Multiplexing 10

Overview of Multiplexing

This chapter describes how STREAMS multiplexing configurations are created and also discusses multiplexing drivers. A STREAMS multiplexer is a driver with multiple Streams connected to it. The primary function of the multiplexing driver is to switch messages among the connected Streams. Multiplexer configurations are created from user level by system calls.

STREAMS-related system calls are used to set up the “plumbing,” or Stream interconnections, for multiplexing drivers. The subset of these calls that allows a user to connect (and disconnect) Streams below a driver is referred to as the multiplexing facility. This type of connection is referred to as a one-to-M, or lower, multiplexer configuration. This configuration must always contain a multiplexing driver, which is recognized by STREAMS as having special characteristics.

Multiple Streams can be connected above a driver by use of `open`(2) calls. This was done for the loop-around driver and for the driver handling multiple minor devices in Chapter 9, “Drivers". There is no difference between the connections to these drivers, only the functions performed by the driver are different. In the multiplexing case, the driver routes data between multiple Streams. In the device driver case, the driver routes data between user processes and associated physical ports. Multiplexing with Streams connected above is referred to as an N-to-1, or upper, multiplexer. STREAMS does not provide any facilities beyond `open`(2) and `close`(2) to connect or disconnect upper Streams for multiplexing.

From the driver's perspective, upper and lower configurations differ only in the way they are initially connected to the driver. The implementation requirements are the same: route the data and handle flow control. All multiplexer drivers require special developer-provided software to perform the multiplexing data routing and to handle flow control. STREAMS does not directly support flow control among multiplexed Streams.

M-to-N multiplexing configurations are implemented by using both of the above mechanisms in a driver.

As discussed in Chapter 9, "Drivers", the multiple Streams that represent minor devices are actually distinct Streams in which the driver keeps track of each Stream attached to it. The STREAMS subsystem does not recognize any relationship between the Streams. The same is true for STREAMS multiplexers of any configuration. The multiplexed Streams are distinct and the driver must be implemented to do most of the work.

In addition to upper and lower multiplexers, more complex configurations can be created by connecting Streams containing multiplexers to other multiplexer drivers. With such a diversity of needs for multiplexers, it is not possible to provide general-purpose multiplexer drivers. Rather, STREAMS provides a general purpose multiplexing facility. The facility allows users to set up the inter-module/driver plumbing to create multiplexer configurations of generally unlimited interconnection.

Building a Multiplexer

This section builds a protocol multiplexer with the multiplexing configuration shown in Figure 10-1. To free users from the need to know about the underlying protocol structure, a user-level daemon process will be built to maintain the multiplexing configuration. Users can then access the transport protocol directly by opening the transport protocol (TP) driver device node.

An internetworking protocol driver (IP) routes data from a single upper Stream to one of two lower Streams. This driver supports two STREAMS connections beneath it. These connections are to two distinct networks; one for the IEEE 802.3 standard via the 802.3 driver, and other to the IEEE 802.4 standard via the 802.4 driver. The TP driver multiplexes upper Streams over a single Stream to the IP driver.

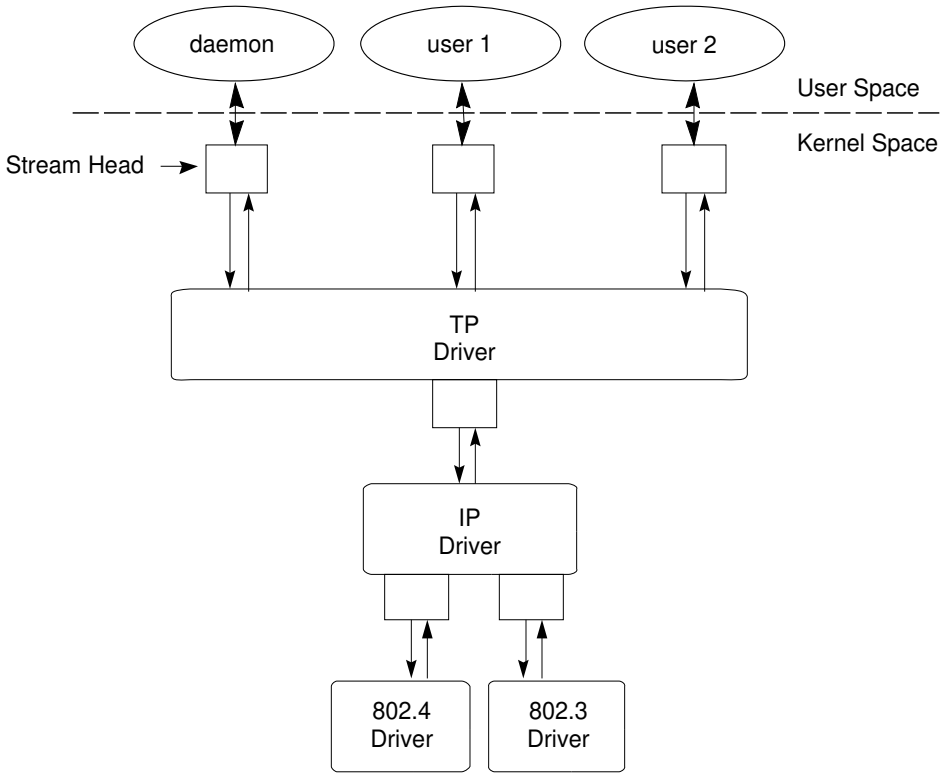

Figure 10-1 MultiplexerProtocol Multiplexer

Code Example 10-1 shows how this daemon process sets up the protocol multiplexer. The necessary declarations and initialization for the daemon program follow:

Code Example 10-1 Protocol Daemon

```
#include <fcntl.h>
#include <stropts.h>
void
main()
{
    int         fd_802_4,
```

Code Example 10-1 Protocol Daemon

```
            fd_802_3,
            fd_ip,
            fd_tp;
    /* daemon-ize this process */

    switch (fork()) {
    case 0:
            break;
    case -1:
            perror("fork failed");
            exit(2);
    default:
            exit(0);
    }
    (void)setsid();
```

This multi-level multiplexed Stream configuration will be built from the bottom up. Therefore, the example begins by first constructing the Internal Protocol (IP) multiplexer. This multiplexing device driver is treated like any other software driver. It owns a node in the Solaris file system and is opened just like any other STREAMS device driver.

The first step is to open the multiplexing driver and the 802.4 driver, thus creating separate Streams above each driver as shown inFigure 10-2. The Stream to the 802.4 driver may now be connected below the multiplexing IP driver using the `I_LINK ioctl` call.

