by Peter Collinson, Hillside Systems

MICHELLE FRIESENHAHN WILBY

Talking to the World

Most processes on UNIX acquire or emit data as byte streams. They open a file, read a stream of data until the source is exhausted, or write data until they have nothing more to say. Finally, they close the file. The open, close, read and write actions are system calls used by the process to affect the outside world. Essentially, a system call allows the process to run a piece of kernel code that undertakes some task. When the task is over, the kernel will return a result to the calling process. The result may be a value saying, "Yup I did that, and things went OK" or an error return saying, "Tough luck, there were problems." If the kernel is moving data into the address space of the process, then the return value can be the number of bytes that were moved.

System calls access routines in the kernel that do the work. These routines also mask the nasty truth about the real world by providing the process with an ideal model of how the outside world operates. For example, a process can write a 1K file by making 1,024 system calls that each send a single byte, or by creating a buffer in the process address space and making a single system call that tells the kernel to write 1,024 bytes. The UNIX model says that these two very different sets of actions should be equivalent because the process is writing a stream of bytes. In general, and there are exceptions, the act of making a read or write system call implies nothing about the data.

So when a process writes information, the data traveling between the process and the kernel is a stream of bytes. The order is significant, but the number of write system calls is irrelevant. The same thinking is true when a process is reading data: The read system call tells the kernel how much data the process is expecting. If the kernel has the correct number of bytes available, then it will move the data into the process address space. The kernel may have more data available and is expected to hang onto it until the process requests more. Alternatively, the kernel may not have the correct amount of data and can wait for more to come in, or can simply return what it holds. Notice that the model implies that the kernel buffers data for the process. In fact, managing the memory resources to provide data buffering is one of the most important jobs of the kernel.

If the process is dealing with a disk file, then the UNIX model is able to select the position in the file where the reading or writing is to start. Setting the position in the file is done with the seek system call, known as `lseek` because its argument changed from a 16-bit pointer to a 32-bit "long" integer when UNIX Version 7 was released on the PDP-11. The system call in Solaris is now called `llseek` because it deals with a 64-bit file pointer. The existence of the seek call doesn't break the byte stream model. The data is still a stream, but the programmer can affect the positioning of the read or write pointer in the stream.

Record-Oriented Behavior

I suppose the fact that I am discussing the byte stream model may not be surprising to you, because you are convinced that it's the only way the world works. Well, there are other schools of thought. Many systems use record-oriented I/O, where the act of writing creates a record that is a complete single unit of data. For example, a record could be a card image, 80 bytes of data stored in chunks. The act of reading will return a record, and if the process has not asked for all the bytes in the record, then some information will be discarded.

We see record-oriented behavior when UNIX deals with magnetic tapes, where the normal rules about byte streams have been pragmatically ignored. The original 9-track reel-to-reel tape storage systems used variable-length records written on the tape. The end of the record was marked by a special data block that the hardware recognized. Several modern tape-based systems still operate using variable-length records. UNIX needed to have a way of creating a record of a known size on the tape, and a pragmatic solution was provided by setting the size of the record equal to the number of bytes in the write system call. Later in this article, I'll discuss some other reasons for providing record-oriented I/O for tapes.

Each file on the tape should have records of the same size; otherwise there can be problems when the data is read back. If the program that is reading asks for too little data in one read, then some of the tape block will be thrown away. Each program dealing with the tape device needs to know how the tape interface behaves to ensure the integrity of the data.

Most of the programs that deal with tapes are actually special utilities written specifically for tape manipulation; for example, `tar`, or the original `dump/restore` programs. So the person writing the code knows that tape devices needed special handling. Every write to the output device means a new record on the tape, and when the tape is read back, the programmer can provide a memory buffer that is the maximum blocking factor on the tape.

Problems can arise when general-purpose programs are used with tapes. For example, you can use `cat` with a tape device, but you cannot guarantee what block sizes will be used. The `cpio` program has an option to set the blocking factor, and the option is designed to be used when `cpio` is dealing with tapes. However, it's up to the programmer to remember, or record, the blocking factor that is used by a particular data set.

UNIX treats its hardware devices as part of the file system. Each device is accessed through a "special" file, providing a name in the file system tree. The special file behaves like any other file on the system. It can be opened by quoting its name, data can be read or written, and the file can be closed. The special device is implemented by a kernel device driver, which provides device-specific routines that are called in place of the

regular file-based I/O routines.

