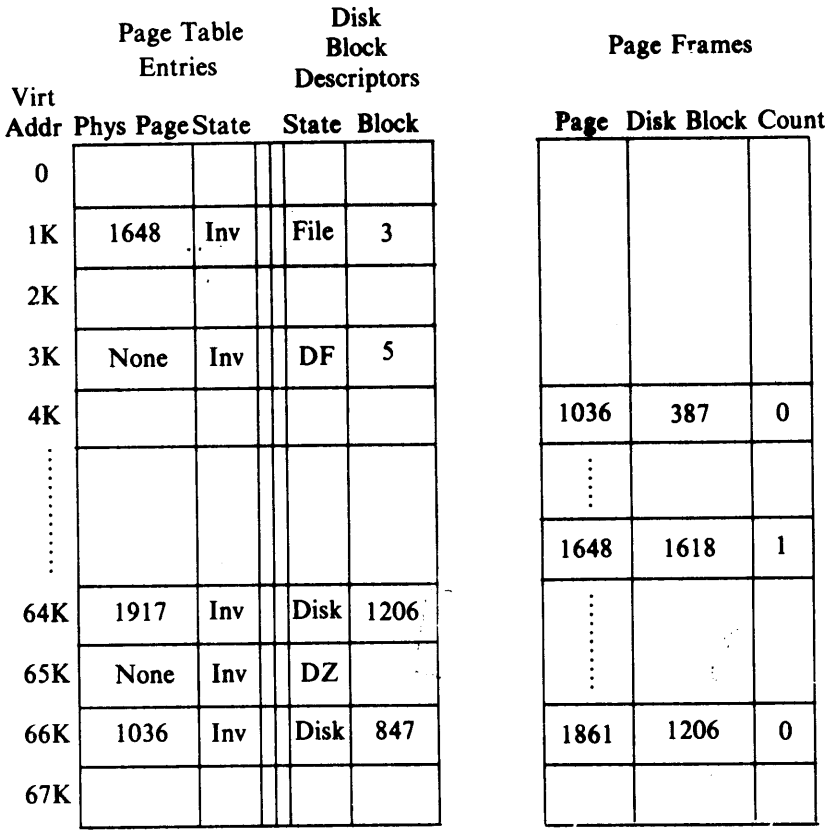


Figure 9.22. Occurrence of a Validity Fault

a process faults when accessing virtual address 64K in Figure 9.22. Searching the page cache, the kernel finds that page frame 1861 is associated with disk block 1206, as is the disk block descriptor. It resets the page table entry for virtual address 64K to point to page 1861, sets the *valid* bit, and returns. The disk block number thus associates a page table entry with a pfdata table entry, explaining why both tables save it.

Similarly, the fault handler does not have to read the page into memory if another process had faulted on the same page but had not completely read it in yet. The fault handler finds the region containing the page table entry locked by another instance of the fault handler. It sleeps until the other instance of the fault handler completes, finds the page now valid, and returns. Figure 9.24 depicts such a scenario.

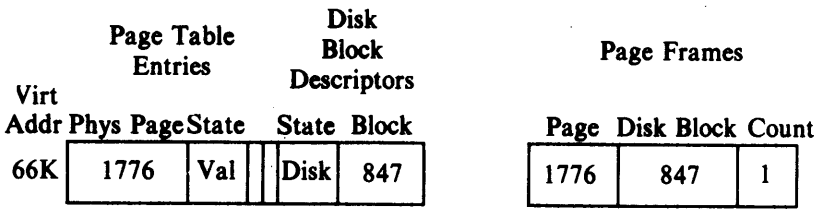


Figure 9.23. After Swapping Page into Memory

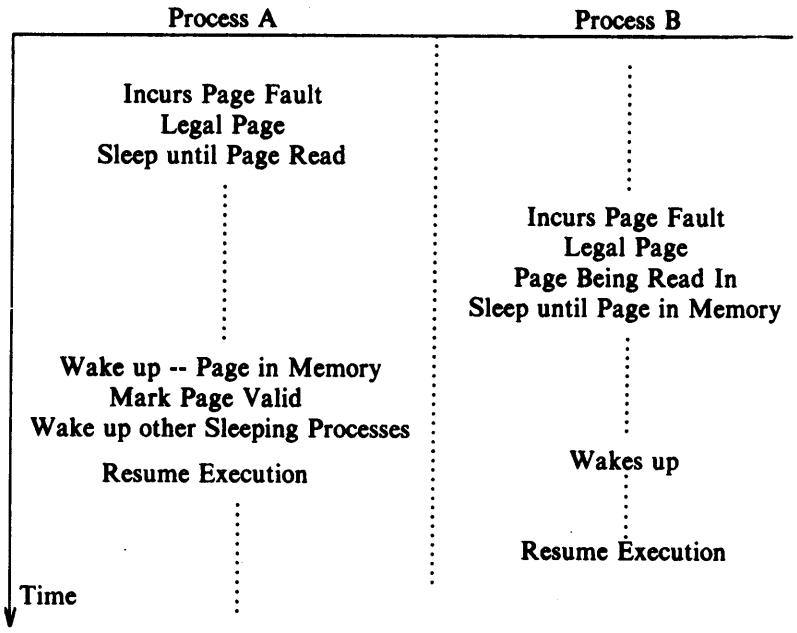


Figure 9.24. Double Fault on a Page

If a copy of the page does not exist on a swap device but is in the original executable file (case 3), the kernel reads the page from the original file. The fault handler examines the disk block descriptor, finds the logical block number in the file that contains the page, and finds the inode associated with the region table entry. It uses the logical block number as an offset into the array of disk block numbers attached to the inode during *exec*. Knowing the disk block number, it reads the page into memory. For example, the disk block descriptor for virtual

address 1K in Figure 9.22 shows that the page contents are in logical block 3 in the executable file.

If a process incurs a page fault for a page marked "demand fill" or "demand zero" (cases 4 and 5), the kernel allocates a free page in memory and updates the appropriate page table entry. For "demand zero," it also clears the page to zero. Finally, it clears the "demand fill" or "demand zero" flags: The page is now valid in memory and its contents are not duplicated on a swap device or in a file system. This would happen when accessing virtual addresses 3K and 65K in Figure 9.22: No process had accessed those pages since the file was *exec*ed.

The validity fault handler concludes by setting the *valid* bit of the page and clearing the *modify* bit. It recalculates the process priority, because the process may have slept in the fault handler at a kernel-level priority, giving it an unfair scheduling advantage when returning to user mode. Finally, if returning to user mode, it checks for receipt of any signals that occurred while handling the page fault.

9.2.3.2 Protection Fault Handler

The second kind of memory fault that a process can incur is a *protection* fault, meaning that the process accessed a valid page but the permission bits associated with the page did not permit access. (Recall the example of a process attempting to write its text space, in Figure 7.22.) A process also incurs a protection fault when it attempts to write a page whose *copy on write* bit was set during the *fork* system call. The kernel must determine whether permission was denied because the page requires a *copy on write* or whether something truly illegal happened.

The hardware supplies the protection fault handler with the virtual address where the fault occurred, and the fault handler finds the appropriate region and page table entry (Figure 9.25). It locks the region so that the page stealer cannot steal the page while the protection fault handler operates on it. If the fault handler determines that the fault was caused because the *copy on write* bit was set, and if the page is shared with other processes, the kernel allocates a new page and copies the contents of the old page to it; the other processes retain their references to the old page. After copying the page and updating the page table entry with the new page number, the kernel decrements the reference count of the old pfdata table entry. Figure 9.26 illustrates the scenario: Three processes share physical page 828. Process B writes the page but incurs a protection fault, because the *copy on write* bit is set. The protection fault handler allocates page 786, copies the contents of page 828 to the new page, decrements the reference count of page 828, and updates the page table entry accessed by process B to point to page 786.

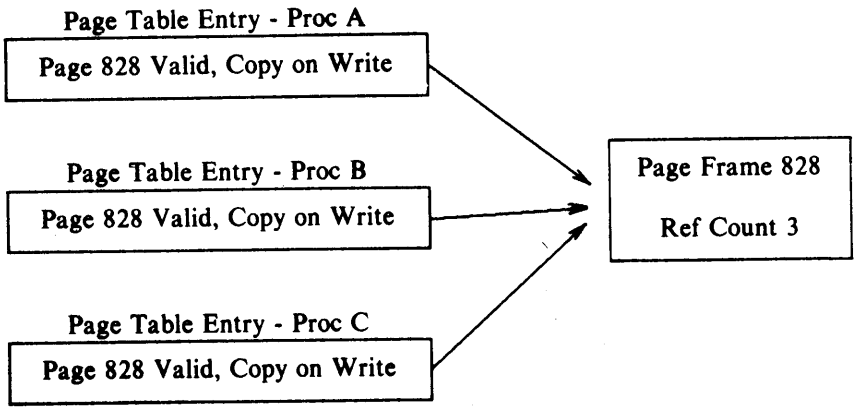
If the *copy on write* bit is set but no other processes share the page, the kernel allows the process to reuse the physical page. It turns off the *copy on write* bit and disassociates the page from its disk copy, if one exists, because other processes may share the disk copy. It then removes the pfdata table entry from the page queue, because the new copy of the virtual page is not on the swap device. Then, it

```
algorithm pfault              /* protection fault handler */
input:      address where process faulted
output:      none
{
      find region, page table entry, disk block descriptor,
                  page frame for address, lock region;
      if (page not valid in memory)
            goto out;
      if (copy on write bit not set)
            goto out;               /* real program error - signal */
      if (page frame reference count > 1)
      {
            allocate a new physical page;
            copy contents of old page to new page;
            decrement old page frame reference count;
            update page table entry to point to new physical page;
      }
      else      /* "steal" page, since nobody else is using it */
      {
            if (copy of page exists on swap device)
                  free space on swap device, break page association;
            if (page is on page hash queue)
                  remove from hash queue;
      }
      set modify bit, clear copy on write bit in page table entry;
      recalculate process priority;
      check for signals;
 out: unlock region;
}
```

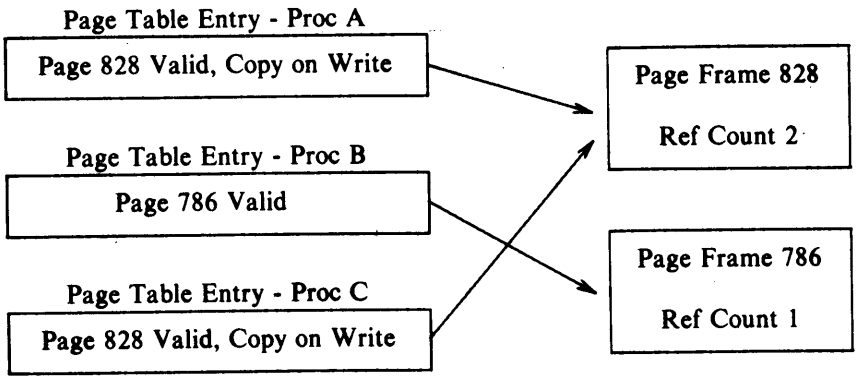
Figure 9.25. Algorithm for Protection Fault Handler

decrements the swap-use count for the page and, if the count drops to 0, frees the swap space (see exercise 9.11).

If a page table entry is invalid and its *copy on write* bit is set to cause a protection fault, let us assume that the system handles the validity fault first when a process accesses the page (exercise 9.17 covers the reverse case). Nevertheless, the protection fault handler must check that a page is still valid, because it could sleep when locking a region, and the page stealer could meanwhile swap the page from memory. If the page is invalid (the *valid* bit is clear), the fault handler returns immediately, and the process will incur a validity fault. The kernel handles the validity fault, but the process will incur the protection fault again. More than likely, it will handle the final protection fault without any more interference, because it will take a long time until the page will age sufficiently to be swapped out. Figure 9.27 illustrates this sequence of events.



(a) Before Proc B Incurs Protection Fault



(b) After Protection Fault Handler Runs for Proc B

Figure 9.26. Protection Fault with Copy on Write Set

When the protection fault handler finishes executing, it sets the *modify* and *protection* bits, but clears the *copy on write* bit. It recalculates the process priority and checks for signals, as is done at the end of the validity fault handler.