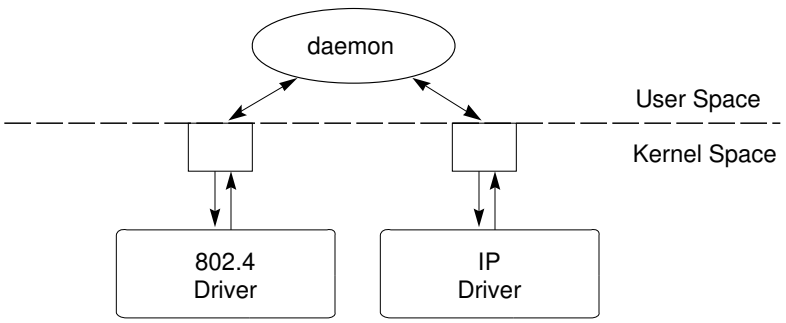

Figure 10-2 Before Link

The sequence of instructions to this point is:

```
if ((fd_802_4 = open("/dev/802_4", O_RDWR)) < 0) {
        perror("open of /dev/802_4 failed");
        exit(1);
}
if ((fd_ip = open("/dev/ip", O_RDWR)) < 0) {
        perror("open of /dev/ip failed");
        exit(2);
}
/* now link 802.4 to underside of IP */
if (ioctl(fd_ip, I_LINK, fd_802_4) < 0) {
        perror("I_LINK ioctl failed");
        exit(3);
}
```

`I_LINK` takes two file descriptors as arguments. The first file descriptor, *fd_ip*, must reference the Stream connected to the multiplexing driver, and the second file descriptor, *fd_802_4*, must reference the Stream to be connected below the multiplexer. Figure 10-3 shows the state of these Streams following the `I_LINK` call. The complete Stream to the 802.4 driver has been connected below the IP driver. The Stream head's queues of the 802.4 driver will be used by the IP driver to manage the lower half of the multiplexer.

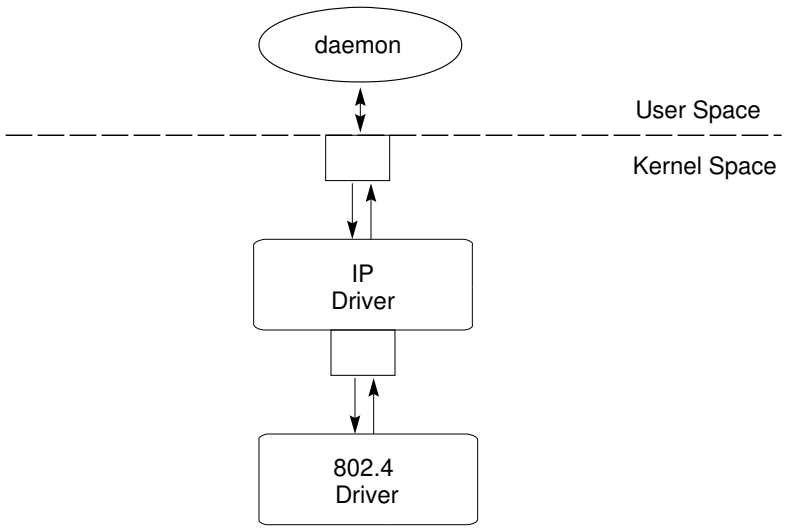

Figure 10-3 IP Multiplexer After First Link

`I_LINK` will return an integer value, called *muxid*, which is used by the multiplexing driver to identify the Stream just connected below it. This *muxid* is ignored in the example, but it is useful for dismantling a multiplexer or routing data through the multiplexer. Its significance is discussed later.

The following sequence of system calls is used to continue building the internetworking protocol multiplexer (IP):

```
if ((fd_802_3 = open("/dev/802_3", O_RDWR)) < 0) {
        perror("open of /dev/802_3 failed");
        exit(4);
}
if (ioctl(fd_ip, I_LINK, fd_802_3) < 0) {
        perror("I_LINK ioctl failed");
        exit(5);
}
```

All links below the IP driver have now been established, giving the configuration in Figure 10-4

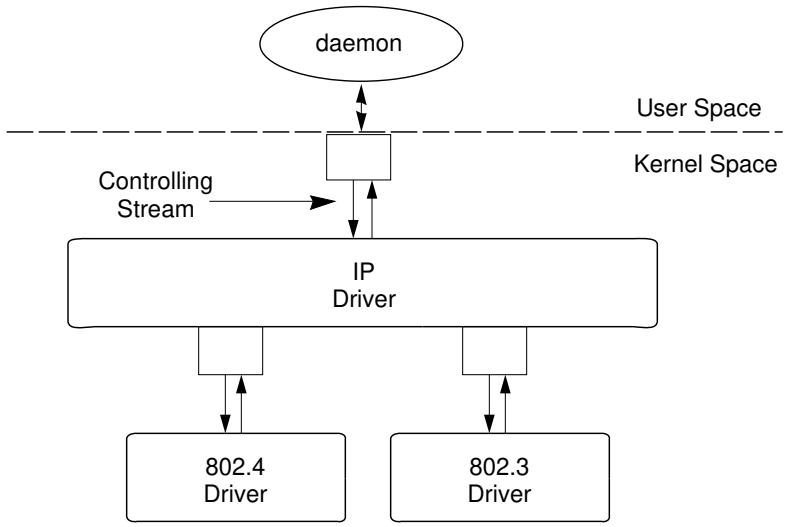

Figure 10-4 IP Multiplexer

The Stream above the multiplexing driver used to establish the lower connections is the controlling Stream and has special significance when dismantling the multiplexing configuration. This will be illustrated later in this chapter. The Stream referenced by *fd_ip* is the controlling Stream for the IP multiplexer.

Note – The order in which the Streams in the multiplexing configuration are opened is unimportant. If it is necessary to have intermediate modules in the Stream between the IP driver and media drivers, these modules must be added to the Streams associated with the media drivers (using `I_PUSH`) before the media drivers are attached below the multiplexer.

The number of Streams that can be linked to a multiplexer is restricted by the design of the particular multiplexer. The manual page describing each driver (typically found in section 7) describes such restrictions. However, only one `I_LINK` operation is allowed for each lower Stream; a single Stream cannot be linked below two multiplexers simultaneously.

Continuing with the example, the IP driver will now be linked below the transport protocol (TP) multiplexing driver. As seen inFigure 10-1, only one link will be supported below the transport driver. This link is formed by the following sequence of system calls:

```
if ((fd_tp = open("/dev/tp", O_RDWR)) < 0) {
        perror("open of /dev/tp failed");
        exit(6);
}
if (ioctl(fd_tp, I_LINK, fd_ip) < 0) {
        perror("I_LINK ioctl failed");
        exit(7);
}
```

The multi-level multiplexing configuration shown in Figure 10-5 has now been created.