There are two types of special files: *block* and *character*. The block special files are used for any device that is intended to be used as part of the file system. Block special files are usually disk partitions that can be mounted to make the single tree-structured file system that UNIX supports. The device drivers deal with the hardware and manage the blocks of data that form part of the file system.

It's usual for the hardware of disks to operate using direct memory access (DMA). A disk controller is given a command to move data between a point on its surface and the memory of the machine. The disk controller performs the command, reading or writing the memory with no intervention from the CPU of the computer to which it is attached. The CPU is notified with an interrupt when the transfer has taken place.

The hardware of the disk is a shared resource, and so is the file system that is written on its surfaces. It makes sense for all the processes on the system to share the pool of data buffers used to communicate with the disk. When several processes update the same file, perhaps a directory, they are really updating a single block or set of blocks held in the kernel memory. The data block in memory will eventually be copied out to the disk surface.

The pool of buffers managed by the disk drivers acts as a cache, storing data in the memory of the machine. The cache is a win for small temporary files that are written and immediately read. The files need never exist on the surface of the disk, they can remain in memory for their entire lifetime. The cache is also a win for files that have been read and whose contents are currently in memory when the same files are reread. There are many files, mostly directories, that are frequently read by processes.

UNIX also implements "read ahead" of blocks into the cache. Most processes read files sequentially, and the kernel will queue a read of the next block of a file before it is requested by the process. In most circumstances, read ahead is a win, improving processing speed because, while the process is munching on the data that it has just read, the system is getting on with loading the next block.

Block device drivers are generally called by the file system management code on behalf of processes that are accessing files in the file system. When a process writes data to a file, it will travel from the address space of the process into a kernel buffer before being written to the disk. Data that is read will also move from the disk into the kernel and from there to the process that originated the request. The cost of this copy is outweighed by the sharing of data that occurs in the cache, and the ability of the disk driver to control access to the disk surface.

Character Special Files

Character special files primarily support all the other devices on the system, for example, terminals or printers. The kernel imposes the byte stream model on the devices, so a stream of data is sent from a process to a printer; and streams of data are traveling in both directions between a process and the terminal line. Generally, apart from terminals, only one process deals with a particular character device at any one time.

Incidentally, in his 1977 paper in the *Bell System Technical Journal*, Ken Thompson says, "while the term *block I/O* has some meaning, *character I/O* is a complete misnomer." He would prefer the term *unstructured I/O* to be applied to character devices and *structured I/O* to block devices. If you hold fairly tightly onto the idea that block I/O supports the file system and everything else is done using character I/O, then you will not get terribly confused.

Character special devices are also used to provide "raw" access to disks and other DMA devices. Raw devices allow data to be moved directly between the address space of the process and the peripheral, avoiding the kernel copy that is demanded by the block special devices. This allows faster data copying for the raw devices and the DMA of large arbitrarily sized data blocks. Tape handling programs need both features.

There are often restrictions on the size of the data that is moved. The data request must fit in with the hardware, typically moving data in units of 512-byte blocks to and from disks. A read DMA request for tape devices with variable-length records must fit the block of data that is to be read; a write request will cause one record to be written by DMA directly to the tape, creating a tape block of the size of the write call. Thus, there are good hardware reasons for the model of tape activity that I outlined earlier.

Raw devices allow programs to deal with a whole disk or partitions within the disk as sequences of blocks, permitting programs like `df` or `dump` to operate outside the constraints of the normal file system. Most programs that deal with the file system as an entity operate with raw devices. One example is `fsck`, which checks and repairs the file system. Some people run this program on a live system overnight to "see if the file system is OK." Panic can set in when errors are reported. The errors are illusory, and go away when the machine is taken down to single-user mode and `fsck` is run again. The false errors occur because data on the disk which `fsck` is reading via the character device is not synchronized with the live file system, part of which is stored in the buffer cache and is accessed independently via the usual block device driver. Running `fsck` on a live system is a waste of time and nervous energy.

Character Peripherals

As I said earlier, character special devices were designed to deal with peripherals like terminals and printers. Printers are generally attached to the machine by a Centronics parallel interface. The device driver for such an interface is simple. Its job is to take characters from the process and throw them down the line to the printer one by one. The character special device implements a byte stream interface for the processes that call it, so you can use a simple program like the `cat` command to send data to be printed. These days, we wrap the printing command with a complex line printer spooling program, but the central routine of this spooler will simply open the file to be printed, read the data in chunks and write the output to the printer.

The device driver for terminal lines was always considerably

more complex than a printer driver. Terminal device drivers are complicated pieces of software that are configurable from user processes. The hardware is generally simple to drive, but we expect the line to function in many different ways: We can attach a hard-wired terminal, a modem for dial-in, a modem for dial-out, a printer, a mouse–the list of uses goes on and on.