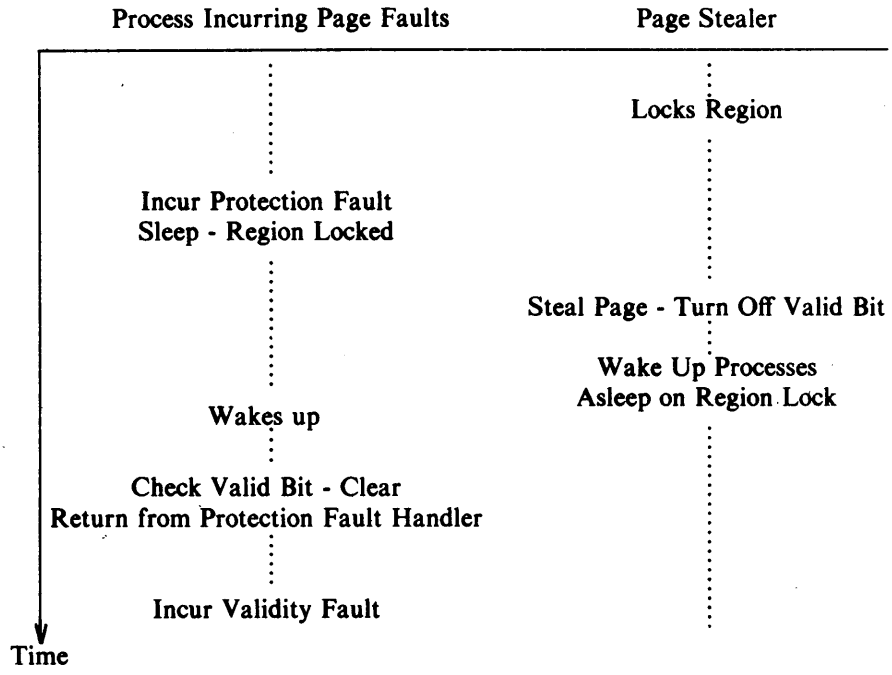


Figure 9.27. Interaction of Protection Fault and Validity Fault

9.2.4 Demand Paging on Less-Sophisticated Hardware

The algorithms for demand paging are most efficient if the hardware sets the *reference* and *modify* bits and causes a protection fault when a process writes a page whose *copy on write* bit is set. However, it is possible to implement the paging algorithms described here if the hardware recognizes only the *valid* and protection bits. If the *valid* bit is duplicated by a *software-valid* bit that indicates whether the page is really valid or not, then the kernel could turn off the hardware *valid* bit and simulate the setting of the other bits in software. For example, the VAX-11 hardware does not have a reference bit (see [Levy 82]). The kernel can turn off the hardware *valid* bit for the page and follow this scenario: If a process references the page, it incurs a page fault because the hardware *valid* bit is off, and the page fault interrupt handler examines the page. Because the *software-valid* bit is set, the kernel knows that the page is really valid and in memory; it sets the software *reference* bit and turns the hardware *valid* bit on, but it will have acquired the knowledge that the page had been referenced. Subsequent references to the page will not incur a fault because the hardware *valid* bit is on. When the page stealer examines the page, it turns off the hardware *valid* bit again, causing

Hardware Valid	Software Valid	Software Reference
Off	On	Off

(a) Before Modifying Page

Hardware Valid	Software Valid	Software Reference
On	On	On

(b) After Modifying Page

Figure 9.28. Mimicking Hardware Modify Bit in Software

processes to fault when referencing the page, repeating the cycle. Figure 9.28 depicts this case.

9.3 A HYBRID SYSTEM WITH SWAPPING AND DEMAND PAGING

Although demand paging systems treat memory more flexibly than swapping systems, situations can arise where the page stealer and validity fault handler thrash because of a shortage of memory. If the sum of the working sets of all processes is greater than the physical memory on a machine, the fault handler will usually sleep, because it cannot allocate pages for a process. The page stealer will not be able to steal pages fast enough, because all pages are in a working set. System throughput suffers because the kernel spends too much time in overhead, rearranging memory at a frantic pace.

The System V kernel runs swapping and demand paging algorithms to avoid thrashing problems. When the kernel cannot allocate pages for a process, it wakes up the swapper and puts the calling process into a state that is the equivalent of "ready to run but swapped." Several processes may be in this state simultaneously. The swapper swaps out entire processes until available memory exceeds the high-water mark. For each process swapped out, it makes one "ready-to-run but swapped" process ready to run. It does not swap those processes in via the normal swapping algorithm but lets them fault in pages as needed. Later iterations of the swapper will allow other processes to be faulted in if there is sufficient memory in the system. This method slows down the system fault rate and reduces thrashing; it is similar in philosophy to methods used in the VAX/VMS operating system ([Levy 82]).

9.4 SUMMARY

This chapter has explored the UNIX System V algorithms for process swapping and demand paging. The swapping algorithm swaps entire processes between main memory and a swap device. The kernel swaps processes from memory if their size grows such that there is no more room in main memory (as a result of a *fork*,

exec, or *sbrk* system call or as a result of normal stack growth), or if it has to make room for a process being swapped in. The kernel swaps processes in via the special swapper process, process 0, invoking it whenever there exists a "ready-to-run" process on the swap device. The swapper swaps in all such processes until there are no more processes on the swap device or until there is no more room in memory. In the latter case, it attempts to swap processes from main memory, but it reduces the amount of thrashing by prohibiting swapping of processes that do not satisfy residency requirements; hence, the swapper is not always successful in swapping all processes into memory during each pass. The clock handler wakes up the swapper every second if it has work to do.

The implementation of demand paging allows processes to execute even though their entire virtual address space is not loaded in memory; therefore the virtual size of a process can exceed the amount of physical memory available in a system. When the kernel runs low on free pages, the page stealer goes through the active pages of every region, marks pages eligible for stealing if they have aged sufficiently, and eventually copies them to a swap device. When a process addresses a virtual page that is currently swapped out, it incurs a validity fault. The kernel invokes the validity fault handler to assign a new physical page to the region and copies the contents of the virtual page to main memory.

With the implementation of the demand paging algorithm, several features improve system performance. First, the kernel uses the *copy on write* bit for *fork*ing processes, removing the need to make physical copies of pages in most cases. Second, the kernel can demand page contents of an executable file from the file system, eliminating the need for *exec* to read the file into memory immediately. This helps performance because such pages may never be needed during the lifetime of a process, and it eliminates extra thrashing caused if the page stealer were to swap such pages from memory before they are used.

9.5 EXERCISES

1. Sketch the design of an algorithm *mfree*, which frees space and returns it to a *map*.
2. Section 9.1.2 states that the system locks a process being swapped so that no other process can swap it while the first operation is underway. What would happen if the system did not lock the process?
3. Suppose the *u area* contains the segment tables and page tables for a process. How can the kernel swap the *u area* out?
4. If the kernel stack is inside the *u area*, why can't a process swap itself out? How would you encode a kernel process to swap out other processes and how should it be invoked?

* 5. Suppose the kernel attempts to swap out a process to make room for processes on a swap device. If there is not enough space on any swap devices, the swapper sleeps until more space becomes available. Is it possible for all processes in memory to be asleep and for all ready-to-run processes to be on the swap device? Describe such a scenario. What should the kernel do to rectify the situation?

6. Reconsider the swapping example in Figure 9.10 if there is room for only 1 process in memory.
7. Reconsider the swapping example in Figure 9.11. Construct an example where a process is permanently starved from use of the CPU. Is there any way to prevent this?

```
main()
{
        f();
        g();
}

f()
{
        vfork();
}

g()
{
        int blast[100], i;
        for (i = 0;  i < 100;  i++)
                blast[i] = i;
}
```

Figure 9.29. Vfork and More Corruption

8. What happens when executing the program in Figure 9.29 on a 4.2 BSD system? What happens to the parent's stack?
9. Why is it advantageous to schedule the child process before the parent after a *fork* call if *copy on write* bits are set on shared pages? How can the kernel force the child to run first?
* 10. The validity fault algorithm presented in the text swaps in one page at a time. Its efficiency can be improved by prepaging other pages around the page that caused the fault. Enhance the page fault algorithm to allow prepaging.
11. The algorithms for the page stealer and for the validity fault handler assume that the size of a page equals the size of a disk block. How should the algorithms be enhanced to handle the cases where the respective sizes are not equal?
* 12. When a process *fork*s, the page use count in the pfdata table is incremented for all shared pages. Suppose the page stealer swaps a (shared) page to a swap device, and one process (say, the parent) later faults it in. The virtual page now resides in a physical page. Explain why the child process will always be able to find a legal copy of the page, even after the parent writes the page. If the parent writes the page, why must it disassociate itself from the disk copy immediately?
13. What should a fault handler do if the system runs out of pages?
* 14. Design an algorithm that pages out infrequently used parts of the kernel. What parts of the kernel cannot be paged and how should they be identified?

15. Devise an algorithm that tracks the allocation of space on a swap device by means of a bit map instead of the maps described in the chapter. Compare the efficiency of the two methods.
16. Suppose a machine has no hardware *valid* bit but has protection bits to allow read, write, and execute from a page. Simulate manipulation of a software *valid* bit.
17. The VAX-11 hardware checks for protection faults before validity faults. What ramifications does this have for the algorithms for the fault handlers?
18. The *plock* system call allows superusers to lock and unlock the text and data regions of the calling process into memory. The swapper and page stealer processes cannot remove locked pages from memory. Processes that use this call never have to wait to be swapped in, assuring them faster response than other processes. How should the system call be implemented? Should there be an option to lock the stack region into memory too? What should happen if the total memory space of *plock*ed regions is greater than the available memory on the machine?
19. What is the program in Figure 9.30 doing? Consider an alternative paging policy, where each process has a maximum allowed number of pages in its working set.

```
struct fourmeg
{
        int page[512];          /* assume int is 4 bytes */
} fourmeg[2048];

main()
{
        for (;;)
        {
                switch(fork())
                {
                case -1:    /* parent can't fork---too many children */
                case 0:     /* child */
                        func();
                default:
                        continue;
                }
        }
}

func()
{
        int i;

        for (;;)
        {
                printf("proc %d loops again\n", getpid());
                for (i = 0;  i < 2048;  i++)
                        fourmeg[i].page[0] = i;
        }
}
```

Figure 9.30. A Misbehaving Program

10

THE I/O SUBSYSTEM

The I/O subsystem allows a process to communicate with peripheral devices such as disks, tape drives, terminals, printers, and networks, and the kernel modules that control devices are known as *device drivers*. There is usually a one-to-one correspondence between device drivers and device types: Systems may contain one disk driver to control all disk drives, one terminal driver to control all terminals, and one tape driver to control all tape drives. Installations that have devices from more than one manufacturer — for example, two brands of tape drives — may treat the devices as two different device types and have two separate drivers, because such devices may require different command sequences to operate properly. A device driver controls many physical devices of a given type. For example, one terminal driver may control all terminals connected to the system. The driver distinguishes among the many devices it controls: Output intended for one terminal must not be sent to another.

The system supports "software devices," which have no associated physical device. For example, it treats physical memory as a device to allow a process access to physical memory outside its address space, even though memory is not a peripheral device. The *ps* command, for instance, *read*s kernel data structures from physical memory to report process statistics. Similarly, drivers may write trace records useful for debugging, and a trace driver may allow users to read the records. Finally, the kernel profiler described in Chapter 8 is implemented as a driver: A process *write*s addresses of kernel routines found in the kernel symbol

table and *read*s profiling results.

This chapter examines the interfaces between processes and the I/O subsystem and between the machine and the device drivers. It investigates the general structure and function of device drivers, then treats disk drivers and terminal drivers as detailed examples of the general interface. It concludes with a description of a new method for implementing device drivers called *streams*.

10.1 DRIVER INTERFACES

The UNIX system contains two types of devices, *block* devices and *raw* or *character* devices. As defined in Chapter 2, block devices, such as disks and tapes, look like random access storage devices to the rest of the system; character devices include all other devices such as terminals and network media. Block devices may have a character device interface, too.

The user interface to devices goes through the file system (recall Figure 2.1): Every device has a name that looks like a file name and is accessed like a file. The device special file has an inode and occupies a node in the directory hierarchy of the file system. The device file is distinguished from other files by the file type stored in its inode, either "block" or "character special," corresponding to the device it represents. If a device has both a block and character interface, it is represented by two device files: its block device special file and its character device special file. System calls for regular files, such as *open*, *close*, *read*, and *write*, have an appropriate meaning for devices, as will be explained later. The *ioctl* system call provides an interface that allows processes to control character devices, but it is not applicable to regular files.[1] However, each device driver need not support every system call interface. For example, the trace driver mentioned earlier allows users to *read* records written by other drivers, but it does not allow users to *write* it.

10.1.1 System Configuration

System configuration is the procedure by which administrators specify parameters that are installation dependent. Some parameters specify the sizes of kernel tables, such as the process table, inode table, and file table, and the number of buffers to be allocated for the buffer pool. Other parameters specify device configuration, telling the kernel which devices are included in the installation and their "address." For instance, a configuration may specify that a terminal board is plugged into a

1. Conversely, the *fcntl* system call provides control of operations at the file descriptor level, not the device level. Other implementations interpret *ioctl* for all file types.

particular slot on the hardware backplane.