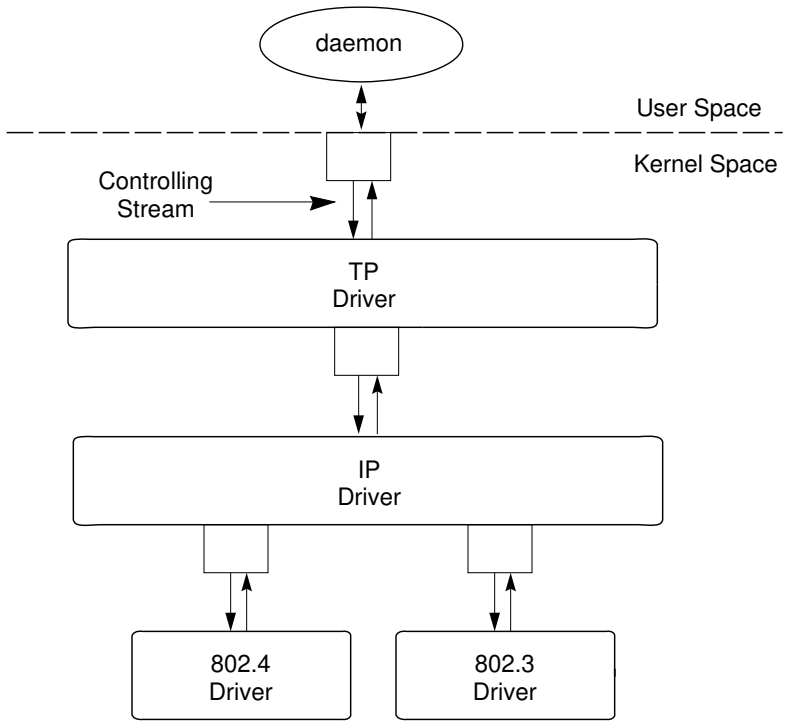

Figure 10-5 TP Multiplexer

Because the controlling Stream of the IP multiplexer has been linked below the TP multiplexer, the controlling Stream for the new multi-level multiplexer configuration is the Stream above the TP multiplexer.

At this point the file descriptors associated with the lower drivers can be closed without affecting the operation of the multiplexer. If these file descriptors are not closed, all subsequent `read`, `write`, `ioctl`, `poll`, `getmsg`, and `putmsg` system calls issued to them will fail. That is because `I_LINK` associates the Stream head of each linked Stream with the multiplexer, so the user may not access that Stream directly for the duration of the link.

The following sequence of system calls completes the daemon example:

```
    close(fd_802_4);
    close(fd_802_3);
    close(fd_ip);
    /* Hold multiplexer open forever or at least til this process
       is terminated by an external UNIX signal */
    pause();
}
```

To summarize,Figure 10-5 shows the multi-level protocol multiplexer. The transport driver supports several simultaneous Streams. These Streams are multiplexed over the single Stream connected to the IP multiplexer. The mechanism for establishing multiple Streams above the transport multiplexer is actually a by-product of the way in which Streams are created between a user process and a driver. By opening different minor devices of a STREAMS driver, separate Streams will be connected to that driver. The driver must be designed with the intelligence to route data from the single lower Stream to the appropriate upper Stream.

The daemon process maintains the multiplexed Stream configuration through an open Stream (the controlling Stream) to the transport driver. Meanwhile, other users can access the services of the transport protocol by opening new Streams to the transport driver; they are freed from the need for any unnecessary knowledge of the underlying protocol configurations and sub-networks that support the transport service.

Multi-level multiplexing configurations should be assembled from the bottom up. That is because the passing of `ioctl`s through the multiplexer is determined by the nature of the multiplexing driver and cannot generally be relied on.

Dismantling a Multiplexer

Streams connected to a multiplexing driver from above with `open`, can be dismantled by closing each Stream with `close`. The mechanism for dismantling Streams that have been linked below a multiplexing driver is less obvious, and is described in the following section.

The I_UNLINK ioctl call is used to disconnect each multiplexer link below a multiplexing driver individually. This command has the form:

```
ioctl(fd, I_UNLINK, muxid);
```

where *fd* is a file descriptor associated with a Stream connected to the multiplexing driver from above, and *muxid* is the identifier that was returned by I_LINK when a driver was linked below the multiplexer. Each lower driver may be disconnected individually in this way, or a special *muxid* value of MUXID_ALL may be used to disconnect all drivers from the multiplexer simultaneously.

In the multiplexing daemon program, the multiplexer is never explicitly dismantled. That is because all links associated with a multiplexing driver are automatically dismantled when the controlling Stream associated with that multiplexer is closed. Because the controlling Stream is open to a driver, only the final call of close for that Stream will close it. In this case, the daemon is the only process that has opened the controlling Stream, so the multiplexing configuration will be dismantled when the daemon exits.

For the automatic dismantling mechanism to work in the multi-level, multiplexed Stream configuration, the controlling Stream for each multiplexer at each level must be linked under the next higher level multiplexer. In the example, the controlling Stream for the IP driver was linked under the TP driver. This resulted in a single controlling Stream for the full, multi-level configuration. Because the multiplexing program relied on closing the controlling Stream to dismantle the multiplexed Stream configuration instead of using explicit I_UNLINK calls, the *muxid* values returned by I_LINK could be ignored.

An important side effect of automatic dismantling on the close is that it is not possible for a process to build a multiplexing configuration with I_LINK and then exit. That is because exit(2) will close all files associated with the process, including the controlling Stream. To keep the configuration intact, the process must exist for the life of that multiplexer. That is the motivation for implementing the example as a daemon process.

However, if the process uses persistent links via the I_PLINK ioctl call, the multiplexer configuration would remain intact after the process exits. “Persistent Links" are described later in this chapter.

Routing Data Through a Multiplexer

As demonstrated, STREAMS provides a mechanism for building multiplexed Stream configurations. However, the criteria by which a multiplexer routes data is driver dependent. For example, the protocol multiplexer might use address information found in a protocol header to determine over which sub-network data should be routed. It is the multiplexing driver's responsibility to define its routing criteria.

One routing option available to the multiplexer is to use the *muxid* value to determine to which Stream data should be routed (remember that each multiplexer link is associated with a *muxid*). `I_LINK` passes the *muxid* value to the driver and returns this value to the user. The driver can therefore specify that the *muxid* value must accompany data routed through it. For example, if a multiplexer routed data from a single upper Stream to one of several lower Streams (as did the IP driver), the multiplexer could require the user to insert the *muxid* of the desired lower Stream into the first four bytes of each message passed to it. The driver could then match the *muxid* in each message with the *muxid* of each lower Stream, and route the data accordingly.

Connecting / Disconnecting Lower Streams

Multiple Streams are created above a driver/multiplexer by use of the `open` system call on either different minor devices, or on a cloneable device file. Note that any driver that handles more than one minor device is considered an upper multiplexer.

To connect Streams below a multiplexer requires additional software within the multiplexer. The main difference between STREAMS lower multiplexers and STREAMS device drivers are that multiplexers are pseudo-devices and that multiplexers have two additional `qinit` structures, pointed to by fields in the `streamtab` structure: the *lower half read side* `qinit` and the *lower half write side* `qinit`.

The multiplexer is conceptually divided into two parts: the lower half (bottom) and the upper half (top). The multiplexer `queue` structures that have been allocated when the multiplexer was opened, use the usual `qinit` entries from the multiplexer's `streamtab`. This is the same as any open of the STREAMS device. When a lower Stream is linked beneath the multiplexer, the `qinit` structures at the Stream head are substituted by the bottom half `qinit` structures of the multiplexers. Once the linkage is made, the multiplexer

switches messages between upper and lower Streams. When messages reach the top of the lower Stream, they are handled by `put` and `service` routines specified in the bottom half of the multiplexer.

