by Peter Collinson, **Hillside Systems**

Device Independence

Computers were once deeply expensive objects, ensconced in their own cathedrals in which their users worshipped. Well, I suppose the users didn't actually get into the cathedral often. Your deck of cards was conveyed to the appropriate entry device by a machine-servant–a priest known as an operator–whose job was to tend to the needs of the machine. The operator returned to you a wad of large pieces of paper containing your results (more likely they stuffed it into an oversize pigeonhole somewhere). If you were lucky, you could move onto the next problem; if not, you needed to edit the cards and try again.

Many of these early batch operating systems were designed to enable their programs to be run without any knowledge of their actual input or output devices. The programs were "device independent." This was important in busy machine rooms where the card readers and printers were bottlenecks. You wanted to be able to feed cards into any reader that had space, and take output from any printer. This became more important with later systems when cards and output were spooled to disk. Even though the program was receiving 80-column card images, or printing formatted output, the actual data was stored on some disk on the system until it was needed.

UNIX was designed with this legacy, the notion that programs should be device independent. Device independence is achieved by making assumptions about how programs behave. The most basic assumption is that all programs read and write streams of data. It's generally true to say programs that manipulate information will read some information, do some work and output the answer. We make the generalization that when you read from any device you're returned a stream of bytes, and, when you want to output to a peripheral or file, we also make the program write a stream of bytes. We assume nothing about the data itself, it is just moved, byte by byte, from one place to another.

So streams of data are typeless, and UNIX makes no assumptions about their contents. Actually this is true about files too. The system doesn't *know* which files on its disks contain binary data and which contain text. It doesn't know that `file.gif` contains a bitmap picture. File names are only used as tags for humans. Typeless files are generally a good thing because the programmer is not constrained to a set of operations the system designer thought were appropriate for that particular file type. It's proved to be an inconvenience when we want to display different icons for different files in visual file manager programs.

Data Streams

Any program that reads or writes data streams makes assumptions about how the streams will behave. The program is at liberty to read as much or as little of its input stream as it likes. However, the program also has to know that when reading data, it may be supplied with less information than it requested. An output data stream means that the program is free to write data in huge chunks or in single bytes. The operating system kernel will cope with converting that erratic output into some suitable form for sending to the target device.

Of course, real-world devices rarely deal in data streams. When you write a file to disk, you typically expect to write 512 byte blocks of data. If a program is writing a data stream to a disk, it's allowed to send irregular-size chunks. We must arrange for the kernel to retain the data until it has a filled block that can be written to the disk. We need to provide some code in the kernel that presents a data stream interface to the program.

Similar actions are needed to provide the program with the illusion it is reading a data stream. If the program is reading from a disk file, then it can ask to be sent small amounts of information and the operating system needs to store the remaining data until it is needed. Conversely, if the program asks for huge amounts of data, then the operating system needs to read several blocks from the disk before it can return control to the calling program having satisfied the read request.

At some point the file will end and the program needs to be told the stream has finished. UNIX tells programs they have reached the end of a file by returning zero in response to a read request. Typically, a UNIX program will process a file by using a loop to read a fixed-size block of data. When taking data from the disk file, the kernel will return exactly the number of bytes requested until the end of the file is reached. The program is always told how many bytes have been read on any successful request. The last read from the file may return some number of bytes that partially fill the fixed-size block. Thus, in the loop that processes the data, the program can never assume it has received the number of bytes it requested. The last block is likely to be smaller and the program must always examine the byte count supplied by the kernel. The next read request will return zero, which the program will recognize as the end of the file.

Serial Devices

The data stream model used for disk files also needs to map onto other peripherals. Many output devices, such as printers or terminal screens, already expect to be sent a stream of bytes. All devices are very slow in comparison to CPU speeds, and any program writing to the device will generate bytes much faster than they can be sent to the peripheral. The kernel will never have enough buffer space to store masses of data. So when the program writes a huge amount of data, the kernel will arrange for the program to wait until some bytes have drained out of the system and onto the peripheral. The trick is to wake the program before the kernel buffers have completely emptied so there is always data ready to be sent to the output device.