There are three stages at which device configuration can be specified. First, administrators can hard-code configuration data into files that are compiled and linked when building the kernel code. The configuration data is typically specified in a simple format, and a configuration program converts it into a file suitable for compilation. Second, administrators can supply configuration information after the system is already running; the kernel updates internal configuration tables dynamically. Finally, self-identifying devices permit the kernel to recognize which devices are installed. The kernel reads hardware switches to configure itself. The details of system configuration are beyond the scope of this book, but in all cases, the configuration procedure generates or fills in tables that form part of the code of the kernel.

The kernel to driver interface is described by the *block device switch* table and the *character device switch* table (Figure 10.1). Each device type has entries in the table that direct the kernel to the appropriate driver interfaces for the system calls. The *open* and *close* system calls of a device file funnel through the two device switch tables, according to the file type. The *mount* and *umount* system calls also invoke the device open and close procedures for block devices. *Read*, *write*, and *ioctl* system calls of character special files pass through the respective procedures in the character device switch table. *Read* and *write* system calls of block devices and of files on mounted file systems invoke the algorithms of the buffer cache, which invoke the device strategy procedure. Some drivers invoke the strategy procedure internally from their read and write procedures, as will be seen. The next section explores each driver interface in greater detail.

The hardware to driver interface consists of machine-dependent control registers or I/O instructions for manipulating devices and interrupt vectors. When a device interrupt occurs, the system identifies the interrupting device and calls the appropriate interrupt handler. Obviously, software devices such as the kernel profiler driver (Chapter 8) do not have a hardware interface, but other interrupt handlers may call a "software interrupt handler" directly. For example, the clock interrupt handler calls the kernel profiler interrupt handler.

Administrators set up device special files with the *mknod* command, supplying file type (block or character) and major and minor numbers. The *mknod* command invokes the *mknod* system call to create the device file. For example, in the command line

```
mknod /dev/tty13 c 2 13
```

"/dev/tty13" is the file name of the device, *c* specifies that it is a character special file (*b* specifies a block special file), 2 is the major number, and 13 is the minor number. The major number indicates a device type that corresponds to the appropriate entry in the block or character device switch tables, and the minor number indicates a unit of the device. If a process *opens* the block special file "/dev/dsk1" and its major number is 0, the kernel calls the routine *gdopen* in entry 0 of the block device switch table (Figure 10.2); if a process *reads* the character

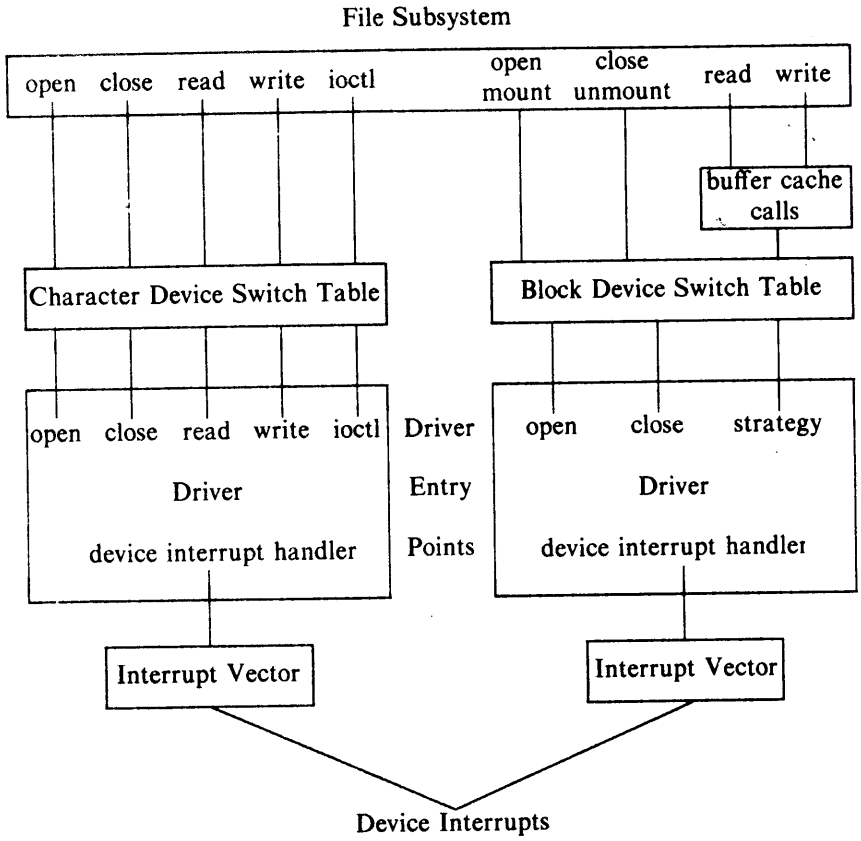


Figure 10.1. Driver Entry Points

special file "/dev/mem" and its major number is 3, the kernel calls the routine *mmread* in entry 3 of the character device switch table. The routine *nulldev* is an "empty" routine, used when there is no need for a particular driver function. Many peripheral devices can be associated with a major device number; the minor device number distinguishes them from each other. Device special files do not have to be created every time the system is booted; they need be changed only if the configuration changes, such as when adding devices to an installation.

10.1.2 System Calls and the Driver Interface

This section describes the interface between the kernel and device drivers. For system calls that use file descriptors, the kernel follows pointers from the user file

block device switch table			
entry	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

character device switch table					
entry	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

Figure 10.2. Sample Block and Character Device Switch Tables

descriptor to the kernel file table and inode, where it examines the file type and accesses the block or character device switch table, as appropriate. It extracts the major and minor numbers from the inode, uses the major number as an index into the appropriate table, and calls the driver function according to the system call being made, passing the minor number as a parameter. An important difference between system calls for devices and regular files is that the inode of a special file is *not* locked while the kernel executes the driver. Drivers frequently sleep, waiting for hardware connections or for the arrival of data, so the kernel cannot determine how long a process will sleep. If the inode was locked, other processes that access the inode (via the *stat* system call, for example) would sleep indefinitely because another process is asleep in the driver.

The device driver interprets the parameters of the system call as appropriate for the device. A driver maintains data structures that describe the state of each unit that it controls; driver functions and interrupt handlers execute according to the state of the driver and the action being done (for example, data being input or output). Each interface will now be described in greater detail.

10.1.2.1 Open

The kernel follows the same procedure for *open*ing a device as it does for *open*ing regular files (see Section 5.1), allocating an in-core inode, incrementing its reference count, and assigning a file table entry and user file descriptor. The kernel eventually returns the user file descriptor to the calling process, so that *open*ing a device looks like *open*ing a regular file. However, it invokes the device-specific *open* procedure before returning to user mode, (Figure 10.3). For a block device, it

```
algorithm open                  /* for device drivers */
input:  pathname
        openmode
output: file descriptor
{
        convert pathname to inode, increment inode reference count,
        allocate entry in file table, user file descriptor,
                as in open of regular file;

        get major, minor number from inode;

        save context (algorithm setjmp) in case of long jump from driver;

        if (block device)
        {
                use major number as index to block device switch table;
                call driver open procedure for index:
                        pass minor number, open modes;
        }
        else
        {
                use major number as index to character device switch table;
                call driver open procedure for index:
                        pass minor number, open modes;
        }

        if (open fails in driver)
                decrement file table, inode counts;
}
```

Figure 10.3. Algorithm for Opening a Device

invokes the *open* procedure encoded in the block device switch table, and for a character device, it invokes the *open* procedure in the character device switch table. If a device is both a block and a character device, the kernel will invoke the appropriate *open* procedure depending on the particular device file the user *opened*: The two open procedures may even be identical, depending on the driver.

The device-specific *open* procedure establishes a connection between the calling process and the *opened* device and initializes private driver data structures. For a terminal, for example, the *open* procedure may put the process to sleep until the machine detects a (hardware) carrier signal indicating that a user is trying to log in. It then initializes driver data structures according to appropriate terminal settings (such as the terminal baud rate). For software devices such as system memory, the *open* procedure may have no initialization to do.

If a process must sleep for some external reason when *open*ing a device, it is possible that the event that should awaken the process from its sleep may never occur. For example, if no user ever logs in to a particular terminal, the *getty* process that *open*ed the terminal (Section 7.9) sleeps until a user attempts to log in, potentially a long time. The kernel must be able to awaken the process from its sleep and cancel the *open* call on receipt of a signal: It must reset the inode, file table entry, and user file descriptor that it had allocated before entry into the driver, because the *open* fails. Hence, the kernel saves the process context using algorithm *setjmp* (Section 6.4.4) before entering the device-specific *open* routine; if the process awakens from its sleep because of a signal, the kernel restores the process context to its state before entering the driver using algorithm *longjmp* (Section 6.4.4) and releases all data structures it had allocated for the *open*. Similarly, the driver can catch the signal and clean up private data structures, if necessary. The kernel also readjusts the file system data structures when the driver encounters error conditions, such as when a user attempts to access a device that was not configured. The *open* call fails in such cases.

Processes may specify various options to qualify the device *open*. The most common option is "no delay," meaning that the process will not sleep during the *open* procedure if the device is not ready. The *open* system call returns immediately, and the user process has no knowledge of whether a hardware connection was made or not. *Open*ing a device with the "no delay" option also affects the semantics of the *read* system call, as will be seen (Section 10.3.4).

If a device is *open*ed many times, the kernel manipulates the user file descriptors and the inode and file table entries as described in Chapter 5, invoking the device specific *open* procedure for each *open* system call. The device driver can thus count how many times a device was *open*ed and fail the *open* call if the count is inappropriate. For example, it makes sense to allow multiple processes to *open* a terminal for writing so that users can exchange messages. But it does not make sense to allow multiple processes to *open* a printer for writing simultaneously, since they could overwrite each other's data. The differences are practical rather than implementational: allowing simultaneous writing to terminals fosters communication between users; preventing simultaneous writing to printers increases the chance of getting readable printouts.[2]

10.1.2.2 Close

A process severs its connection to an *open* device by *clos*ing it. However, the kernel invokes the device-specific *close* procedure only for the last *close* of the

2. In practice, printers are usually controlled by special spooler processes, and permissions are set up so that only the spooler can access the printer. But the analogy is still applicable.

device, that is, only if no other processes have the device *open*, because the device *close* procedure terminates hardware connections; clearly this must wait until no processes are accessing the device. Because the kernel invokes the device *open* procedure during every *open* system call but invokes the device *close* procedure only once, the device driver is never sure how many processes are still using the device. Drivers can easily put themselves out of state if not coded carefully: If they sleep in the *close* procedure and another process *open*s the device before the close completes, the device can be rendered useless if the combination of open and close results in an unrecognized state.

```
algorithm close                    /* for devices */
input:  file descriptor
output: none
{
        do regular close algorithm (chapter 5xxx);
        if (file table reference count not 0)
                goto finish;
        if (there is another open file and its major, minor numbers
                                are same as device being closed)
                goto finish;                    /* not last close after all */
        if (character device)
        {
                use major number to index into character device switch table;
                call driver close routine:  parameter minor number;
        }
        if (block device)
        {
                if (device mounted)
                        goto finish;
                write device blocks in buffer cache to device;
                use major number to index into block device switch table;
                call driver close routine:  parameter minor number;
                invalidate device blocks still in buffer cache;
        }
    finish:
        release inode;
}
```

Figure 10.4. Algorithm for Closing a Device

The algorithm for *clos*ing a device is similar to the algorithm for closing a regular file (Figure 10.4). However, before the kernel releases the inode it does operations specific to device files.

1. It searches the file table to make sure that no other processes still have the device *open*. It is not sufficient to rely on the file table count to indicate the

last close of a device, because several processes may access the device via a different file table entry. It is also not sufficient to rely on the inode table count, because several device files may specify the same device. For example, the results of the following *ls −l* command show two character device files (the first "c" on the line) that refer to one device, because their major and minor numbers (9 and 1) are equal. The link count of 1 for each file implies that there are two inodes.

```
crw--w--w-    1 root   vis    9, 1 Aug 6  1984   /dev/tty01
crw--w--w-    1 root   unix   9, 1 May 3 15:02   /dev/fty01
```

If processes *open* the two files independently, they access different inodes but the same device.

