmt, ...);

void
initialize()
{
     /*
      * Create a pipe.
      */
     if (pipe(pfd) < 0)
         fatal("cannot create pipe");

     /*
      * Push CONNLD on one end to enable unique
      * connections.
      */
     if (ioctl(pfd[1], I_PUSH, "connld") < 0)
         fatal("cannot push CONNLD");

     /*
      * Create a place to mount the pipe.
      */
     close(creat(PIPEPATH, PIPEMODE));

     /*
      * Mount the end of the pipe containing
      * CONNLD on PIPEPATH.
      */
     if (fattach(pfd[1], PIPEPATH) < 0)
         fatal("cannot attach pipe to file system");
}

void
serve()
{
     struct strrecvfd recv;
```

```
        struct strrecvfd conn;
        int okay;

        for (;;) {
            /*
             * Get the file descriptor of the pipe
             * connected to the local client process.
             */
            if (ioctl(pfd[0], I_RECVFD, &conn) < 0)
                continue;

            /*
             * Get the file descriptor to be authenticated.
             */
            if (ioctl(conn.fd, I_RECVFD, &recv) < 0) {
                close(conn.fd);
                continue;
            }

            /*
             * Authenticate the connection.
             */
            okay = doauth(recv.fd);

            /*
             * Send authenticated file descriptor back to
             * the client process.
             */
            if (okay)
                ioctl(conn.fd, I_SENDFD, recv.fd);