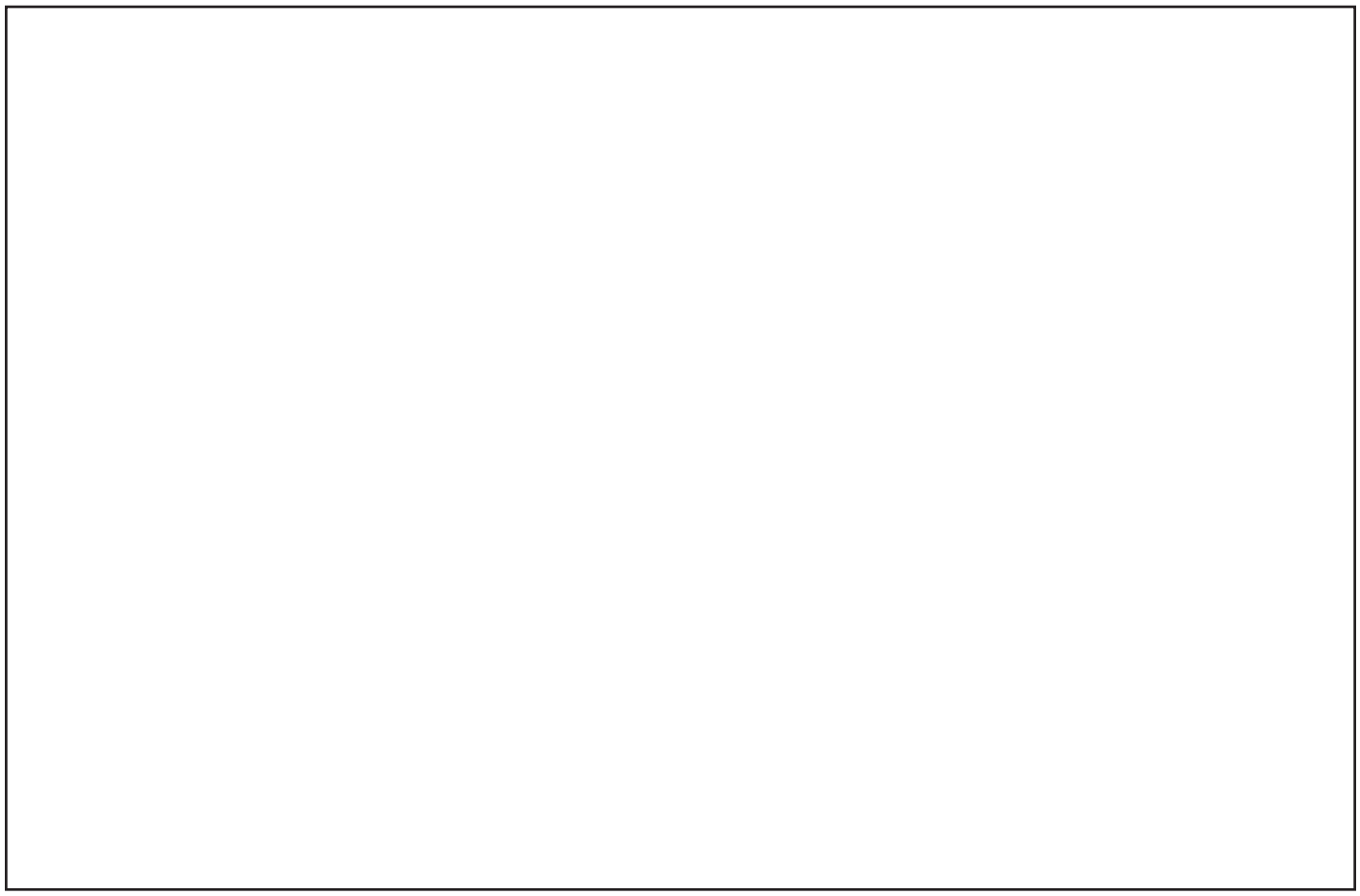
A terminal line is controlled by a small block of data that the kernel interrogates to select different processing options when characters are received from the outside world or sent by a process to output half of the line. In the early UNIX systems, there was a need to set values in this control block, and this was done by the `stty` system call. The values in the block were returned to a process using the `gtty` call. (Of course, the name `stty` is preserved in the command used today to set up and print values from the terminal interface.) Later, it was realized that the `stty`/`gtty` system calls were too specific. The `ioctl` call was born, providing a general-purpose kernel interface to set device characteristics for all character special devices.