2. For a character device, the kernel invokes the device *close* procedure and returns to user mode. For a block device, the kernel searches the mount table to make sure that the device does not contain a mounted file system. If there is a mounted file system from the block device, the kernel cannot invoke the device close procedure, because it is not the last *close* of the device. Even if the device does not contain a mounted file system, the buffer cache could still contain blocks of data that were left over from a previously mounted file system and never written to the device, because they were marked "delayed write." The kernel therefore searches the buffer cache for such blocks and writes them to the device before invoking the device *close* procedure. After *clos*ing the device, the kernel again goes through the buffer cache and invalidates all buffers that contain blocks for the now *close*d device, allowing buffers with useful data to stay in the cache longer.
3. The kernel releases the inode of the device file.

To summarize, the device *close* procedure severs the device connection and reinitializes driver data structures and device hardware, so that the kernel can reopen the device later on.

10.1.2.3 Read and Write

The kernel algorithms for *read* and *write* of a device are similar to those for a regular file. If the process is *read*ing or *writ*ing a character device, the kernel invokes the device driver *read* or *write* procedure. Although there are important cases where the kernel transmits data directly between the user address space and the device, device drivers may buffer data internally. For example, terminal drivers use *clist*s to buffer data (Section 10.3.1). In such cases, the device driver allocates a "buffer," copies data from user space during a *write*, and outputs the data from the "buffer" to the device. The driver write procedure throttles the amount of data being output (called flow control): If processes generate data faster than the device can output it, the write procedure puts processes to sleep until the device can accept more data. For a *read*, the device driver receives the data from the device in a

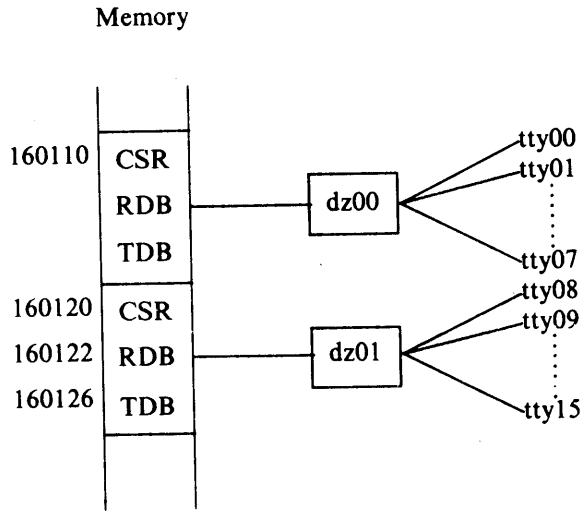


Figure 10.5. Memory Mapped I/O with the VAX DZ11 Controller

buffer and copies the data from the buffer to the user address specified in the system call.

The precise method in which a driver communicates with a device depends on the hardware. Some machines provide *memory mapped I/O*, meaning that certain addresses in the kernel address space are not locations in physical memory but are special registers that control particular devices. By writing control parameters to specified registers according to hardware specifications, the driver controls the device. For example, I/O controllers for the VAX-11 computer contain special registers for recording device status (*control and status registers*) and for data transmission (*data buffer registers*), which are configured at specific addresses in physical memory. In particular, the VAX DZ11 terminal controller controls 8 asynchronous lines for terminal communication (see [Levy 80] for more detail on the VAX architecture). Assume that the *control and status register* of a particular DZ11 is at address 160120, the *transmit data buffer register* is at address 160126, and the *receive data buffer register* is at address 160122 (Figure 10.5). To write a character to terminal "/dev/tty09", the terminal driver writes the number 1 (1 = 9 modulo 8) to a specified bit position in the *control and status register* and then writes the character to the *transmit data buffer register*. The operation of writing the *transmit data buffer register* transmits the data. The DZ11 controller sets a *done* bit in the *control and status register* when it is ready to accept more data. The driver can optionally set a *transmit interrupt enable* bit in the *control and status register*, which causes the DZ11 controller to interrupt the system when it is ready to accept more data. Reading data from the DZ11 is similar.

Other machines have *programmed I/O*, meaning that the machine contains instructions to control devices. Drivers control devices by executing the appropriate instructions. For example, the IBM 370 computer has a *Start I/O* instruction to initiate an I/O operation to a device. The method a driver uses to communicate with peripherals is transparent to the user.

Because the interface between device drivers and the underlying hardware is machine dependent, no standard interfaces exist at this level. For both memory-mapped I/O and programmed I/O, a driver can issue control sequences to a device to set up *direct memory access (DMA)* between the device and memory. The system allows bulk DMA transfer of data between the device and memory in parallel to CPU operations, and the device interrupts the system when such a transfer has completed. The driver sets up the virtual memory mapping so that the correct locations in memory are used for DMA.

High-speed devices can sometimes transfer data directly between the device and the user's address space, without intervention of a kernel buffer. This results in higher transfer speed because there is one less copy operation in the kernel, and the amount of data transmitted per transfer operation is not bounded by the size of kernel buffers. Drivers that make use of this "raw" I/O transfer usually invoke the block strategy interface from the character read and write procedures if they have a block counterpart.

10.1.2.4 Strategy Interface

The kernel uses the *strategy* interface to transmit data between the buffer cache and a device, although as mentioned above, the read and write procedures of character devices sometimes use their (block counterpart) *strategy* procedure to transfer data directly between the device and the user address space. The strategy procedure may queue I/O jobs for a device on a work list or do more sophisticated processing to schedule I/O jobs. Drivers can set up data transmission for one physical address or many, as appropriate. The kernel passes a buffer header address to the driver strategy procedure; the header contains a list of (page) addresses and sizes for transmission of data to or from the device. This is also how the swapping operations discussed in Chapter 9 work. For the buffer cache, the kernel transmits data from one data address; when swapping, the kernel transmits data from many data addresses (pages). If data is being copied to or from the user's address space, the driver must lock the process (or at least, the relevant pages) in memory until the I/O transfer is complete.

For example, after *mount*ing a file system, the kernel identifies every file in the file system by its device number and inode number. The device number is an encoding of the device major and minor numbers. When the kernel accesses a block from a file, it copies the device number and block number into the buffer header, as described in Chapter 3. When the buffer cache algorithms (*bread* or *bwrite*, for example) access the disk, they invoke the strategy procedure indicated by the device major number. The strategy procedure uses the minor number and

block number fields in the buffer header to identify where to find the data on the device, and it uses the buffer address to identify where the data should be transferred. Similarly, if a process accesses a block device directly (that is, the process *open*s the block device and *read*s or *write*s it), it uses the buffer cache algorithms, and the interface works as just described.

10.1.2.5 Ioctl

The *ioctl* system call is a generalization of the terminal-specific *stty* (set terminal settings) and *gtty* (get terminal settings) system calls available in earlier versions of the UNIX system. It provides a general, catch-all entry point for device specific commands, allowing a process to set hardware options associated with a device and software options associated with the driver. The specific actions specified by the *ioctl* call vary per device and are defined by the device driver. Programs that use *ioctl* must know what type of file they are dealing with, because they are device-specific. This is an exception to the general rule that the system does not differentiate between different file types. Section 10.3.3 provides more detail on the use of *ioctl* for terminals.

The syntax of the system call is

```
ioctl(fd, command, arg);
```

where *fd* is the file descriptor returned by a prior *open* system call, *command* is a request of the driver to do a particular action, and *arg* is a parameter (possibly a pointer to a structure) for the *command*. Commands are driver specific; hence, each driver interprets commands according to internal specifications, and the format of the data structure *arg* depends on the command. Drivers can read the data structure *arg* from user space according to predefined formats, or they can write device settings into user address space at *arg*. For instance, the *ioctl* interface allows users to set terminal baud rates; it allows users to rewind tapes on a tape drive; finally, it allows network operations such as specifying virtual circuit numbers and network addresses.

10.1.2.6 Other File System Related Calls

File system calls such as *stat* and *chmod* work for devices as they do for regular files; they manipulate the inode without accessing the driver. Even the *lseek* system call works for devices. For example, if a process *lseek*s to a particular byte offset on a tape, the kernel updates the file table offset but does no driver-specific operations. When the process later *read*s or *write*s, the kernel moves the file table offset to the *u area*, as is done for regular files, and the device *physically* seeks to the correct offset indicated in the *u area*. An example in Section 10.3 illustrates this case.

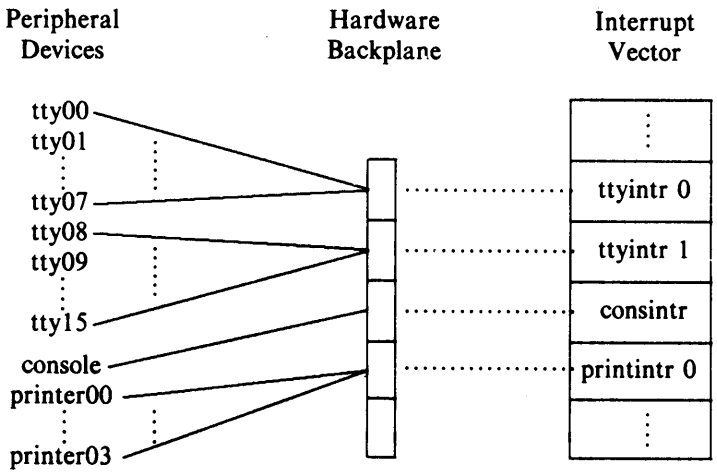


Figure 10.6. Device Interrupts

10.1.3 Interrupt Handlers

As previously explained (Section 6.4.1), occurrence of an interrupt causes the kernel to execute an interrupt handler, based on the correlation of the interrupting device and an offset in the interrupt vector table. The kernel invokes the device specific interrupt handler, passing it the device number or other parameters to identify the specific unit that caused the interrupt. For example, Figure 10.6 shows two entries in an interrupt vector table for handling terminal interrupts ("ttyintr"), each handling interrupts for 8 terminals. If device *tty09* interrupts the system, the system calls the interrupt handler associated with the hardware position of the interrupting device. Because many physical devices can be associated with one interrupt vector entry, the driver must be able to resolve which device caused the interrupt. In the figure, the two interrupt vector entries for "ttyintr" are labeled 0 and 1, implying that the system distinguishes between the two vector entries in some way when calling the interrupt handler, such as using that number as a parameter to the call. The interrupt handler would use that number and other information passed by the interrupt mechanism to ascertain that device *tty09* interrupted the system and not *tty12*, for example. This example is a simplification of what happens on real systems, where several levels of controllers and their interrupt handlers enter the picture, but it illustrates the general principles.

In summary, the device number used by the interrupt handler identifies a hardware unit, and the minor number in the device file identifies a device for the kernel. The device driver correlates the minor device number to the hardware unit number.

10.2 DISK DRIVERS

Historically, disk units on UNIX systems have been configured into sections that contain individual file systems, allowing "the [disk] pack to be broken up into more manageable pieces" (see [System V 84b]). For instance, if a disk contains four file systems, an administrator may leave one unmounted, *mount* another "read-only," and *mount* the last two "read-write." Even though all the file systems coexist on one physical unit, users cannot access files in the unmounted file system using the access methods described in Chapters 4 and 5, nor can any users write files in the "read-only" file system. Furthermore, since each section (and hence file system) spans contiguous tracks and cylinders of the disk, it is easier to copy entire file systems than if they were dispersed throughout an entire disk volume.

The disk driver translates a file system address, consisting of a logical device number and block number, to a particular sector on the disk. The driver gets the address in one of two ways: Either the strategy procedure uses a buffer from the buffer pool and the buffer header contains the device and block number, or the *read* and *write* procedures are passed the logical (minor) device number as a parameter; they convert the byte offset saved in the *u area* to the appropriate block address. The disk driver uses the device number to identify the physical drive and particular section to be used, maintaining internal tables to find the sector that marks the beginning of a disk section. Finally, it adds the block number of the file system to the start sector number to identify the sector used for the I/O transmission.

Section	Start Block	Length in Blocks
	Size of block = 512 bytes	
0	0	64000
1	64000	944000
2	168000	840000
3	336000	672000
4	504000	504000
5	672000	336000
6	840000	168000
7	0	1008000

Figure 10.7. Disk Sections for RP07 Disk

Historically, the sizes and lengths of disk sections have been fixed according to the disk type. For instance, the DEC RP07 disk is partitioned into the sections shown in Figure 10.7. Suppose the files "/dev/dsk0", "/dev/dsk1", "/dev/dsk2" and "/dev/dsk3" correspond to sections 0 through 3 of an RP07 disk and have minor numbers 0 through 3. Assume the size of a logical file system block is the same as that of a disk block. If the kernel attempts to access block 940 in the file system contained in "/dev/dsk3", the disk driver converts the request to access

block 336940 (section 3 starts at block 336000; 336000 + 940 = 336940) on the disk.

