The Linux Terminal System

Andy Goth

Fall 2004

1 Terminal system overview

The Linux terminal system is designed around the traditional model of a dumb terminal connected via serial line to a mainframe running an application. The application's input and output are connected to a tty device node on the mainframe, and all reads from and writes to said device node are handed by the kernel reading data from or writing data to the serial line. At the user's end, the dumb terminal sends keystrokes to and displays characters received from the mainframe, completing the loop between user and application. [SM, p166]

Few modern computer systems actually operate this way. For the common case of Linux running on a PC, both the dumb terminal and the serial line are emulated in software. Input data comes from the keyboard and output data goes to the VGA. But the full generality of the traditional model has been retained because most actual setups are adequately described as a subset thereof, and it offers a surprising amount of flexibility.

It turns out that additional types of emulation besides the above are quite useful. Most interesting is the pseudo-tty system in which the dumb terminal itself is replaced with a second, master application. Communication between the master and the slave is performed through a /dev/pty device node, which can be configured exactly like a real /dev/tty. This is crucial for many programs, such as xterm, GNU screen, ssh secure shell, and VPN-over-pppd.

In addition to the "official" terminal system, Linux supports several other user interface systems: frame buffer, X11, and VNC. The frame buffer system is implemented directly in the Linux kernel, whereas X11 and VNC are applications available on a variety of platforms. But this document only mentions these in passing— its focus is the textual terminal system.

Furthermore, Linux supports a wide variety of methods for interfacing applications with other components of the user's workstation, such as removable media, speakers, joysticks, printers, etc. But these are beyond the scope of this document.

2 `stdin` and `stdout`

Following the lead of the classic Unix tools (`cat`, `grep`, `sed`, ...), most small applications do not concern themselves with any details of the terminal system. Even more advanced programs like `gdb` don't need to deal with any terminal API— simply reading from `stdin` and writing to `stdout` is sufficient.

While this may seem like an extremely limited interface, it's all that's needed for simple programs and is in fact a great boon to complex applications: Complex apps can be be divided into suites of single-purpose utilities chained together via sockets and anonymous pipes. Such utilities are relatively easy to develop and debug and can more readily be used individually or in ways unforseen by the original developers

A comprehensive user interface can be developed using more advanced terminal control techniques (or X11), and even though it may appear to the user to be a single, large application, it is actually just another single-purpose utility in the toolchain: Its purpose is simply to interface the user with the various pipelines comprising the overall system [ESR, p15].

And, naturally, the components in such an architecture are interchangeable. The most visible consequence to the user is the ability to select from a variety of frontends.

As an example, my preferred integrated development environment really isn't "integrated" at all: Rather, it consists of existing development tools (`cc`, `gdb`, `lint`, ...) plus a text editor/source browser capable of calling them (`cc`) or interacting with them (`gdb`) and providing easy access to the results.

3 Escape sequences and function keys

In addition to printing characters and spaces, basic dumb terminals supported a handful of control characters for feeding paper and moving the print carriage. Glass teletypes added the possibility of full-screen displays, changing already-printed characters, and using color and other special effects. These additional features were made accessable through use of the special "escape" character (`ESC`) followed by sequences describing the function to be performed. For example, if $\star$ represents `ESC`, then $\star$`[2J` means "clear the screen" on many terminals.

Simiarly, new keys were added to terminal keyboards, but since the character set didn't support them directly, they were encoded as another form of escape sequence interpreted by the application. On my system, the "↑" key is encoded as $\star$`[A`.

Naturally, terminals didn't all agree on capabilities, escape sequences, or function keys, causing a major headache for programmers of terminal-oriented applications. A couple standards eventually emerged, the best known of which being "ANSI" (ISO 6429/ECMA 48/ANSI X3.64). Most terminal emulators nowadays provide a superset of vt100.

4 $TERM and termcap/terminfo

The $TERM environment variable is used to track the current terminal type. For instance, when using the Linux console (vt100 emulation), $TERM is "linux". Programs such as telnet and ssh must be careful to set $TERM in the remote environment so remote applications will know what sort of terminal control sequences to generate to correctly interact with the user's terminal.

Historically, terminal-oriented programs read /etc/termcap to map from values of $TERM to lists of available terminal capabilities and their associated escape sequences. As the number of terminal types grew, /etc/termcap became unmanageably large and has been superseded by the terminfo database.

My system's terminfo database has 2,344 entries spread over 42 subdirectories of /usr/share/terminfo for a total of 2,250,526 bytes of compiled data. The equivalent /etc/termcap might be three times that in size or more. Repeatedly searching through such a large, unindexed file would be quite difficult for humans and programs alike.

Terminfo provides not only an indexing mechanism (multiple files in subdirectories), but it also supports both compiled and human-editable forms of the terminal capability files, which can be converted from one form to the other using the tic command.

But with terminfo alone, writing terminal-oriented programs is still exceedingly difficult and repetitive, just as it was with termcap. For example, there's a good deal of complexity involved in locating the terminal capability file and converting it into structured data, and duplicating this code in every terminal-oriented program would be a serious waste of time, effort, and RAM.

5 ioctls and termios

Unix systems, especially Linux, arrange for as many things as possible to be accessable through a character-based stream interface usable with read() and write(). This unification is at the heart of Unix's tremendously powerful system of piping and redirection.

Unfortunately, many important operations do not map to the stream paradigm. Unix and Linux handle this problem with what are known as ioctls. An ioctl performs miscellaneous operations on file descriptors identifying already-opened files, nodes, anonymous pipes, devices, sockets, and so on.