Connecting Lower Streams

A lower multiplexer is connected as follows: the initial `open` to a multiplexing driver creates a Stream, as in any other driver. `open` uses the first two `streamtab` structure entries to create the driver queues. At this point, the only distinguishing characteristic of this Stream are non-NULL entries in the `streamtab` *st_muxrinit* and *st_muxwinit* fields.

These fields are ignored by `open` (see the rightmost Stream in Figure 10-6). Any other Stream subsequently opened to this driver will have the same `streamtab` and thereby the same mux fields.

Next, another file is opened to create a (soon-to-be) lower Stream. The driver for the lower Stream is typically a device driver (see the leftmost Stream in Figure 10-6). This Stream has no distinguishing characteristics. It can include any driver compatible with the multiplexer. Any modules required on the lower Stream must be pushed onto it now.

Next, this lower Stream is connected below the multiplexing driver with an `I_LINK ioctl` call [see `streamio(7I)`]. The Stream head points to the Stream head routines as its procedures (known via its `queue`). An `I_LINK` to the upper Stream, referencing the lower Stream, causes STREAMS to modify the contents of the Stream head's queues in the lower Stream. The pointers to the Stream head routines, and other values, in the Stream head's queues are replaced with those contained in the mux fields of the multiplexing driver's `streamtab`. Changing the Stream head routines on the lower Stream means that all subsequent messages sent upstream by the lower Stream's driver will, ultimately, be passed to the `put` procedure designated in *st_muxrinit*, the multiplexing driver. The `I_LINK` also establishes this upper Stream as the control Stream for this lower Stream. STREAMS remembers the relationship between these two Streams until the upper Stream is closed, or the lower Stream is unlinked.

Finally, the Stream head sends an `M_IOCTL` message with *ioc_cmd* set to `I_LINK` to the multiplexing driver. The `M_DATA` part of the `M_IOCTL` contains a `linkblk` structure. The multiplexing driver stores information from the

linkblk structure in private storage and returns an M_IOCACK message (acknowledgment). *l_index* is returned to the process requesting the I_LINK. This value can be used later by the process to disconnect this Stream.

An I_LINK is required for each lower Stream connected to the driver. Additional upper Streams can be connected to the multiplexing driver by open calls. Any message type can be sent from a lower Stream to user processes along any of the upper Streams. The upper Streams provide the only interface between the user processes and the multiplexer.

Note that no direct data structure linkage is established for the linked Streams. The read queue's *q_next* will be NULL and the write queue's *q_next* will point to the first entity on the lower Stream. Messages flowing upstream from a lower driver (a device driver or another multiplexer) will enter the multiplexing driver put procedure with *l_qbot* as the queue value. The multiplexing driver has to route the messages to the appropriate upper (or lower) Stream. Similarly, a message coming downstream from user space on any upper Stream has to be processed and routed, if required, by the driver.

Also note that the lower Stream (see the headers and file descriptors in Figure 10-7) is no longer accessible from user space. This causes all system calls to the lower Stream to return EINVAL, with the exception of close. This is why all modules have to be in place before the lower Stream is linked to the multiplexing driver.

Finally, note that the absence of direct linkage between the upper and lower Streams means that STREAMS flow control has to be handled by special code in the multiplexing driver. The flow control mechanism cannot see across the driver.

In general, multiplexing drivers should be implemented so that new Streams can be dynamically connected to (and existing Streams disconnected from) the driver without interfering with its ongoing operation. The number of Streams that can be connected to a multiplexer is implementation dependent.

Disconnecting Lower Streams

Dismantling a lower multiplexer is accomplished by disconnecting (unlinking) the lower Streams. Unlinking can be initiated in three ways:

- An I_UNLINK ioctl referencing a specific Stream
- An I_UNLINK indicating all lower Streams

- The last `close` of the control Stream

As in the link, an unlink sends a `linkblk` structure to the driver in an `M_IOCTL` message. The `I_UNLINK` call, which unlinks a single Stream, uses the *l_index* value returned in the `I_LINK` to specify the lower Stream to be unlinked. The latter two calls must designate a file corresponding to a control Stream, which causes all the lower Streams that were previously linked by this control Stream to be unlinked. However, the driver sees a series of individual unlinks.

If no open references exist for a lower Stream, a subsequent unlink will automatically close the Stream. Otherwise, the lower Stream must be closed by `close` following the unlink. STREAMS will automatically dismantle all cascaded multiplexers (below other multiplexing Streams) if their controlling Stream is closed. An `I_UNLINK` will leave lower, cascaded multiplexing Streams intact unless the Stream file descriptor was previously closed.

Multiplexer Construction Example

This section describes an example of multiplexer construction and usage. Figure 10-6 shows the Streams before their connection to create the multiplexing configuration of Figure 10-7. Multiple upper and lower Streams interface to the multiplexer driver. The user processes of Figure 10-7 are not shown in Figure 10-6.

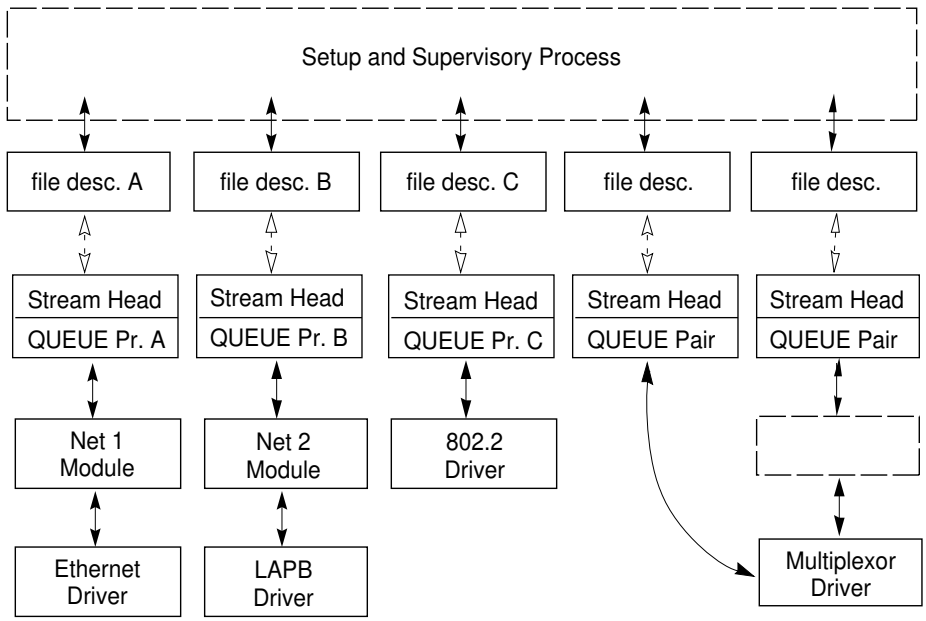

Figure 10-6 Internet Multiplexer Before Connecting

The Ethernet, LAPB and IEEE 802.2 device drivers terminate links to other nodes. The multiplexer driver is an Internet Protocol (IP) multiplexer that switches data among the various nodes or sends data upstream to a user(s) in the system. The net modules would typically provide a convergence function that matches the multiplexer driver and device driver interface.