The sizes of disk sections vary, and administrators configure file systems in sections of the appropriate size: Large file systems go into large sections, and so on. Sections may overlap on disk. For example, Sections 0 and 1 in the RP07 disk are disjoint, but together they cover blocks 0 to 1008000, the entire disk. Section 7 also covers the entire disk. The overlap of sections does not matter, provided that the file systems contained in the sections are configured such that *they* do not overlap. It is advantageous to have one section include the entire disk, since the entire volume can thus be quickly copied.

The use of fixed sections restricts the flexibility of disk configuration. The hard-coded knowledge of disk sections should not be put into the disk driver but should be placed in a configurable volume table of contents on the disk. However, it is difficult to find a generic position on all disks for the volume table of contents and retain compatibility with previous versions of the system. Current implementations of System V expect the boot block of the first file system on a disk to occupy the first sector of the volume, although that is the most logical place for a volume table of contents. Nevertheless, the disk driver could contain hard-coded information on where the volume table of contents is stored for that particular disk, allowing variable sized disk sections.

Because of the high level of disk traffic typical of UNIX systems, the disk driver must maximize data throughput to get the best system performance. Most modern disk controllers take care of disk job scheduling, positioning the disk arm, and transferring data between the disk and the CPU; otherwise, the disk driver must do these tasks.

Utility programs can use either the raw or block interface to access disk data directly, bypassing the regular file system access method investigated in Chapters 4 and 5. Two important programs that deal directly with the disk are *mkfs* and *fsck*. *Mkfs* formats a disk section for a UNIX file system, creating a super block, inode list, linked list of free disk blocks, and a root directory on the new file system. *Fsck* checks the consistency of an existing file system and corrects errors, as presented in Chapter 5.

Consider the program in Figure 10.8 and the files "/dev/dsk15" and "/dev/rdsk15", and suppose the *ls* command prints the following information.

```
ls -l /dev/dsk15 /dev/rdsk15

br--------   2 root   root   0, 21 Feb 12 15:40   /dev/dsk15
crw-rw----   2 root   root   7, 21 Mar  7 09:29   /dev/rdsk15
```

It shows that "/dev/dsk15" is a block device owned by "root," and only "root" can *read* it directly. Its major number is 0, and its minor number is 21. The file "/dev/rdsk15" is a character device owned by "root" but allows read and write permission for the owner and group (both root here). Its major number is 7, and its minor number is 21. A process *open*ing the files gains access to the device via

```
#include "fcntl.h"
main()
{
        char buf1[4096], buf2[4096];
        int fd1, fd2, i;

        if (((fd1 = open("/dev/dsk5", O_RDONLY)) == -1) ||
                        ((fd2 = open("/dev/rdsk5", O_RDONLY)) == -1))
        {
                printf("failure on open\n");
                exit();
        }

        lseek(fd1, 8192L, 0);
        lseek(fd2, 8192L, 0);

        if ((read(fd1, buf1, sizeof(buf1)) == -1) || (read(fd2, buf2, sizeof(buf2)) == -1))
        {
                printf("failure on read\n");
                exit();
        }

        for (i = 0;  i < sizeof(buf1);  i++)
                if (buf1[i] != buf2[i])
                {
                        printf("different at offset %d\n", i);
                        exit();
                }
        printf("reads match\n");
}
```

Figure 10.8. Reading Disk Data Using Block and Raw Interface

the block device switch table and the character device switch table, respectively, and the minor number 21 informs the driver which disk section is being accessed — for example, physical drive 2, section 1. Because the minor numbers are identical for each file, both refer to the same disk section, assuming this is one device.[3] Thus, a process executing the program *open*s the same driver twice (through different interfaces), *lseek*s to byte offset 8192 in the devices, and *read*s data from that

3. There is no way to verify that a character driver and a block driver refer to the same device, except by examination of the system configuration tables and the driver code.

position. The results of the *read* calls should be identical, assuming no other file system activity.

Programs that *read* and *write* the disk directly are dangerous because they can read or write sensitive data, jeopardizing system security. Administrators must protect the block and raw interfaces by putting the appropriate permissions on the disk device files. For example, the disk files "/dev/dsk15" and "/dev/rdsk15" should be owned by "root," and their permissions should allow "root" to read the file but should not allow any other users to read or write.

Programs that *read* and *write* the disk directly can also destroy the consistency of file system data. The file system algorithms explained in Chapters 3, 4, and 5 coordinate disk I/O operations to maintain a consistent view of disk data structures, including linked lists of free disk blocks and pointers from inodes to direct and indirect data blocks. Processes that access the disk directly bypass these algorithms. Even if they are carefully encoded, there is still a consistency problem if they run while other file system activity is going on. For this reason, *fsck* should not be run on an active file system.

The difference between the two disk interfaces is whether they deal with the buffer cache. When accessing the block device interface, the kernel follows the same algorithm as for regular files, except that after converting the logical byte offset into a logical block offset (recall algorithm *bmap* in Chapter 4), it treats the logical block offset as a physical block number in the file system. It then accesses the data via the buffer cache and, ultimately, the driver strategy interface. However, when accessing the disk via the raw interface, the kernel does not convert the byte offset into the file but passes the offset immediately to the driver via the *u area*. The driver *read* or *write* routine converts the byte offset to a block offset and copies the data directly to the user address space, bypassing kernel buffers.

Thus, if one process *write*s a block device and a second process then *read*s a raw device at the same address, the second process may not read the data that the first process had written, because the data may still be in the buffer cache and not on disk. However, if the second process had *read* the block device, it would automatically pick up the new data, as it exists in the buffer cache.

Use of the raw interface may also introduce strange behavior. If a process *read*s or *write*s a raw device in units smaller than the block size, for example results are driver-dependent. For instance, when issuing 1-byte *write*s to a tape drive, each byte may appear in different tape blocks.

The advantage of using the raw interface is speed, assuming there is no advantage to caching data for later access. Processes accessing block devices transfer blocks of data whose size is constrained by the file system logical block size. For example, if a file system has a logical block size of 1K bytes, at most 1K bytes are transferred per I/O operation. However, processes accessing the disk as a raw device can transfer many disk blocks during a disk operation, subject to the capabilities of the disk controller. Functionally, the process sees the same result, but the raw interface may be much faster. In Figure 10.8 for example, when a process *read*s 4096 bytes using the block interface for a file system with 1K bytes

per block, the kernel loops internally four times and accesses the disk during each iteration before returning from the system call, but when it *read*s the raw interface, the driver may satisfy the *read* with one disk operation. Furthermore, use of the block interface entails an extra copy of data between user address space and kernel buffers, which is avoided in the raw interface.

10.3 TERMINAL DRIVERS

Terminal drivers have the same function as other drivers: to control the transmission of data to and from terminals. However, terminals are special, because they are the user's interface to the system. To accommodate interactive use of the UNIX system, terminal drivers contain an internal interface to *line discipline* modules, which interpret input and output. In *canonical* mode, the line discipline converts raw data sequences typed at the keyboard to a canonical form (what the user really meant) before sending the data to a receiving process; the line discipline also converts raw output sequences written by a process to a format that the user expects. In *raw* mode, the line discipline passes data between processes and the terminal without such conversions.

For example, programmers are notoriously fast but error-prone typists. Terminals provide an "erase" key (or such a key can be so designated) such that the user can logically erase part of the typed sequence and enter corrections. The terminal sends the entire sequence to the machine, including the erase characters.[4] In canonical mode, the line discipline buffers the data into lines (the sequence of characters until a carriage-return[5] character) and processes erase characters internally before sending the revised sequence to the reading process.

The functions of a line discipline are

- to parse input strings into lines;
- to process erase characters;
- to process a "kill" character that invalidates all characters typed so far on the current line;
- to echo (write) received characters to the terminal;
- to expand output such as tab characters to a sequence of blank spaces;
- to generate signals to processes for terminal hangups, line breaks, or in response to a user hitting the delete key;
- to allow a raw mode that does not interpret special characters such as erase, kill or carriage return.

---

4. This section will assume the use of dumb terminals, which transmit all characters typed by the user without processing them.

5. This chapter will use the generic term "carriage return" for "carriage return" and "new-line" characters.

The support of raw mode implies the use of an asynchronous terminal, because processes can *read* characters as they are typed instead of waiting until a user hits a carriage return or "enter" key.

Ritchie notes that the original terminal line disciplines used during system development in the early 1970s were in the shell and editor programs, not in the kernel (see page 1580 of [Ritchie 84]). However, because their function is needed by many programs, their proper place is in the kernel. Although the line discipline performs a function that places it logically between the terminal driver and the rest of the kernel, the kernel does not invoke the line discipline directly but only through the terminal driver. Figure 10.9 shows the logical flow of data through the terminal driver and line discipline and the corresponding flow of control through the terminal driver. Users can specify what line discipline should be used via an *ioctl* system call, but it is difficult to implement a scheme such that one device uses several line disciplines simultaneously, where each line discipline module successively calls the next module to process the data in turn.

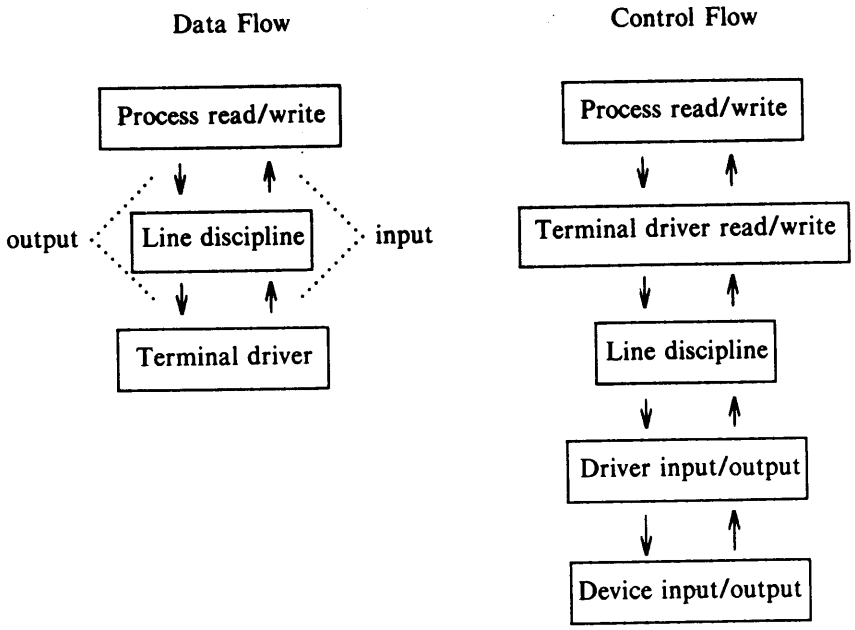


Figure 10.9. Call Sequence and Data Flow through Line Discipline

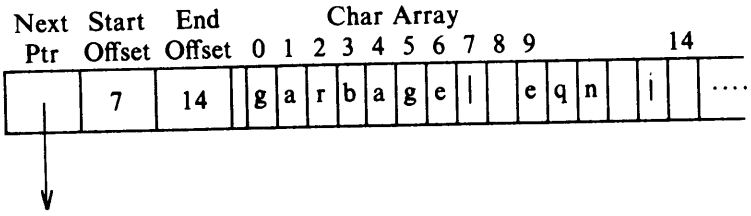


Figure 10.10. A Cblock

10.3.1 Clists

Line disciplines manipulate data on *clists*. A clist, or character list, is a variable-length linked list of *cblocks* with a count of the number of characters on the list. A cblock contains a pointer to the next cblock on the linked list, a small character array to contain data, and a set of offsets indicating the position of the valid data in the cblock (Figure 10.10). The *start* offset indicates the first location of valid data in the array, and the *end* offset indicates the first location of nonvalid data.

The kernel maintains a linked list of free cblocks and has six operations on clists and cblocks.

1. It has an operation to assign a cblock from the free list to a driver.
2. It also has an operation to return a cblock to the free list.
3. The kernel can retrieve the first character from a clist: It removes the first character from the first cblock on the clist and adjusts the clist character count and the indices into the cblock so that subsequent operations will not retrieve the same character. If a retrieval operation consumes the last character of a cblock, the kernel places the empty cblock on the free list and adjusts the clist pointers. If a clist contains no characters when a retrieval operation is done, the kernel returns the null character.
4. The kernel can place a character onto the end of a clist by finding the last cblock on the clist, putting the character onto it, and adjusting the offset values. If the cblock is full, the kernel allocates a new cblock, links it onto the end of the clist, and places the character into the new cblock.
5. The kernel can remove a group of characters from the beginning of a clist one cblock at a time, the operation being equivalent to removing all the characters in the cblock one at a time.
6. The kernel can place a cblock of characters onto the end of a clist.