The `ioctl` call is used to establish many aspects of the terminal interface. The stream of text flowing to the terminal can be processed by the code in several ways. For example, UNIX stores the end of a line as a single character (the newline character; actually, the ASCII line feed character), and the interface provides automatic conversion to the carriage return and line feed that most devices need to move down to the next line.

The text flowing into the system from the user can also be processed. For example, it's possible to stop the interface from echoing characters, used to suppress your password being displayed when you log in. Also, in the default state, the interface will store all the text that is typed and will only pass it to a process waiting in the read system call when the user hits return. The interface processes the stored data, applying the control characters that delete a character and delete a line, ensuring clean data is sent to the process. The interface will also look for control characters that mean interrupt (often control-C), end of file (control-D), suspend (control-Z), and will cause the appropriate signal to be sent to processes attached to the terminal.

It's common for a process to wish to change the default action of the interface. For example, the `vi` program wants to see any character the user types immediately after the key is pressed and will disable much of the input character processing. If the process gets things wrong, or dies unexpectedly, then the terminal can be left in an unusable state.

In today's Solaris systems, terminal functionality is provided by a Streams module called `ldterm`. The Streams-based device drivers were originally designed by Dennis Ritchie, who saw a need for a clean way to glue parts of the kernel together. Streams modules are pieces of code that have well-defined input and output interfaces. Messages containing information flow in both



directions through the module. A Streams module can be arbitrarily connected to any other module, forming a processing stack. Each module will perform its own characteristic processing on the messages passing through. Data from a user process is translated into a message, which is then passed down into the kernel via various modules and out the hardware driver. Inbound data is also translated into a message and passed up through the set of modules on its way to the user process. Streams modules can be used with any appropriate Streams hardware driver, but I will concentrate on terminal lines.

If we look at one of the serial lines on a Sun, the hardware driver is a Streams driver called `zs`. If we connect to this, we can send raw character data in both directions. If we want to give the interface the standard terminal functionality, we push the `ldterm` module onto the line. We probably also want to maintain `ioctl` compatibility with older programs, and can push the `ttcompat` module to handle conversions from the old format to the new. This loading of modules happens automatically, usually before we open the line.

Streams allows us to change the use of a terminal line. For example, if we wanted to run the IP Point-to-Point (PPP) protocol over the line, we would replace the modules that handle regular terminal I/O with the stack that handles the PPP protocol. The `strconf` command can be used to find out what Streams are being used on your terminal line. For example, on my terminal emulator running under X, I get output like this:

```
ttcompat
ldterm
ptem
pts
```

showing four Streams modules in use. `ttcompat` and `ldterm` provide the standard terminal functionality. `ptem` and `pts` handle the client side of the pseudoterminal. When you use X, or log in from `telnet` or `rlogin`, there has to be a way of presenting the programs you run with a system interface that is identical to the terminal interface. However, the data from these programs that is written to the terminal has to be turned into a TCP/IP data stream that perhaps connects internally to the X server, or may be sent over the network to a client handling the `telnet` or `rlogin` protocols.

The pseudoterminal driver provides the necessary functionality. It consists of two special devices in the file system. The devices are connected, so if you push data in one end it pops up at the other. The client end, discussed above, emulates a terminal. It passes data down into the kernel using the `pts` device driver. The data pops up again on the master end of the connection translated into a series of messages. These messages are translated into the necessary protocol and sent on their way to the server. Data from the server travels the reverse route, ending up in the processes that are being run on the machine.

Further Information

The book *The Magic Garden Explained*, by Berny Goodheart and James Cox, published by Prentice Hall, ISBN 013-098138-9, describes the internals of UNIX System V, Release 4. I suspect that the Sun Solaris kernel has diverged from the basic release in several ways, but this book provides enough clues about where things started for you to work out what is going on in your own system.

Streams have made device drivers into complicated beasts. For simpler explanations, take a look at *The Design and Implementation of the 4.4 Operating System* by Marshall Kirk McKusick, Keith Bostic, Michael Karels and John Quarterman, published by Addison-Wesley, ISBN 0-201-54979-4. This is a revised edition of the 4.3BSD book.

To return to UNIX roots and see the complete operating system code with a clear explanation of how device drivers work, read *Lion's Commentary on UNIX, 6th Edition* by John Lions, recently republished by Peer-to-Peer Communications, ISBN 1-57396-013-7. This is a reprint of the seminal book that inspired many of us who started with UNIX all those years ago.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.