Terminals have ioctls for adjusting attributes such as flow control, data rate, stop bits, CR/LF conversion, delays, echoing, and control character definitions.

These ioctls are clumsy and nonportable to use directly, so the termios system was developed to provide a standard low-level terminal control interface. All terminal settings are described in a struct termios control block, which users and applications can query, modify, and commit via the stty program or the tcgetattr() and tcsetattr() calls. [WRS, p344]

6 Canonical and non-canonical modes

The Linux terminal has two primary modes of operation: canonical and non-canonical mode. These names are just convenient terms to cover combinations of terminal settings, but they're useful to talk about. Canonical and non-canonical modes are also known as cooked and raw modes, respectively. Details about what settings these terms refer to can be found in the `stty(1)` and `termios(3)` man pages.

Programs that don't do any terminal control of their own generally assume the terminal is already in canonical mode, but as they're not really concerned so they could also be connected to anonymous pipes or files and will still function correctly. In canonical mode, the terminal itself (or, in the case of Linux, the in-kernel emulation) manages line editing, so the user can use backspace and send entire lines rather than individual characters. I presume this was originally a feature built into relatively advanced terminals to save bandwidth and provide a more user-friendly interface. Now it just means that programs don't need to worry about handling backspaces.

Unfortunately, canonical mode's line editing system is inadequate. It doesn't support nondestructive backspace, history browsing, tab completion, or many other in-demand features. Besides, not all programs have line-oriented interfaces. For programs that need it, non-canonical mode gives fine-grained control over terminal parameters, most importantly the timeout and the minimum number of characters in a completed read. In non-canonical mode, programs are themselves responsible for line buffering and echoing. Of course, this adds complexity, but the GNU readline library provides a handy wrapper, making it easy to write powerful line-based user interfaces.

7 curses, ncurses, and S-Lang

The curses library eliminates a great deal of the complexity inherent in dealing with widely-varying terminal types and capability sets. It internally handles the termcap database and the `$TERM` variable. It makes ioctls and termios calls to set canonical and non-canonical modes. It decodes function key sequences. And it provides a handy standardized function call interface to programmers. Curses makes full-screen programming easy, at least relatively so.

Ncurses is "new curses". It adds support for terminfo plus utilities for managing the database. It also supports advanced features like terminal resizing and probably a great deal else. Internally it attempts to calculate the optimal series of escape sequences to get the terminal to update as quickly as possible.

S-Lang's C library provides terminal interface functions that serve as an alternative to curses and ncurses that some programmers find more attractive. Originally, Linux's `dosemu` program used ncurses but at some point switched to S-Lang, resulting in simplification of the display code and enhanced performance.

8 Modern terminal emulators

Genuine terminals are less popular these days due to the availability of cheap computers, which either can serve as thin clients or can run both the applications and the user interface. So terminal emulator programs are used to provide compatibility with the existing terminal interface.

9 Linux vt100 emulation

Linux emulates a superset of vt100. It provides extra sequences for things like changing the screen palette. It also has extra ioctls for changing the font and other miscellaneous settings.

10 GNU Screen

GNU Screen emulates a superset of vt100. It provides terminal multiplexing, whereby multiple virtual screens can be displayed within a single physical screen. This is essential for using classic Unix dialup shell accounts, where there is only one terminal connection.

Screen also allows for keeping its applications alive when it loses its connection to the physical terminal. It supports multiple physical terminals at the same time, which is useful (or at least novel) for pair programming.

11 Xterms

Most X terminal emulators provide a superset of vt100. Additionally, the official `xterm` program emulates a large number of other terminal types, including graphics terminals. In fact, I am using it at this moment to write this document.

12 Terminal hacks: VPN over pppd

A trick to implement a Virtual Private Network serves as an example of the surprising flexibility of the Linux terminal system, of the way it finds utility in unexpected places.

pppd is typically used to connect to a remote host's network, such as that of an Internet Service Provider, over a serial link. The chat program first dials the modem and executes the login script. Then, once the serial link is established, it spawns a copy of pppd whose input and output are connected to the serial device, and pppd itself creates a routable, configurable network device linked to the remote host. But since pppd itself doesn't care where its input and output are connected, it's quite possible to instead connect it to a pseudo-tty. One very useful possibility is to connect to a remote host using ssh, and on each host run pppd connected to the pseudo-tty provided by ssh. Then configure each system

to route network packets through the ppp devices created by pppd. This is a very effective virtual private network implemented using only software already available on most Linux-based systems.

13 Conclusion

Ironically, the Linux terminal system is lucky to have the burden of supporting thousands of disparate terminals in addition to hundreds of terminal emulators and terminal-like applications. It was able to inherit an existing, working terminal system and retain compatibility with all existing programs. While complex at heart, wrapper libraries are available to unify the interface.

But the Linux terminal system doesn't support the full range of hardware available at modern workstations, so it is augmented by programs like X11, VNC, aRTs, esound, NAS, and others. It would pay to pull together all these various programs to form a new terminal interface implementing the modern reality of the terminal.

References

[SM] Richard Stones & Neil Matthew, Beginning Linux Programming, Second Edition, 1996 & 1999, Wrox Press

[ESR] Eric Steven Raymond, The Art of Unix Programming, 2004, Addison-Wesley

[CDM] Rémy Card & Éric Dumas & Franck Mével, translation by Chris Skrimshire, The Linux Kernel Book, 1997 & 1998, John Wiley & Sons

[WRS] W. Richard Stevens, Advanced Programming in the UNIX Environment, 1993, Addison-Wesley

[WWW] Kurt Wall & Mark Watson & Mark Whitis et al., Linux Programming Unleashed, 1999, Sams