Figure 10-6 shows only a portion of the full, larger Stream. In the dotted rectangle above the multiplexer driver, there generally would be an upper transmission control protocol (TCP) multiplexer, additional modules and, possibly, additional multiplexers in the Stream. Multiplexers could also be cascaded below the IP driver if the device drivers were replaced by multiplexer drivers.

Figure 10-7 shows that the file descriptors for the lower device driver Streams are left dangling. The primary purpose in creating these Streams was to provide parts for the multiplexer. Those not used for control and not required for error recovery have no further function. (For example, they could be

reconnected through a `I_UNLINK` and `I_LINK sequence`) These lower Streams can be closed to free the file descriptor without any effect on the multiplexer.

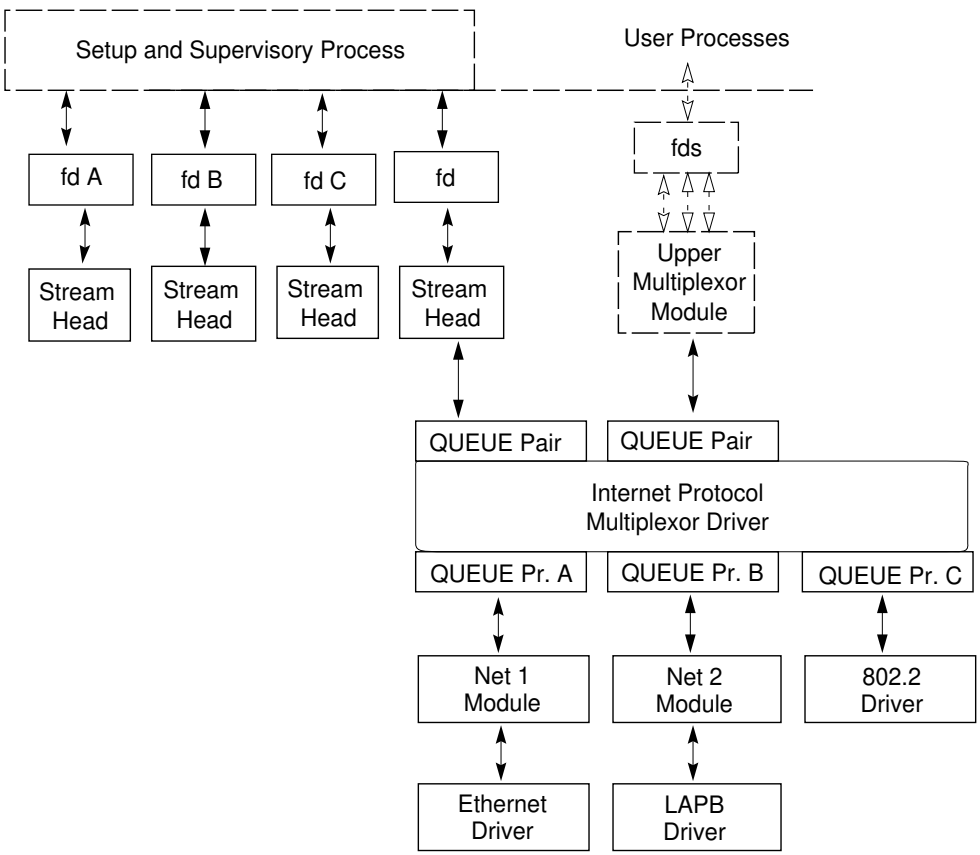

Figure 10-7 Internet Multiplexer After Connecting

Streams A, B, and C are opened by the process, and modules are pushed as needed. Two upper Streams are opened to the IP multiplexer. The rightmost Stream represents multiple Streams, each connected to a process using the network. The Stream second from the right provides a direct path to the multiplexer for supervisory functions. It is the control Stream, leading to a

process which sets up and supervises this configuration. It is always directly connected to the IP driver. Although not shown, modules can be pushed on the control Stream.

After the Streams are opened, the supervisory process typically transfers routing information to the IP drivers (and any other multiplexers above the IP), and initializes the links. As each link becomes operational, its Stream is connected below the IP driver. If a more complex multiplexing configuration is required, the IP multiplexer Stream with all its connected links can be connected below another multiplexer driver.

Multiplexing Driver

This section contains an example of a multiplexing driver that implements an N-to-1 configuration. This configuration might be used for terminal windows, where each transmission to or from the terminal identifies the window. This resembles a typical device driver, with two differences: the device handling functions are performed by a separate driver, connected as a lower Stream, and the device information (that is, relevant user process) is contained in the input data rather than in an interrupt call.

Each upper Stream is created by `open`(2). A single lower Stream is opened and then it is linked by use of the multiplexing facility. This lower Stream might connect to the `tty` driver. The implementation of this example is a foundation for an M-to-N multiplexer.

As in the loop-around driver (Chapter 9, “Drivers"), flow control requires the use of standard and special code, since physical connectivity among the Streams is broken at the driver. Different approaches are used for flow control on the lower Stream, for messages coming upstream from the device driver, and on the upper Streams, for messages coming downstream from the user processes.

Note – The code presented here for the multiplexing driver represents a single threaded, uni-processor implementation. Multi-processor and multithreading issues such as locking for data corruption and to prevent race conditions are not discussed. See Chapter 13, “Multi-Threaded STREAMS" for details.

Code Example 10-2 is of multiplexer declarations:

Code Example 10-2 Multiplexer Declarations

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/errno.h>
#include <sys/cred.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

static int muxopen (queue_t*, dev_t*, int, int, cred_t*);
static int muxclose (queue_t*, int, cred_t*);
static int muxuwput (queue_t*, mblk_t*);
static int muxlwsrv (queue_t*);
static int muxlrput (queue_t*, mblk_t*);
static int muxuwsrv (queue_t*);

static struct module_info info = {
    0xaabb, "mux", 0, INFPSZ, 512, 128 };

static struct qinit urinit = {              /* upper read */
    NULL, NULL, muxopen, muxclose, NULL, &info, NULL };

static struct qinit uwinit = {              /* upper write */
    muxuwput, muxuwsrv, NULL, NULL, NULL, &info, NULL };

static struct qinit lrinit = { /* lower read */
    muxlrput, NULL, NULL, NULL, NULL, &info, NULL };

static struct qinit lwinit = { /* lower write */
    NULL, muxlwsrv, NULL, NULL, NULL, &info, NULL };

struct streamtab muxinfo = {
    &urinit, &uwinit, &lrinit, &lwinit };

struct mux {
    queue_t *qptr;            /* back pointer to read queue */
    int     bufcid;           /* bufcall return value */
};
extern struct mux mux_mux[];
extern int mux_cnt;      /* max number of muxes */
```

Code Example 10-2 Multiplexer Declarations