Clists provide a simple buffer mechanism, useful for the small volume of data transmission typical of slow devices such as terminals. They allow manipulation of data one character at a time or in groups of cblocks. For example, Figure 10.11 depicts the removal of characters from a clist; the kernel removes one character at a time from the first cblock on the clist (Figure 10.11a–c) until there are no more

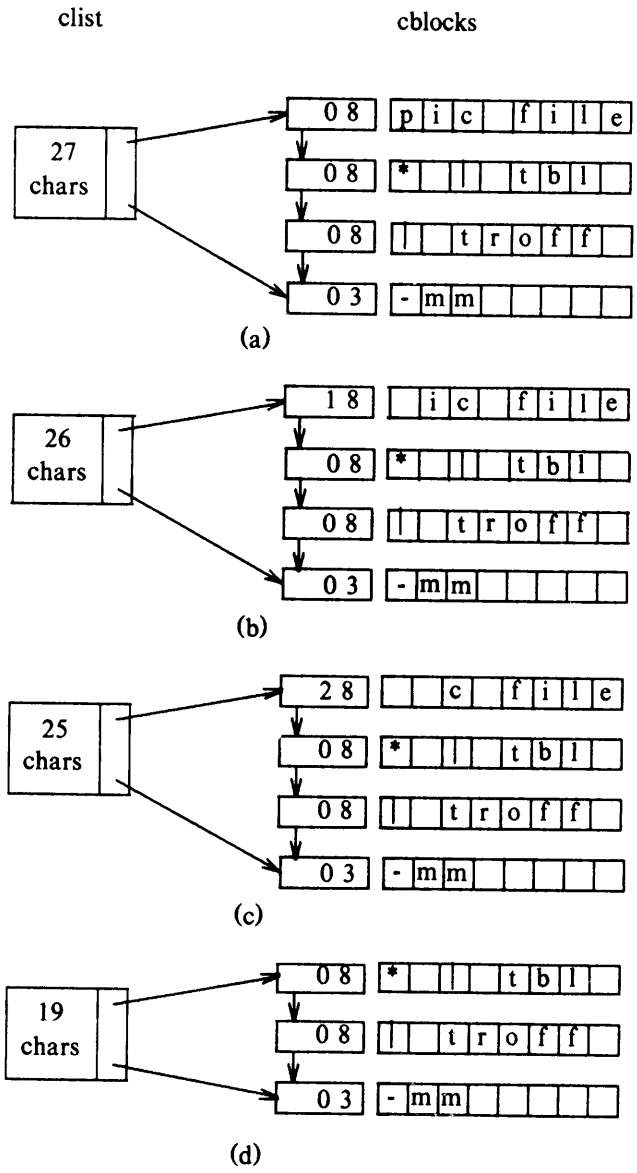


Figure 10.11. Removing Characters from a Clist

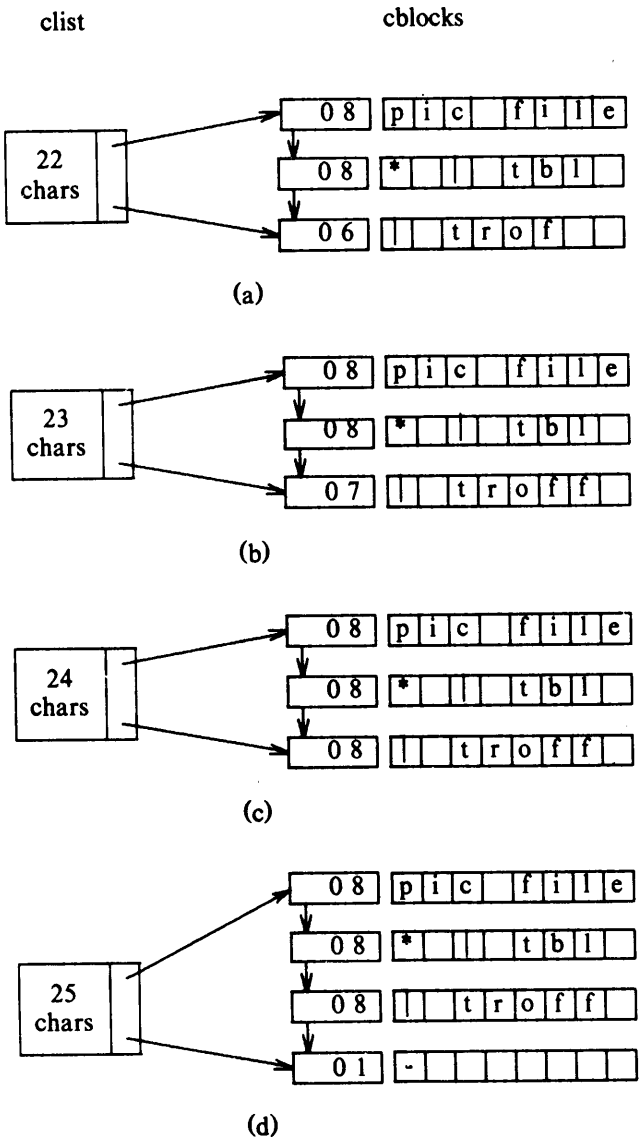


Figure 10.12. Placing a Character on a Clist

characters in the cblock (Figure 10.11d); then, it adjusts the clist pointer to point to the next cblock, which becomes the first one on the linked list. Similarly, Figure 10.12 depicts how the kernel puts characters onto a clist; assuming a cblock holds up to 8 characters, the kernel links a new cblock onto the end of the linked list (Figure 10.12d).

10.3.2 The Terminal Driver in Canonical Mode

The data structures for terminal drivers have three clists associated with them: a clist to store data for output to the terminal, a clist to store "raw" input data provided by the terminal interrupt handler as the user typed it in, and a clist to store "cooked" input data, after the line discipline converts special characters in the raw clist, such as the erase and kill characters.

```
algorithm terminal_write
{
        while (more data to be copied from user space)
        {
                if (tty flooded with output data)
                {
                        start write operation to hardware with data
                                                on output clist;
                        sleep (event: tty can accept more data);
                        continue;       /* back to while loop */
                }
                copy cblock size of data from user space to output clist:
                        line discipline converts tab characters, etc;
        }

        start write operation to hardware with data on output clist;
}
```

Figure 10.13. Algorithm for Writing Data to a Terminal

When a process writes a terminal (Figure 10.13), the terminal driver invokes the line discipline. The line discipline loops, reading output characters from user address space and placing them onto the output clist, until it exhausts the data. The line discipline processes output characters, expanding tab characters to a series of space characters, for example. If the number of characters on the output clist becomes greater than a high-water mark, the line discipline calls driver procedures to transmit the data on the output clist to the terminal and puts the writing process to sleep. When the amount of data on the output clist drops below a low-water mark, the interrupt handler awakens all processes asleep on the event the terminal can accept more data. The line discipline finishes its loop, having copied all the

output data from user space to the output clist, and calls driver procedures to transmit the data to the terminal, as described earlier.

If multiple processes write to a terminal, they follow the given procedure independently. The output could be garbled; that is, data written by the processes may be interleaved on the terminal. This could happen because a process may *write* the terminal using several *write* system calls. The kernel could switch context while the process is in user mode between successive *write* system calls, and newly scheduled processes could *write* the terminal while the original process sleeps. Output data could also be garbled at a terminal because a writing process may sleep in the middle of a *write* system call while waiting for previous output data to drain from the system. The kernel could schedule other processes that *write* the terminal before the original process is rescheduled. Because of this case, the kernel does *not* guarantee that the contents of the data buffer to be output by a *write* system call appear contiguously on the terminal.

```
char form[] = "this is a sample output string from child ";
main()
{
        char output[128];
        int i;

        for (i = 0;  i < 18;  i++)
        {
                switch (fork())
                {
                case -1:        /* error --- hit max procs */
                        exit();

                default:        /* parent process */
                        break;

                case 0:                 /* child process */
                        /* format output string in variable output */
                        sprintf(output, "%s%d\n%s%d\n", form, i, form, i);
                        for (;;)
                                write(1, output, sizeof(output));
                }
        }
}
```

Figure 10.14. Flooding Standard Output with Data

Consider the program in Figure 10.14. The parent process creates up to 18 children; each child process formats a string (via the library function *sprintf*) in the array *output*, which includes a message and the value of *i* at the time of the *fork*

and then goes into a loop, *writ*ing the string to its standard output file during each iteration. If the standard output is the terminal, the terminal driver regulates the flow of data to the terminal. The output string is more than 64 characters long, too large to fit into a cblock (64 bytes long) in System V implementations. Hence, the terminal driver needs more than one cblock for each *write* call, and output could become garbled. For example, the following lines were part of the output produced when running the program on an AT&T 3B20 computer:

```
this is a sample output string from child 1
this is a sample outthis is a sample output string from child 0
```

Reading data from a terminal in canonical mode is a more complex operation. The *read* system call specifies the number of bytes the process wants to *read*, but the line discipline satisfies the *read* on receipt of a carriage return even though the character count is not satisfied. This is practical, since it is impossible for a process to predict how many characters the user will enter at the keyboard, and it does not make sense to wait for the user to type a large number of characters. For example, users type command lines to the shell and expect the shell to respond to the command on receipt of a carriage return character. It makes no difference whether the commands are simple, such as "date" or "who," or whether they are more complicated command sequences such as

```
pic file* | tbl | eqn | troff -mm -Taps | apsend
```

The terminal driver and line discipline know nothing about shell syntax, and rightly so, because other programs that read terminals (such as editors) have different command syntax. Hence, the line discipline satisfies *read* calls on receipt of a carriage return.

Figure 10.15 shows the algorithm for reading a terminal. Assume the terminal is in canonical mode; Section 10.3.3 will cover the case of raw mode. If no data is currently on either input clist, the reading process sleeps until the arrival of a line of data. When data is entered, the terminal interrupt handler invokes the line discipline "interrupt handler," which places the data on the raw clist for input to *read*ing processes and on the output clist for echoing back to the terminal. If the input string contains a carriage return, the interrupt handler awakens all sleeping reader processes. When a reading process runs, the driver removes characters from the raw clist, does erase and kill character processing, and places the characters on the canonical clist. It then copies characters to user address space until the carriage return character or until it satisfies the count in the *read* system call, whichever number is smalier. However, a process may find that the data for which it woke up no longer exists: Other processes may *read* the terminal and remove the data from the raw clist before the first process is rescheduled. This is similar to what happens when multiple processes read data from a pipe.

Character processing in the input and output direction is asymmetric, evidenced by the two input clists and the one output clist. The line discipline outputs data from user space, processes it, and places it on the output clist. To be symmetric,

```
algorithm terminal_read
{
      if (no data on canonical clist)
      {
            while (no data on raw clist)
            {
                  if (tty opened with no delay option)
                        return;
                  if (tty in raw mode based on timer and timer not active)
                        arrange for timer wakeup (callout table);
                  sleep (event: data arrives from terminal);
            }

            /* there is data on raw clist */
            if (tty in raw mode)
                  copy all data from raw clist to canonical clist;
            else      /* tty is in canonical mode */
            {
                  while (characters on raw clist)
                  {
                        copy one character at a time from raw clist
                              to canonical clist:
                              do erase, kill processing;
                        if (char is carriage return or end-of-file)
                              break;          /* out of while loop */
                  }
            }
      }

      while (characters on canonical list and read count not satisfied)
            copy from cblocks on canonical list to user address space;
}
```

Figure 10.15. Algorithm for Reading a Terminal

there should be only one input clist. However, this would require the interrupt handler to process erase and kill characters, making it more complex and time consuming, and blocking out other interrupts at a critical time. Use of two input clists means that the interrupt handler can simply dump characters onto the raw clist and wake up *read*ing processes, which properly incur the expense of processing input data. Nevertheless, the interrupt handler puts input characters immediately on the output clist, so that the user experiences minimal delay in seeing typed characters on the terminal.

Figure 10.16 shows a program where a process creates many child processes that *read* their standard input file, contending for terminal data. Terminal input is

```
char input[256];

main()
{
        register int i;

        for (i = 0;  i < 18;  i++)
        {
                switch (fork())
                {
                case -1:        /* error */
                        printf("error cannot fork\n");
                        exit();

                default:        /* parent process */
                        break;

                case 0:                 /* child process */
                        for (;;)
                        {
                                read(0, input, 256);        /* read line */
                                printf("%d read %s\n", i, input);
                        }
                }
        }
}
```

Figure 10.16. Contending for Terminal Input Data

usually too slow to satisfy all the *read*ing processes, so the processes will spend most of their time sleeping in the *terminal_read* algorithm, waiting for input data. When a user enters a line of data, the terminal interrupt handler awakens all the *read*ing processes; since they slept at the same priority level, they are eligible to run at the same priority. The user cannot predict which process runs and *read*s the line of data; the successful process prints the value of *i* at the time it was spawned. All other processes will eventually be scheduled to run, but they will probably find no input data on the input clists and go back to sleep. The entire procedure is repeated for every input line; it is impossible to guarantee that one process does not hog all the input data.