            /*
             * Close the file descriptors we no longer need.
             */
            close(conn.fd);
            close(recv.fd);
        }
    }
```

The second routine implements a sample service loop. The first `I_RECVFD` obtains the file descriptor for one end of the unique pipe connection to the client. The second `I_RECVFD` obtains the file descriptor of the network connection that is to be authenticated. If authentication succeeds, the authenticated file descriptor is passed back to the client. If authentication fails, the pipe is closed, resulting in the client's `I_RECVFD` `ioctl` request failing. □

Another interesting problem occurs with using pipes for communication between unrelated processes. By placing a pipe in the file system namespace, applications can expose themselves to unwanted communication, through either error or mischief. If the access permissions of a mounted pipe are unrestrictive, as they often must be, processes other than those intended can attempt communication. Since they might not follow the proper protocol, server applications have to be less trusting.

For example, suppose an application were written to expect a message following a fixed format, and a client only wrote half of the message. Then the next process to write to the pipe would have the first part of its message interpreted as the second half of the previous (incomplete) message.

Applications can be made less susceptible to these types of errors by following a few simple rules:

1. If you read less than you expect to, discard the data.
2. Validate the data received, if possible.
3. If you are using `read` and get `EBADMSG`, there might be a protocol message or file-descriptor message on the stream head read queue. If it is a protocol message, retrieve it using `getmsg` and discard the information. If it is a file-descriptor message, use `I_RECVFD` to get the file descriptor and then close it right away. If you are using `getmsg` and get `EBADMSG`, then there is a file-descriptor message on the stream head read queue. As in the read case, receive it and close it.
4. Put the stream in *message discard* mode so any data remaining in the message after you have read what you needed are discarded (see the next section for more details).

3.7 ADVANCED TOPICS

This section will cover some less commonly used aspects of STREAMS, including read modes, write modes, and priority bands.

Read Modes

By default, `read(2)` treats data on the stream head read queue as a byte stream. This means it ignores message boundaries. If the user issues a `read` large enough, data will be retrieved from multiple messages on the queue until either the queue is empty or the `read` request size is satisfied. As discussed earlier, `read` only works if the message on the front of the stream head read queue is of type `M_DATA`.

These characteristics can be changed with the `I_SRDOPT ioctl` command. Of the six modes, three control how `read` processes data in messages on the queue and three control how `read` treats nondata messages. The current read modes in effect for a stream can be obtained with the `I_GRDOPT ioctl` command.

Usually, *byte stream mode* (`RNORM`, the default) will suffice for most applications, but there are two other mutually exclusive modes that applications can use. The first, *message discard mode* (`RMSGD`), will prevent `read` from returning data in more than one message if the amount requested is greater than the amount of data in the first message on the stream head read queue. If the application requests less, then the remainder of the message is discarded. This is useful when an application employs a protocol that uses fixed-size messages. If a process sends more data than required, the excess will be discarded when read by the application. If a process writes less than required, the `read` will not consume successive messages to satisfy the amount of data missing from the first message.

The second mode is *message nondiscard mode* (`RMSGN`). Like `RMSGD` mode, this mode will prevent `read` from returning data in more than one message if the amount requested is greater than the amount of data in the first message on the stream head read queue. Unlike `RMSGD` mode, however, if the application reads less than the amount of data in the first message, the remainder of the message is placed back on the front of the stream head read queue, to be obtained by the next `read`. This mode is useful when an application does not know how much data to expect, wants to do large reads for performance, and expects data to be written in one message. It is equivalent to `getmsg` without control information.

Three mutually exclusive modes control the treatment of protocol messages (`M_PROTO` and `M_PCPROTO`) by the `read` system call. The default mode, `RPROTNORM`, causes `read` to fail with `errno` set to `EBADMSG` when a protocol message is at the front of the stream head read queue. Applications can use this mode to recognize when protocol messages arrive.

The second mode is `RPROTDIS`, *protocol discard mode*. If an `M_PROTO` or `M_PCPROTO` message is at the front of the stream head read queue and a process tries to read it in this mode, the control portion is discarded and any data that were contained in `M_DATA` blocks linked to the protocol message are delivered to the process as the data that are read. This mode might be set by an application preparing a stream for reading by another application that has no knowledge of the service interface being used, as long as only data-transfer primitives are used.

The third mode is *protocol data mode*, `RPROTDAT`. In it, the protocol messages are converted to data messages and their contents are returned to the user as data when `read` is used. It has no immediate use, but was added to the system because it was a logical extension of the other modes, easily fell out of the design of read modes, and might prove useful in the future in implementing some applications found in other UNIX variants.

Write Modes

The two `write(2)` modes can be set with the `I_SWROPT` `ioctl` command and obtained with the `I_GWROPT` `ioctl` command.

The first mode only applies to pipes. For compatibility, a `write` of zero bytes to a pipe will have no effect. The `SNDZERO` option enables zero-length message generation. Applications usually use zero-length data messages to represent end-of-file, but some existing programs write zero bytes to their standard output. When used in a pipeline, these applications might not work as expected. So if an application wants to generate zero-length data messages on a pipe, it needs to enable the `SNDZERO` option explicitly.

The second `write` mode applies to both `write` and `putmsg` on any stream. If a stream is in either error mode or hangup mode, the `SNDPIPE` option will cause `SIGPIPE` to be sent to processes that try to use `write` or `putmsg` on the stream. This is used to support 4BSD socket semantics over STREAMS.

Priority Bands

Each STREAMS message belongs to a particular priority band for the purposes of flow control and message queueing. The priority band of a message determines where it is placed on a queue with respect to other messages. In addition, each priority band has a separate set of flow-control parameters. Users usually do not have to concern themselves with priority bands unless the service interface they are using requires it.

Priority bands are used for features like expedited data (out-of-band data), where one class of data message has a higher priority than another. The OSI definition of expedited data specifies that they are unaffected by the flow-control constraints of normal data. This was the primary motivation for the addition of priority bands. Rather than add a facility to support only expedited data (one out-of-band data path), a more general facility was implemented, supporting up to 256 bands of data flow. Normal data belong in band 0. Expedited data are implemented in band 1. It is not expected that many more bands than this will be used, but if the need ever arises, the operating system will not have to be changed again.

High-priority messages are always first in a queue. These are followed, in order of decreasing band number, by the other message types. Normal-priority (band 0) messages are found at the end of the queue.

Two system calls were added to deal with priority bands at the service interface between the user and the kernel. An application can use `putpmsg(2)` to generate a message in a particular priority band:

```
#include <stropts.h>

int putpmsg(int fd, const struct strbuf *ctlp,
     const struct strbuf *datp, int band, int flags);
```

The first three arguments are the same as in `putmsg`. The `band` argument specifies the priority band between 0 and 255, inclusive. If `flags` is set to `MSG_BAND`, then a message is generated in the priority band specified by `band`. Otherwise, if `flags` is set to `MSG_HIPRI`, `band` must be set to 0 and an `M_PCPROTO` message is generated. The call

```
putmsg(fd, ctlp, datp, 0);
```

is equivalent to the call

```
putpmsg(fd, ctlp, datp, 0, MSG_BAND);
```

An application can use `getpmsg(2)` to retrieve a message from a particular priority band:

```
#include <stropts.h>

int getpmsg(int fd, struct strbuf *ctlp,
     struct strbuf *datp, int *bandp, int *flagsp);
```

The integer referenced by `flagsp` can be one of `MSG_HIPRI`, `MSG_BAND`, or `MSG_ANY`, to retrieve a high-priority message, a message from *at least* the band specified by the integer referenced by `bandp`, or any message, respectively. If

MSG_BAND is set, it is possible to obtain a message from a band greater than requested, and it is possible to obtain a high-priority message. (This behavior is analogous to the way getmsg works if the RS_HIPRI flag is not used. Note that the flags are different between the original system calls and their priority band versions.) On return, bandp points to an integer containing the priority band of the message, and flagsp points to an integer indicating the message received was high-priority (MSG_HIPRI) or not (MSG_BAND). The call