```
static queue_t *muxbot;          /* linked lower queue */
static int muxerr;               /* set if error of hangup on
lower strm */
```

The four `streamtab` entries correspond to the upper read, upper write, lower read, and lower write `qinit` structures. The multiplexing `qinit` structures replace those in each (in this case there is only one) lower Stream head after the `I_LINK` has concluded successfully. In a multiplexing configuration, the processing performed by the multiplexing driver can be partitioned between the upper and lower queues. There must be an upper Stream write `put` procedure and lower Stream read `put` procedure. If the queue procedures of the opposite upper/lower queue are not needed, the queue can be skipped , and the message put to the following queue.

In the example, the upper read side procedures are not used. The lower Stream read queue `put` procedure transfers the message directly to the read queue upstream from the multiplexer. There is no lower write `put` procedure because the upper write `put` procedure directly feeds the lower write queue downstream from the multiplexer.

The driver uses a private data structure, *mux*. *mux_mux*[*dev*] points back to the opened upper read queue. This is used to route messages coming upstream from the driver to the appropriate upper queue. It is also used to find a free major/minor device for a `CLONEOPEN` driver open case.

Code Example 10-3,the upper queue open, contains the canonical driver open code:

Code Example 10-3 Upper Queue Open

```
static int
muxopen(queue_t *q, dev_t *devp, int flag,
                int sflag, cred_t *credp)
{
    struct mux *mux;
    minor_t device;

    if (q->q_ptr)
```

Code Example 10-3 Upper Queue Open

```
            return(EBUSY);

    if (sflag == CLONEOPEN) {
            for (device = 0; device < mux_cnt; device++)
                if (mux_mux[device].qptr == 0)
                        break;

            *devp=makedevice(getmajor(*devp), device);
    }
    else {
            device = getminor(*devp);
            if (device >= mux_cnt)
                return ENXIO;
    }

    mux = &mux_mux[device];
    mux->qptr = q;
    q->q_ptr = (char *) mux;
    WR(q)->q_ptr = (char *) mux;
    qprocson(q);
    return (0);
}
```

muxopen checks for a clone or ordinary open call. It initializes *q_ptr* to point at the *mux_mux*[] structure.

The core multiplexer processing is the following: downstream data written to an upper Stream is queued on the corresponding upper write message queue if the lower Stream is flow controlled. This allows flow control to propagate towards the Stream head for each upper Stream. A lower write `service` procedure, rather than a write `put` procedure, is used so that flow control, coming up from the driver below, may be handled.

On the lower read side, data coming up the lower Stream are passed to the lower read `put` procedure. The procedure routes the data to an upper Stream based on the first byte of the message. This byte holds the minor device number of an upper Stream. The `put` procedure handles flow control by testing the upper Stream at the first upper read queue beyond the driver. That is, the `put` procedure treats the Stream component above the driver as the next queue.

Upper Write Put Procedure

muxuwput, the upper queue write `put` procedure, traps `ioctls`, in particular `I_LINK` and `I_UNLINK`:

```
static int
/*
* This is our callback routine usesd by bufcall() to inform us
*when buffers become available
*/
static void mux_qenable(long ql)
{
        queue_t *q = (queue_t *ql);
        struct mux *mux;

        mux = (struct mux *)(q->q_ptr);
        mux->bufcid = 0;
        qenable(q);
}
muxuwput(queue_t *q, mblk_t *mp)
{
    struct mux *mux;

    mux = (struct mux *)q->q_ptr;
    switch (mp->b_datap->db_type) {
    case M_IOCTL: {
            struct iocblk *iocp;
            struct linkblk *linkp;
            /*
             * ioctl. Only channel 0 can do ioctls. Two
             * calls are recognized: LINK, and UNLINK
             */
            if (mux != mux_mux)
                goto iocnak;

            iocp = (struct iocblk *) mp->b_rptr;
            switch (iocp->ioc_cmd) {
            case I_LINK:
                /*
                 *Link. The data contains a linkblk structure
                 *Remember the bottom queue in muxbot.
                 */
                if (muxbot != NULL)
```

```
                    goto iocnak;

                linkp=(struct linkblk *) mp->b_cont->b_rptr;
                muxbot = linkp->l_qbot;
                muxerr = 0;

                mp->b_datap->db_type = M_IOCACK;
                iocp->ioc_count = 0;
                qreply(q, mp);
                break;
            case I_UNLINK:
                /*
                 * Unlink. The data contains a linkblk struct.
                 * Should not fail an unlink. Null out muxbot.
                 */
                linkp=(struct linkblk *) mp->b_cont->b_rptr;
                muxbot = NULL;
                mp->b_datap->db_type = M_IOCACK;
                iocp->ioc_count = 0;
                qreply(q, mp);
                break;
            default:
            iocnak:
                /* fail ioctl */
                mp->b_datap->db_type = M_IOCNAK;
                qreply(q, mp);
            }
            break;
    }
    case M_FLUSH:
            if (*mp->b_rptr & FLUSHW)
                flushq(q, FLUSHDATA);
            if (*mp->b_rptr & FLUSHR) {
                *mp->b_rptr &= ~FLUSHW;
                qreply(q, mp);
            } else
                freemsg(mp);
            break;
```

```
    case M_DATA:{
             */
             * Data. If we have no bottom queue --> fail
             * Otherwise, queue the data and invoke the lower
             * service procedure.
        mblk_t *bp;
            if (muxerr || muxbot == NULL)
                goto bad;
        if ((bp = allocb(1, BPRI_MED)) == NULL) {
                putbq(q, mp);
                mux->bufcid = bufcall(1, BPRI_MED,
                    mux_qenable, (long)q);
                break;
            }
            *bp->b_wptr++ = (struct mux *)q->ptr - mux_mux;
            bp->b_cont = mp;
            putq(q, bp);
            break;
}
    default:
    bad:
            /*
             * Send an error message upstream.
             */
            mp->b_datap->db_type = M_ERROR;
            mp->b_rptr = mp->b_wptr = mp->b_datap->db_base;
            *mp->b_wptr++ = EINVAL;
            qreply(q, mp);
    }
}
```

First, there is a check to enforce that the Stream associated with minor device 0 will be the single, controlling Stream. The ioctls are only accepted on this Stream. As described previously, a controlling Stream is the one that issues the `I_LINK`. Having a single control Stream is a recommended practice. `I_LINK` and `I_UNLINK` include a linkblk structure containing:

l_qtop
The upper write queue from which the `ioctl` is coming. It always equals *q* for an `I_LINK`, and NULL for `I_PLINK`.

l_qbot
The new lower write queue. It is the former Stream head write queue. It is of most interest since that is where the multiplexer gets and puts its data.

l_index
A unique (system-wide) identifier for the link. It can be used for routing or during selective unlinks. Since the example only supports a single link, *l_index* is not used.

For `I_LINK`, *l_qbot* is saved in *muxbot* and a positive acknowledgment is generated. From this point on, until an `I_UNLINK` occurs, data from upper queues will be routed through *muxbot*. Note that when an `I_LINK`, is received, the lower Stream has already been connected. This allows the driver to send messages downstream to perform any initialization functions. Returning an `M_IOCNAK` message (negative acknowledgment) in response to an `I_LINK` will cause the lower Stream to be disconnected.

The `I_UNLINK` handling code nulls out *muxbot* and generates a positive acknowledgment. A negative acknowledgment should not be returned to an `I_UNLINK`. The Stream head assures that the lower Stream is connected to a multiplexer before sending an `I_UNLINK` `M_IOCTL`.

Drivers can handle the persistent link requests—`I_PLINK` and `I_PUNLINK` ioctls (see page 240) in the same manner, except that *l_qtop* in the *linkblk* structure passed to the put routine will be `NULL` instead of identifying the controlling Stream.

muxuwput handles `M_FLUSH` messages as a normal driver would, except that there are no messages queued on the upper read queue, so there is no need to call flushq if `FLUSHR` is set.

`M_DATA` messages are not placed on the lower write message queue. They are queued on the upper write message queue. When flow control subsides on the lower Stream, the lower `service` procedure, *muxlwsrv*, is scheduled to start output. This is similar to starting output on a device driver.

Upper Write Service Procedure

The following example shows the code for the upper multiplexer write `service` procedure:

```
static int muxuwsrv(queue_t *q)
{
    mblk_t *mp;
    struct mux *muxp;
    muxp = (struct mux *)q->q_ptr;

    if (!muxbot) {
            flushq(q, FLUSHALL);
            return (0);
    }
    if (muxerr) {
            flushq(q, FLUSHALL);
            return (0);
    }
    while (mp = getq(q)) {
            if (canputnext(muxbot))
                putnext(muxbot, mp);
            else {
                putbq(q, mp);
                return(0);
            }
    }
    return (0);
}
```

As long as there is a Stream still linked under the multiplexer and there are no errors, the `service` procedure will take a message off the queue and send it downstream, if flow control allows.

Lower Write Service Procedure

muxlwsrv, the lower (linked) queue write `service` procedure is scheduled as a result of flow control subsiding downstream (it is back-enabled).

```
static int muxlwsrv(queue_t *q)
{
    int i;

    for (i = 0; i < mux_cnt; i++)
            if (mux_mux[i].qptr && mux_mux[i].qptr->q_first)
                qenable(mux_mux[i].qptr);
    return (0);
}
```

muxlwsrv steps through all possible upper queues. If a queue is active and there are messages on the queue, then its upper write `service` procedure is enabled via `qenable`().

Lower Read Put Procedure

The lower (linked) queue read `put` procedure is:

```
static int
muxlrput(queue_t *q, mblk_t *mp)
{
    queue_t *uq;
    int device;

    if(muxerr) {
            freemsg(mp);
            return (0);
    }

    switch(mp->b_datap->db_type) {
    case M_FLUSH:
            /*
             * Flush queues. NOTE: sense of tests is reversed
             * since we are acting like a "stream head"
             */
            if (*mp->b_rptr & FLUSHW) {
```

```
                *mp->b_rptr &= ~FLUSHR;
                qreply(q, mp);
            } else
                freemsg(mp);
            break;
    case M_ERROR:
    case M_HANGUP:
            muxerr = 1;
            freemsg(mp);
            break;
    case M_DATA:
            /*
             * Route message. First byte indicates
             * device to send to. No flow control.
             *
             * Extract and delete device number. If the
             * leading block is now empty and more blocks
             * follow, strip the leading block.
             */
            device = *mp->b_rptr++;

            /* Sanity check. Device must be in range */
            if (device < 0 || device >= mux_cnt) {
                freemsg(mp);
                break;
            }
            /*
             * If upper stream is open and not backed up,
             * send the message there, otherwise discard it.
             */
            uq = mux_mux[device].qptr;
            if (uq != NULL && canputnext(uq))
                putnext(uq, mp);
            else
                freemsg(mp);
            break;
    default:
            freemsg(mp);
    }
    return (0);
}
```

muxlrput receives messages from the linked Stream. In this case, it is acting as

a Stream head. It handles M_FLUSH messages. Note the code is reversed from that of a driver, handling M_FLUSH messages from upstream. There is no need to flush the read queue because no data is ever placed in it.

muxlrput also handles M_ERROR and M_HANGUP messages. If one is received, it locks-up the upper Streams by setting *muxerr*.

M_DATA messages are routed by looking at the first data byte of the message. This byte contains the minor device of the upper Stream. Several sanity checks are made:

- Check whether the device is in range
- Check whether the upper Stream is open
- Check whether the upper Stream is not full

This multiplexer does not support flow control on the read side. It is merely a router. If it passes all sanity checks, the message is put to the proper upper queue. Otherwise, the message is discarded.

The upper Stream close routine simply clears the mux entry so this queue will no longer be found. Outstanding bufcalls are not cleared.

```
/*
* Upper queue close
*/
static int
muxclose(queue_t *q, int flag, cred_t *credp)
{
    struct mux *mux;

    mux = (struct mux *) q->q_ptr;
    qprocsoff(q);
    if (mux->bufcid != 0)
        unbufcall(mux->bufcid);
    mux->bufcid = 0;
    mux->ptr = NULL;
    q->q_ptr = NULL;
    WR(q)->q_ptr = NULL;
    return(0);
}
```

10

Persistent Links

With `I_LINK` and `I_UNLINK` `ioctl`s the file descriptor associated with the Stream above the multiplexer used to set up the lower multiplexer connections must remain open for the duration of the configuration. Closing the file descriptor associated with the controlling Stream will dismantle the whole multiplexing configuration. Some applications may not want to keep a process running merely to hold the multiplexer configuration together. Therefore, "free-standing" links below a multiplexer are needed. A persistent link is such a link. It is similar to a STREAMS multiplexer link, except that a process is not needed to hold the links together. After the multiplexer has been set up, the process may close all file descriptors and exit, and the multiplexer will remain intact.

Two `ioctl`s, `I_PLINK` and `I_PUNLINK`, are used to create and remove persistent links that are associated with the Stream above the multiplexer. `close`(2) and `I_UNLINK` are not able to disconnect the persistent links (see `strconf`(1) and `strchg`(1)).

The format of `I_PLINK` is:

```
ioctl(fd0, I_PLINK, fd1)
```

The first file descriptor, *fd0*, must reference the Stream connected to the multiplexing driver and the second file descriptor, *fd1*, must reference the Stream to be connected below the multiplexer. The persistent link can be created in the following way:

```
upper_stream_fd = open("/dev/mux", O_RDWR);
lower_stream_fd = open("/dev/driver", O_RDWR);
muxid = ioctl(upper_stream_fd, I_PLINK, lower_stream_fd);
/*
 * save muxid in a file
 */
exit(0);
```

Figure 10-8 shows how `open`(2) establishes a Stream between the device and the Stream head.

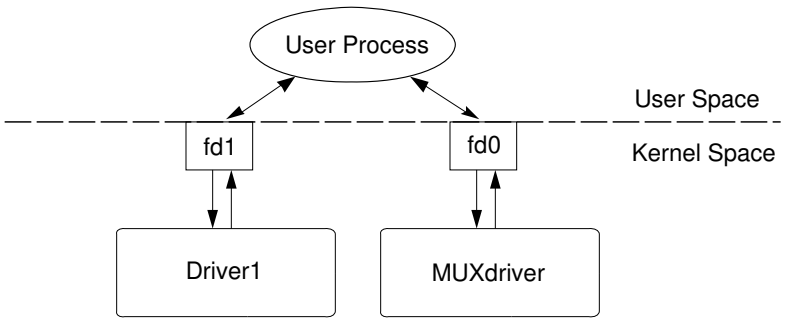

Figure 10-8 `open`() of MUXdriver and Driver1

The persistent link can still exist even if the file descriptor associated with the upper Stream to the multiplexing driver is closed. The `I_PLINK` `ioctl` returns an integer value, *muxid*, that can be used for dismantling the multiplexing configuration. If the process that created the persistent link still exists, it may pass the *muxid* value to some other process to dismantle the link, if the dismantling is desired, or it can leave the *muxid* value in a file so that other processes may find it later.Figure 10-9 shows a multiplexer after `I_PLINK`.

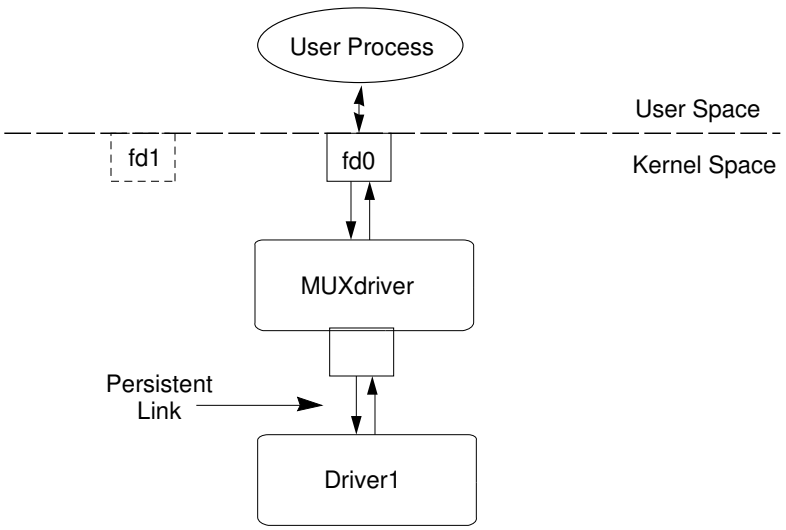

Figure 10-9 Multiplexer After `I_PLINK`

Several users can open the MUXdriver and send data to the Driver1 since the persistent link to the Driver1 remains intact. This is shown in the Figure 10-10.

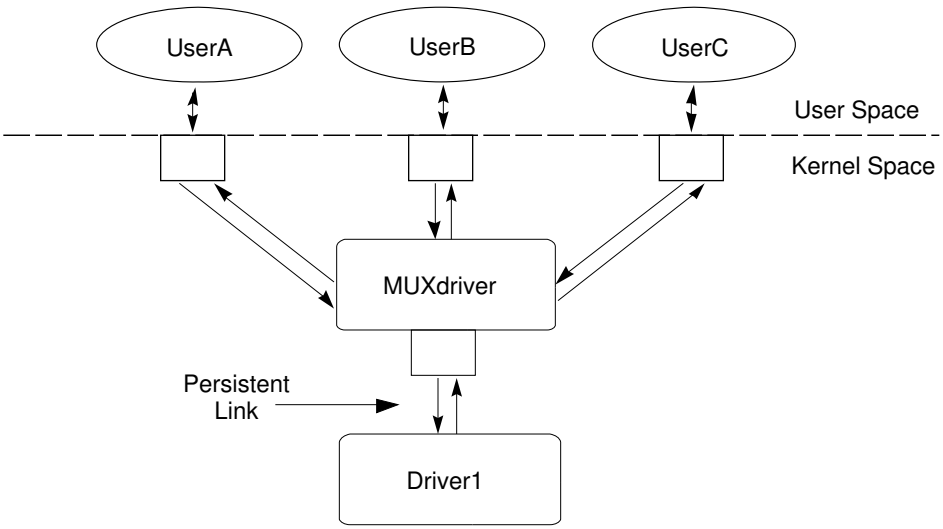

Figure 10-10 Other Users Opening a MUXdriver

The `ioctl I_PUNLINK` is used for dismantling the persistent link. Its format is:

```
ioctl(fd0, I_PUNLINK, muxid)
```

where the `fd0` is the file descriptor associated with Stream connected to the multiplexing driver from above. The `muxid` is returned by the `ioctl I_PLINK` for the Stream that was connected below the multiplexer. The `I_PUNLINK` removes the persistent link between the multiplexer referenced by the `fd0` and the Stream to the driver designated by the `muxid`. Each of the bottom persistent links can be disconnected individually. An `I_PUNLINK ioctl` with the `muxid` value of `MUXID_ALL` will remove all persistent links below the multiplexing driver referenced by the `fd0`.

The following code example shows how to dismantle the previously given configuration:

```
fd = open("/dev/mux", O_RDWR);
/*
 * retrieve muxid from the file
 */
ioctl(fd, I_PUNLINK, muxid);
exit(0);
```

The use of the `ioctls` `I_PLINK` and `I_PUNLINK` should not be mixed with the `I_LINK` and `I_UNLINK`. Any attempt to unlink a regular link via the `I_PUNLINK` or to unlink a persistent link via the `I_UNLINK` `ioctl` will cause the *errno* value of `EINVAL` to be returned.

Since multi-level multiplexing configurations are allowed in STREAMS, it is possible to have a situation where persistent links exist below a multiplexer whose Stream is connected to the above multiplexer by regular links. Closing the file descriptor associated with the controlling Stream will remove the regular link but not the persistent links below it. On the other hand, regular links are allowed to exist below a multiplexer whose Stream is connected to the above multiplexer via persistent links. In this case, the regular links will be removed if the persistent link above is removed and no other references to the lower Streams exist.

The construction of cycles is not allowed when creating links. A cycle could be constructed by creating a persistent link of multiplexer 2 below multiplexer 1 and then closing the controlling file descriptor associated with the multiplexer 2 and reopening it again and then linking the multiplexer 1 below the multiplexer 2. This is not allowed. The operating system prevents a multiplexer configuration from containing a cycle to ensure that messages can not be routed infinitely, thus creating an infinite loop or overflowing the kernel stack.

Design Guidelines

The following lists general multiplexer design guidelines:

- The upper half of the multiplexer acts like the end of the upper Stream. The lower half of the multiplexer acts like the head of the lower Stream. `Service` procedures are used for flow control.

- Message routing is based on multiplexer specific criteria.
- When one Stream is being fed by many Streams, flow control may have to take place. Then all feeding Streams on the other end of the multiplexer will have to be enabled when the flow control is relieved.
- When one Stream is feeding many Streams, flow control may also have to take place. Be careful not to starve other Streams when one becomes flow controlled.

10