Also, the operating system may need to do some work to translate internal sequences used in UNIX into other characters the device may need. For example, the device may not support tab characters, so the kernel will need to translate a tab character into an appropriate number of spaces. More commonly, there is the problem of new lines. In disk files, UNIX uses a single character (line feed) to represent the "end of a line." When the output driver sees the end-of-line indicator, it will typically need to translate it into two characters: carriage return and line feed. For a physical printing device, carriage return makes the printing head return to the start of the current line, and line feed moves the paper up one line. Splitting the new line action into its components allows for overprinting and also gives time for the printing mechanism to settle down. Electronic devices like VDUs with no printing head or paper mimic this behavior.

Incidentally, there is no representation standard for the end of a line. MS-DOS descendants store both a carriage return and a line feed character at the end of each line, while Macintosh operating systems use a single carriage return. These differences may cause problems if you copy files between systems without doing the correct conversion.

Terminals, Keyboards and Screens

The most common serial device is a keyboard attached to a terminal–and terminals present a more complex set of problems. We need to map somewhat erratic human typing behavior into a stream of data that is sent to a running program (a *process*) when it asks for data with read request. At the same time, we will be presenting the user with output from processes. Of course, these days, you're typing into a window on a screen, but the mechanisms you're using are largely unchanged from the original interfaces designed to support terminals attached by a serial line to the computer.

To get a handle on the complexity, let's start at the beginning and look at normal behavior. We'll assume that a process has decided to read information from your keyboard, has issued a read request and is waiting for something to happen. When you type a single character on the keyboard, it will be echoed on the screen. The character has left your keyboard, traveled into the machine and been sent automatically by the kernel into the code that transmits data to your screen. Note that the process waiting for some characters might want to inhibit the echoing of characters and we need to provide it with the ability to do that. There's instant complexity here. When typing something into the machine, you are expecting to see the character coming back out to you. The input and output halves of the terminal interface are inextricably linked.

We now have the character you typed sitting in the kernel. We could decide to return the character to the waiting process immediately, telling it that it's got one character from you. In fact, a process can set up the terminal interface to send every character as it is received. This mode of operation is called "raw," and is used by visual editors, and these days, by shells, too. Normal working is usually called "cooked" mode, because it's not raw. If you are running a program that doesn't want to have fine control over the terminal interface and uses cooked mode, then the kernel will retain all your characters until you hit the return key. The kernel will also perform line editing, allowing your chosen delete key to remove characters from the stored data (which again is more complex than it seems–consider erasing a tab character that has installed a variable number of spaces on the screen). Meanwhile the program is doing nothing. It won't wake up until you have completed a line and hit return.

On early machines that ran UNIX, cooked mode was an important part of making the system appear to work quickly. Most input was done in cooked mode and most of the time the machine was waiting for a user to hit return. Each user received a fast response to their typing because each single character was echoed back quickly by the kernel.

Raw mode means the system has to do much more work, waking up processes and passing single characters in both directions between the processes and their terminals.

When you hit a return in cooked mode, the kernel will send all the stored bytes to the process, waking it up and supplying it with a complete line of pre-edited data. The process will start to run and deal with the information that you have typed. What happens then? It would be possible to implement the interaction so that the kernel will only accept characters when a process has issued a read request and is waiting for some input. Users of PCs will know that this is very annoying. The machine appears dead while data processing takes place. Control is finally returned to the user who is able to type more text.

On UNIX, the terminal interface will continue to accept characters even if there is no process ready to take the data from the kernel buffers. If the kernel input buffers get full and still no process has appeared to read the data, then the kernel will "beep" at the user and refuse to store any more characters. Incidentally, in cooked mode, the kernel continues to return a line of data to any calling process, even if it's got more than a line stored in its buffers.

The keyboard interface also supports a great many configuration options. First, it understands a set of control characters mapping onto UNIX signals that are sent to processes, which are talking to the terminal. Second, you can force the terminal interface to send an end-of-file indicator, which you will recall is a read request that returns zero bytes. Control-D is usually used to mean this. I still use this character to exit from shells rather that typing "logout" or "exit." When I type Control-D, my shell sees an end-of-file indicator and knows that it's time to exit.

UNIX also has support for delays that can be automatically inserted by the terminal device driver for certain character sequences. These delays are of less importance these days because we all use fast electronic devices to see the output from our computers. But the delays are there to provide support for devices that take time to settle when the printing head zooms back across the paper and the platen moves it up one line.

Finally, you can make the terminal interface react to a pair of characters that control data flow, known as the XON/XOFF protocol. The output interface will stop sending characters when the input side receives a Control-S character, and will restart when a Control-Q arrives. The input side can also use the protocol to control its own buffers, sending a Control-S when they are getting full and a Control-Q when they have drained to an acceptable level.