```
    int flags = 0;

    getmsg(fd, ctlp, datp, &flags);
```

is equivalent to the call

```
    int flags = MSG_ANY;
    int band = 0;

    getpmsg(fd, ctlp, datp, &band, &flags);
```

In addition to getpmsg and putpmsg, four ioctl commands were added to support priority band processing [see streamio(7) for more details]. Priority bands also became visible in the STREAMS event mechanisms, namely poll and signal generation.

Autopush

Usually when a stream is opened and the device is not already open, no modules are present in the stream. Applications have to push the modules they want explicitly. When the terminal subsystem was ported to STREAMS, this became a problem. In the clist-based implementation, when a terminal device was opened, a line discipline was already associated with the terminal line. Originally, this was not true with STREAMS-based TTY drivers. To avoid having to modify all applications that opened terminal devices, the *autopush* feature was developed.

The autopush feature allows administrators to specify a list of modules to be automatically pushed whenever a device is opened. The autopush(1M) command provides the administrative interface to configure devices for autopush. For a particular device, administrators have the choice of configuring all the minors, a range of minors, or individual minors.

If a device is configured for autopush, when it is opened the stream head will take care of pushing the modules on the stream before returning from the open system call. This way, applications can open terminal devices and automatically have a line discipline associated with the terminal line, maintaining compatibility.

However, there is a drawback to the autopush mechanism. After opening a driver, applications have no way of knowing *a priori* whether or not there are modules on the stream. If it matters to the applications, they must check explicitly for the presence of modules.

Example 3.7.1. This example illustrates how applications can check for the presence of modules on a stream. The function returns the number of modules on the stream on success, or a negative number on failure.

```
#include <sys/types.h>
#include <unistd.h>
#include <stropts.h>

int
nmod(int fd)
{
     int n;

     n = ioctl(fd, I_LIST, NULL);
     return(n - 1);
}
```

The `I_LIST` `ioctl` command is used to return a list of names of all the modules and the driver in the stream. With a `NULL` argument, however, it returns the number of names so that the caller can allocate the necessary space for the list. We use this to see if any modules are pushed on the stream. Since the driver is included in the list, we return one less than the number returned by `I_LIST`. If the `ioctl` fails, then we just return a negative number. The `I_LIST` ioctl should never return 0, so that case is treated as an error as well. □

Summary

We have briefly covered the user-level interface to the STREAMS subsystem in System V. Those interested in more details of how the STREAMS mechanism works can refer to Chapter 9. Each of the next five chapters covers networking-related facilities that are built on top of the STREAMS subsystem.

Exercises

3.1 Write a routine that reads a message that always starts with an indication of the message's size.

3.2 Modify Example 3.4.2 to use nonblocking I/O in conjunction with polling.

3.3 Describe how to determine whether the message at the front of the stream head read queue is a protocol message or a file-descriptor message [hint: see `streamio(7)`].

3.4 Assume processes send you the following message over a pipe:

```
struct msg {
     uid_t   uid;
     pid_t   pid;
     char    text[256];
};
```

Write a routine that reads messages with this format, taking into account the four rules from Section 3.6 for guarding against corrupt and illegitimate data.

Bibliographic Notes

The Streams mechanism, invented by Dennis Ritchie, was introduced in the Eighth Edition Research UNIX System [Ritchie 1984]. STREAMS was first commercially available in UNIX System V Release 3.0 [AT&T 1986]. Presotto and Ritchie [1985, 1990] present Stream pipes, mounted streams, and CONNLD as they appeared in the Ninth Edition Research UNIX System.

Olander, McGrath, and Israel [1986] discuss the work done to provide support for service interfaces in System V STREAMS. Rago [1989] describes changes made to the STREAMS subsystem to provide support for multiple bands of data flow.

To contrast the `clist` mechanism with the STREAMS mechanism that replaced it, refer to Bach [1986].

This page intentionally left blank

Index

Most functions and constants necessary for UNIX System V netw ork programming are listed in this inde x. The "definition of" entries for functions listed here refer to places in the book where the function prototypes can be found. Similar entries are provided to identify structure def initions. References have been omitted to places where common functions, such as `exit`, are used. Page numbers printed in boldface refer to the location in the book where the source code implementing the corresponding function can be found.