It is inherently ambiguous to allow multiple readers of a terminal, but the kernel copes with situation as best as it can. On the other hand, the kernel *must* allow multiple processes to read a terminal, otherwise processes spawned by the shell that read standard input would never work, because the shell still accesses standard input, too. In short, processes must synchronize terminal access at user level.

When the user types an "end of file" character (ASCII control-d), the line discipline satisfies terminal *read*s of the input string up to, but not including, the end of file character. It returns no data (return value 0) for the *read* system call that encounters *only* the end of file on the clists; the calling process is responsible for recognizing that it has read the end of file and that it should no longer *read* the terminal. Referring to the code examples for the shell in Chapter 7, the shell loop terminates when a user types control-d: The *read* call returns 0, and the shell *exit*s.

This section has considered the case of dumb terminal hardware, which transmits data to the machine one character at a time, precisely as the user types it. Intelligent terminals cook their input in the peripheral, freeing the CPU for other work. The structure of their terminal drivers resembles that of dumb terminal drivers, although the functions of the line discipline vary according to the capabilities of the peripherals.

10.3.3 The Terminal Driver in Raw Mode

Users set terminal parameters such as erase and kill characters and retrieve the values of current settings with the *ioctl* system call. Similarly, they control whether the terminal echoes its input, set the terminal baud rate (the rate of bit transfers), flush input and output character queues, or manually start up or stop character output. The terminal driver data structure saves various control settings (see [SVID 85] page 281), and the line discipline receives the parameters of the *ioctl* call and sets or gets the relevant fields in the terminal data structure. When a process sets terminal parameters, it does so for all processes using the terminal. The terminal settings are not automatically reset when the process that changed the settings *exit*s.

Processes can also put the terminal into *raw* mode, where the line discipline transmits characters exactly as the user typed them: No input processing is done at all. Still, the kernel must know when to satisfy user *read* calls, since the carriage return is treated as an ordinary input character. It satisfies *read* system calls after a minimum number of characters are input at the terminal, or after waiting a fixed time from the receipt of any characters from the terminal. In the latter case, the kernel times the entry of characters from the terminal by placing entries into the callout table (Chapter 8). Both criteria (minimum number of characters and fixed time) are set by an *ioctl* call. When the particular criterion is met, the line discipline interrupt handler awakens all sleeping processes. The driver moves all characters from the raw clist to the canonical clist and satisfies the process *read* request, following the same algorithm as for the canonical case. Raw mode is particularly important for screen oriented applications, such as the screen editor *vi*, which has many commands that do not terminate with a carriage return. For example, the command *dw* deletes the word at the current cursor position.

Figure 10.17 shows a program that does an *ioctl* to save the current terminal settings of file descriptor 0, the standard input file descriptor. The *ioctl* command

```
#include      <signal.h>
#include      <termio.h>
struct termio savetty;
main()
{
      extern sigcatch();
      struct termio newtty;
      int nrd;
      char buf[32];
      signal(SIGINT, sigcatch);
      if (ioctl(0, TCGETA, &savetty) == -1)
      {
            printf("ioctl failed: not a tty\n");
            exit();
      }
      newtty = savetty;
      newtty.c_lflag &= ~ICANON;      /* turn off canonical mode */
      newtty.c_lflag &= ~ECHO;       /* turn off character echo */
      newtty.c_cc[VMIN] = 5;          /* minimum 5 chars */
      newtty.c_cc[VTIME] = 100;       /* 10 sec interval */
      if (ioctl(0, TCSETAF, &newtty) == -1)
      {
            printf("cannot put tty into raw mode\n");
            exit();
      }
      for (;;)
      {
            nrd = read(0, buf, sizeof(buf));
            buf[nrd] = 0;
            printf("read %d chars '%s'\n", nrd, buf);
      }
}
sigcatch()
{
      ioctl(0, TCSETAF, &savetty);
      exit();
}
```

Figure 10.17. Raw Mode — Reading 5-Character Bursts

TCGETA instructs the driver to retrieve the settings and save them in the structure *savetty* in the user's address space. This command is commonly used to determine if a file is a terminal or not, because it does not change anything in the system: If it fails, processes assume the file is not a terminal. Here, the process does a second *ioctl* call to put the terminal into raw mode: It turns off character echo and arranges to satisfy terminal *read*s when at least 5 characters are received from the

terminal or when any number of characters are received and about 10 seconds elapse since the first was received. When it receives an interrupt signal, the process resets the original terminal options and terminates.

```
#include <fcntl.h>

main()
{
        register int i, n;
        int fd;
        char buf[256];

        /* open terminal read-only with no-delay option */
        if ((fd = open("/dev/tty", O_RDONLY | O_NDELAY)) == -1)
                exit();

        n = 1;
        for (;;)        /* for ever */
        {
                for (i = 0;  i < n;  i++)
                        ;

                if (read(fd, buf, sizeof(buf)) > 0)
                {
                        printf("read at n %d\n", n);
                        n--;
                }
                else        /* no data read; returns due to no-delay */
                        n++;
        }
}
```

Figure 10.18. Polling a Terminal

10.3.4 Terminal Polling

It is sometimes convenient to poll a device, that is, to *read* it if there is data present but to continue regular processing otherwise. The program in Figure 10.18 illustrates this case. By *open*ing the terminal with the "no delay" option, subsequent *read*s will not sleep if there is no data present but will return immediately (refer to algorithm *terminal_read*, Figure 10.15). Such a method also works if a process is monitoring many devices: it can *open* each device "no delay" and poll all of them, waiting for input from any of them. However, this method wastes processing power.

The BSD system has a *select* system call that allows device polling. The syntax of the call is

select(nfds, rfds, wfds, efds, timeout)

where *nfds* gives the number of file descriptors being selected, and *rfds*, *wfds* and *efds* point to bit masks that "select" open file descriptors. That is, the bit $1 << fd$ (1 shifted left by the value of the file descriptor) is set if a user wants to select that file descriptor. *Timeout* indicates how long *select* should sleep, waiting for data to arrive, for example; if data arrives for any file descriptors and the timeout value has not expired, *select* returns, indicating in the bit masks which file descriptors were selected. For instance, if a user wished to sleep until receiving input on file descriptors 0, 1 or 2, *rfds* would point to the bit mask 7; when *select* returns, the bit mask would be overwritten with a mask indicating which file descriptors had data ready. The bit mask *wfds* does a similar function for write file descriptors, and the bit mask *efds* indicates when exceptional conditions exist for particular file descriptors, useful in networking.

10.3.5 Establishment of a Control Terminal

The control terminal is the terminal on which a user logs into the system, and it controls processes that the user initiates from the terminal. When a process *opens* a terminal, the terminal driver opens the line discipline. If the process is a process group leader as the result of a prior *setpgrp* system call and if the process does not have an associated control terminal, the line discipline makes the opened terminal the control terminal. It stores the major and minor device number of the terminal device file in the *u area*, and it stores the process group number of the opening process in the terminal driver data structure. The opening process is the control process, typically the login shell, as will be seen later.

The control terminal plays an important role in handling signals. When a user presses the delete, break, rubout, or quit keys, the interrupt handler invokes the line discipline, which sends the appropriate signal to all processes in the control process group. Similarly, if the user hangs up, the terminal interrupt handler receives a hangup indication from the hardware, and the line discipline sends a hangup signal to all processes in the process group. In this way, all processes initiated at a particular terminal receive the hangup signal; the default reaction of most processes is to *exit* on receipt of the signal; this is how stray processes are killed when a user suddenly shuts off a terminal. After sending the hangup signal, the terminal interrupt handler disassociates the terminal from the process group so that processes in the process group can no longer receive signals originating at the terminal.

10.3.6 Indirect Terminal Driver

Processes frequently have a need to read or write data directly to the control terminal, even though the standard input and output may have been redirected to other files. For example, a shell script can send urgent messages directly to the terminal, although its standard output and standard error files may have been redirected elsewhere. UNIX systems provide "indirect" terminal access via the device file "/dev/tty", which designates the control terminal for every process that has one. Users logged onto separate terminals can access "/dev/tty", but they access different terminals.

There are two common implementations for the kernel to find the control terminal from the file name "/dev/tty". First, the kernel can define a special device number for the indirect terminal file with a special entry in the character device switch table. When invoking the indirect terminal, the *driver* for the indirect terminal gets the major and minor number of the control terminal from the *u area* and invokes the real terminal driver through the character device switch table. The second implementation commonly used to find the control terminal from the name "/dev/tty" tests if the major number is that of the indirect terminal before calling the driver *open* routine. If so, it releases the inode for "/dev/tty", allocates the inode for the control terminal, resets the file table entry to point to the control terminal inode, and calls the *open* routine of the terminal driver. The file descriptor returned when opening "/dev/tty" refers directly to the control terminal and its regular driver.

10.3.7 Logging In

As described in Chapter 7, process 1, *init*, executes an infinite loop, reading the file "/etc/inittab" for instructions about what to do when entering system states such as "single user" or "multi-user." In multi-user state, a primary responsibility of init is to allow users to log into terminals (Figure 10.19). It spawns processes called *getty* (for get terminal or get "tty") and keeps track of which *getty* process opens which terminal; each *getty* process resets its process group using the *setpgrp* system call, *opens* a particular terminal line, and usually sleeps in the *open* until the machine senses a hardware connection for the terminal. When the *open* returns, *getty execs* the *login* program, which requires users to identify themselves by login name and password. If the user logs in successfully, *login* finally *execs* the shell, and the user starts working. This invocation of the shell is called the *login shell*. The shell process has the same process ID as the original *getty* process, and the login shell is therefore a process group leader. If a user does not log in successfully, *login exits* after a suitable time limit, closing the opened terminal line, and *init* spawns another *getty* for the line. Init *pauses* until it receives a death of child signal. On waking up, it finds out if the zombie process had been a login shell and, if so, spawns another *getty* process to *open* the terminal in place of the one that died.

```
algorithm login              /* procedure for logging in */
{
        getty process executes:
        set process group (setpgrp system call);
        open tty line;               /* sleeps until opened */
        if (open successful)
        {
                exec login program:
                prompt for user name;
                turn off echo, prompt for password;
                if (successful)          /* matches password in /etc/passwd */
                {
                        put tty in canonical mode (ioctl);
                        exec shell;
                }
                else
                        count login attempts, try again up to a point;
        }
}
```

Figure 10.19. Algorithm for Logging In

10.4 STREAMS

The scheme for implementation of device drivers, though adequate, suffers from some drawbacks, which have become apparent over the years. Different drivers tend to duplicate functionality, particularly drivers that implement network protocols, which typically include a device-control portion and a protocol portion. Although the protocol portion should be common for all network devices, this has not been the case in practice, because the kernel did not provide adequate mechanisms for common use. For example, clists would be useful for their buffering capability, but they are expensive because of the character-by-character manipulation. Attempts to bypass this mechanism for greater performance cause the modularity of the I/O subsystem to break down. The lack of commonality at the driver level percolates up to the user command level, where several commands may accomplish common logical functions but over different media. Another drawback of the driver scheme is that network protocols require a line discipline-like capability, where each discipline implements one part of a protocol and the component parts can be combined in a flexible manner. However, it is difficult to stack conventional line disciplines together.

Ritchie has recently implemented a scheme called *streams* to provide greater modularity and flexibility for the I/O subsystem. The description here is based on his work [Ritchie 84b], although the implementation in System V differs slightly. A *stream* is a full-duplex connection between a process and a device driver. It consists of a set of linearly linked queue pairs, one member of each pair for input

and the other for output. When a process *writes* data to a stream, the kernel sends the data down the output queues; when a device driver receives input data, it sends the data up the input queues to a reading process. The queues pass messages to neighboring queues according to a well-defined interface. Each queue pair is associated with an instance of a kernel module, such as a driver, line discipline, or protocol, and the modules manipulate data passed through its queues.

Each queue is a data structure that contains the following elements:

- An open procedure, called during an *open* system call
- A close procedure, called during a *close* system call
- A "put" procedure, called to pass a message into the queue
- A "service" procedure, called when a queue is scheduled to execute
- A pointer to the next queue in the stream
- A pointer to a list of messages awaiting service
- A pointer to a private data structure that maintains the state of the queue
- Flags and high- and low-water marks, used for flow control, scheduling, and maintaining the queue state

The kernel allocates queue pairs, which are adjacent in memory; hence, a queue can easily find the other member of the pair.