A flow-control protocol is needed because terminal lines have become a common way of connecting random serial devices to a computer. Serial devices will range in complexity from other computers down to the humble mouse. Serial lines were often used to connect computers together in the days before local area networks were invented. Of course, using serial lines to carry network packets is still done and many people use serial lines connected to modems to carry Internet traffic today.

I've managed to stray a long way from the simple data stream model that I talked about above. Data streams are used by naive programs that will not change any settings on the terminal interface. Programs that use the configurable options of the interface know they are dealing with a bidirectional serial line and are programmed appropriately. Such programs are not seeking device independence. The real trick is that the terminal device interfaces `\fIcan\fP` support data streams, and do so in the default case.

The File System

When Ken Thompson designed the UNIX file system in the early 1970s, he created "special files"–names in the file system address space that map onto physical devices. This completes the illusion of device independence on UNIX. All devices are addressed as if they were files. If you want to save the output of a program on disk, you simply point the output channel of the program to the file. If you want to print the output, then you point the output channel of the program to a special file that connects to a device driver for the printer. The device driver supports the data stream model and prints any data it is sent.

The key here is all programs are coded with the same set of system calls that read or write data streams. If you change the output destination of the program to point to a printer's special file, then the same system call in the same compiled binary will send output to that peripheral. The kernel recognizes the output file is special and calls the appropriate device driver routines when the process asks to write some data.

Also, tying a device to name in the file system makes it easier for programs that are not device independent to acquire access to the device. Such programs use the same set of system calls that are normally employed to deal with regular files, except they will use some extra system calls to configure the device interface.

Of course, special files are also subject to the same access control rules that apply everywhere else on the file system. By setting permissions appropriately on the files themselves, we can permit or deny access by individuals or groups on the machine. So, for example, it's possible for a user to use a standard data stream copy command like `cat` to send data to a printer. However, if we have a line printer spooler, we don't want users to circumvent the order of print jobs in the spooler queue. Rather than allowing anyone to write to the printer, we ensure that only the spooler system has access. We make use of the standard UNIX ideas of ownership and set appropriate file access permissions on the printer special file.

Should we need to, we can give known programs access to files using the UNIX notion of the `setuid` program. A `setuid` program has a bit set in its file description information. The bit tells the kernel that when the program is run, it should have the same access rights as the owner of the file in which the command lives. Thus, we can give a particular program special rights to access a file. A normal program will be run with the access rights set to the user who is running the command. Placing a `setuid` on a program's executable file is often used to allow that program to access devices that wouldn't normally be accessible. For example, the `df` command tells you how much free space there is on a disk. To return accurate results, the program needs to delve into the special device used to address the disk. However, having the disk device open to all would circumvent system security, so the `df` command is `setuid` to permit it to look at the parts of the disk structure, obtaining the information that it needs.

The UNIX file system model means that "everything" on the computer can be addressed and accessed as a name in the file system address space. The model has been extended by successive sets of programmers. For example on Solaris, we have the `/proc` file system that places a name in the file system for every process that's running on the machine. Each running process is represented as a directory, while files within the directory supply information about the process.

The `/proc` file system provides a simple mechanism for one process to inspect the memory of another. It's not unusual to wish to have this ability. In fact, we normally want to prohibit any process from accessing another's address space for both security and program safety reasons. Before the invention of the `/proc` file system, processes used several ad hoc methods to look at the address space of other processes. The methods employed a special file that mapped onto the virtual memory maintained by the kernel and meant that the programs doing the work of accessing information needed operation system and processor specific knowledge.

So when `ps` prints the command name or other information about processes on the system, it does so by accessing the `/proc` file system and not by delving messily into the memory of another running process. Incidentally, the ownership of files in `/proc` is also used to prevent the file system from being used as a back door by me to look at your programs.

Finally…

One of the reasons UNIX is surviving is because it implements simple models of how computing should be done. The models are easily understood and make it easy to write UNIX programs. Because the models are simple and coherent, it's also easy to see why and where you need to break away from the model and do something special. I suspect that UNIX is also surviving because it's had a tradition of adapting to whatever needs are thrown at it. "Never having to say no" is actually a good recommendation for anything.

PS. I must apologize to my readers (and also the brace of Jeffs Copeland and Haemer–see Page 57) for failing to make any mention of "bazaar" in this article. I did try, honest. But, I did manage to get a "cathedral" or two into the text.

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`.