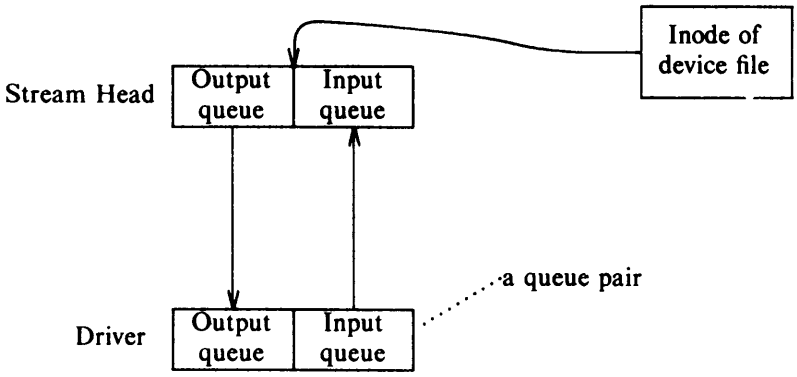


Figure 10.20. A Stream after Open

A device with a streams driver is a character device; it has a special field in the character device switch table that points to a streams initialization structure, containing the addresses of routines and high- and low-water marks mentioned above. When the kernel executes the *open* system call and discovers that the device file is character special, it examines the new field in the character device switch table. If there is no entry there, the driver is not a streams driver, and the kernel follows the usual procedure for character devices. However, for the first open of a streams driver, the kernel allocates two pairs of queues, one for the *stream-head*

and the other for the driver. The stream-head module is identical for all instances of open streams: It has generic put and service procedures and is the interface to higher-level kernel modules that implement the *read*, *write*, and *ioctl* system calls. The kernel initializes the driver queue structure, assigning queue pointers and copying addresses of driver routines from a per-driver initialization structure, and invokes the driver open procedure. The driver open procedure does the usual initialization but also saves information to recall the queue with which it is associated. Finally, the kernel assigns a special pointer in the in-core inode to indicate the stream-head (Figure 10.20). When another process *opens* the device, the kernel finds the previously allocated stream via the inode pointer and invokes the open procedure of all modules on the stream.

Modules communicate by passing messages to neighboring modules on a stream. A message consists of a linked list of message block headers; each block header points to the start and end location of the block's data. There are two types of messages — control and data — identified by a type indicator in the message header. Control messages may result from *ioctl* system calls or from special conditions, such as a terminal hang-up, and data messages may result from *write* system calls or the arrival of data from a device.

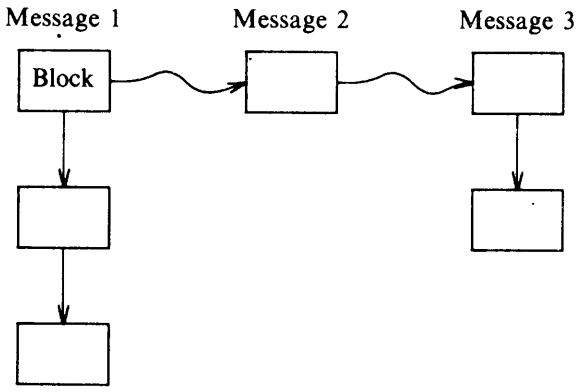


Figure 10.21. Streams Messages

When a process *writes* a stream, the kernel copies the data from user space into message blocks allocated by the stream-head. The stream-head module invokes the put procedure of the next queue module, which may process the message, pass it immediately to the next queue, or enqueue it for later processing. In the latter case, the module links the message block headers on a linked list, forming a two-way linked list (Figure 10.21). Then it sets a flag in its queue data structure to indicate that it has data to process, and schedules itself for servicing. The module places the queue on a linked list of queues requesting service and invokes a

scheduling mechanism; that scheduler calls the service procedures of each queue on the list. The kernel could schedule modules by software interrupt, similar to how it invokes functions in the callout table (as described in Chapter 8); the software interrupt handler calls the individual service procedures.

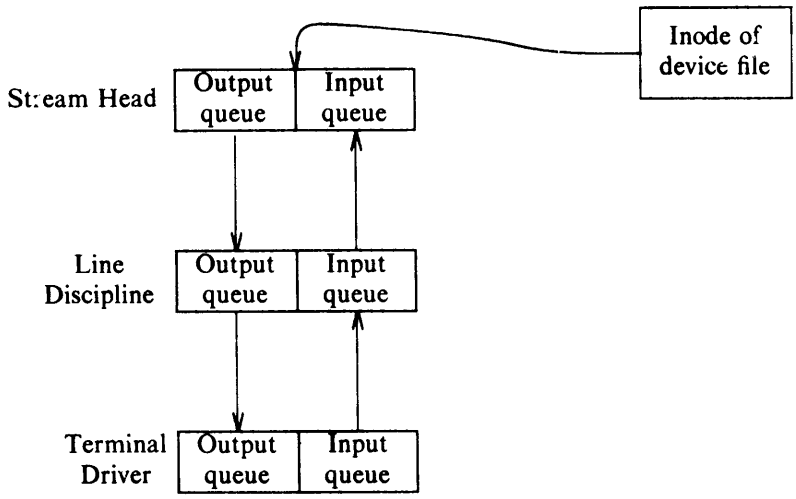


Figure 10.22. Pushing a Module onto a Stream

Processes can "push" modules onto an opened stream by issuing *ioctl* system calls. The kernel inserts the pushed module immediately below the stream head and connects the queue pointers to keep the structure of the doubly linked list. Lower modules on the stream do not care whether they are communicating with the stream head or with a pushed module: The interface is the put procedure of the next queue on the stream; the next queue belongs to the module just pushed. For example, a process can push a line discipline module onto a terminal driver stream to do erase and kill character processing (Figure 10.22); the line discipline module does not have the same interfaces as the line disciplines described in Section 10.3, but its function is the same. Without the line discipline module, the terminal driver does not process input characters, and such characters arrive unaltered at the stream-head. A code segment that *opens* a terminal and pushes a line discipline may look like this:

```
fd = open("/dev/ttyxy", O_RDWR);
ioctl(fd, PUSH, TTYLD);
```

where *PUSH* is the command name and *TTYLD* is a number that identifies the line discipline module. There is no restriction to how many modules can be pushed onto a stream. A process can "pop" the modules off a stream in last-in-first-out order,

using another *ioctl* system call.

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

Given that a terminal line discipline module implements regular terminal processing functions, the underlying device can be a network connection instead of a connection to a single terminal device. The line discipline module works the same way, regardless of the module below it. This example shows the greater flexibility derived from the combination of kernel modules.

10.4.1 A More Detailed Example of Streams

Pike describes an implementation of multiplexed virtual terminals using streams (see [Pike 84]). The user sees several virtual terminals, each occupying a separate *window* on a physical terminal. Although Pike's paper describes a scheme for an ntelligent graphics terminal, it would work for dumb terminals, too; each window would occupy the entire screen, and the user would type a control sequence to switch between virtual windows.

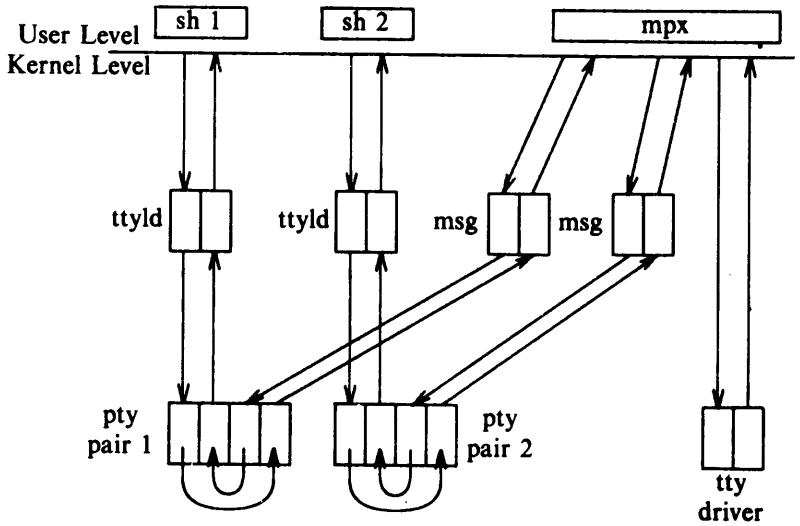


Figure 10.23. Windowing Virtual Terminals on a Physical Terminal

Figure 10.23 shows the arrangement of processes and kernel modules. The user invokes a process, *mpx*, to control the physical terminal. *Mpx reads* the physical terminal line and waits for notification of control events, such as creation of a new window, switching control to another window, deletion of a window, and so on.

```
/* assume file descriptors 0 and 1 already refer to physical tty */
for (;;)      /* loop */
{
      select(input);              /* wait for some line with input */
      read input line;
      switch (line with input data)
      {
      case physical tty:              /* input on physical tty line */
            if (control command)      /* e.g. create new window */
            {
                  open a free pseudo-tty;
                  fork a new process:
                  if (parent)
                  {
                        push a msg discipline on mpx side;
                        continue;      /* back to for loop */
                  }
                  /* child here */
                  close unnecessary file descriptors;
                  open other member of pseudo-tty pair, get
                                    stdin, stdout, stderr;
                  push tty line discipline;
                  exec shell;          /* looks like virtual tty */
            }
            /* "regular" data from tty coming up for virtual tty */
            demultiplex data read from physical tty, strip off
                        headers and write to appropriate pty;
            continue;           /* back to for loop */

      case logical tty:           /* a virtual tty is writing a window */
            encode header indicating what window data is for;
            write header and data to physical tty;
            continue;             /* back to for loop */
      }
}
```

Figure 10.24. Pseudo-code for Multiplexing Windows

When it receives notification that a user wants to create a new window, *mpx* creates a process to control the new window and communicates with it over a *pseudo-terminal* (abbreviated *pty*). A *pty* is a software device that operates in pairs: Output directed to one member of the pair is sent to the input of the other member; input is sent to the upstream module. To set up a window (Figure 10.24), *mpx* allocates a *pty* pair and *opens* one member, establishing a stream to it (the driver *open* insures that the *pty* was not previously allocated). *Mpx forks*, and the

new process *open*s the other member of the *pty* pair. *Mpx* pushes a message module onto its *pty* stream to convert control messages to data messages (explained in the next paragraph), and the child process pushes a line discipline module onto its *pty* stream before *exec*ing the shell. That shell is now running on a virtual terminal; to the user, it is indistinguishable from a physical terminal.

The *mpx* process is a multiplexer, forwarding output from the virtual terminals to the physical terminal and demultiplexing input from the physical terminal to the correct virtual terminal. *Mpx* waits for the arrival of data on any line, using the *select* system call. When data arrives from the physical terminal, *mpx* decides whether it is a control message, informing it to create a new window or delete an old one, or whether it is a data message to be sent to processes reading a virtual terminal. In the latter case, the data has a header that identifies the target virtual terminal; *mpx* strips the header from the message and *write*s the data to the appropriate *pty* stream. The *pty* driver routes the data through the terminal line discipline to reading processes. The reverse procedure happens when a process *write*s the virtual terminal: *mpx* prepends a header onto the data, informing the physical terminal which window the data should be printed to.

If a process issues an *ioctl* on a virtual terminal, the terminal line discipline sets the necessary terminal settings for its virtual line; settings may differ for each virtual terminal. However, some information may have to be sent to the physical terminal, depending on the device. The message module converts the control messages that are generated by the *ioctl* into data messages suitable for reading and writing by *mpx*, and these messages are transmitted to the physical device.

10.4.2 Analysis of Streams

Ritchie mentions that he tried to implement streams only with put procedures or only with service procedures. However, the service procedure is necessary for flow control, since modules must sometimes enqueue data if neighboring modules cannot receive any more data temporarily. The put procedure interface is also necessary, because data must sometimes be delivered to a neighboring module right away. For example, a terminal line discipline must echo input data back to the terminal as quickly as possible. It would be possible for the *write* system call to invoke the put procedure of the next queue directly, which in turn would call the put procedure of the next queue, and so on, without the need for a scheduling mechanism. A process would sleep if the output queues were congested. However, modules cannot sleep on the input side, because they are invoked by an interrupt handler and an innocent process would be put to sleep. Intermodule communication would not be symmetric in the input and output directions, detracting from the elegance of the scheme.

It would also have been preferable to implement each module as a separate process, but use of a large number of modules could cause the process table to overflow. They are implemented with a special scheduling mechanism — software interrupt — independent of the normal process scheduler. Therefore, modules