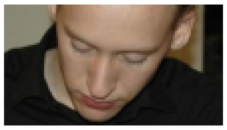


linusakesson.net

(show navigation)

The TTY demystified

Real teletypes in the 1940s.

The TTY subsystem is central to the design of Linux, and UNIX in general. Unfortunately, its importance is often overlooked, and it is difficult to find good introductory articles about it. I believe that a basic understanding of TTYs in Linux is essential for the developer and the advanced user.

Beware, though: What you are about to see is not particularly elegant. In fact, the TTY subsystem — while quite functional from a user's point of view — is a twisty little mess of special cases. To understand how this came to be, we have to go back in time.

History

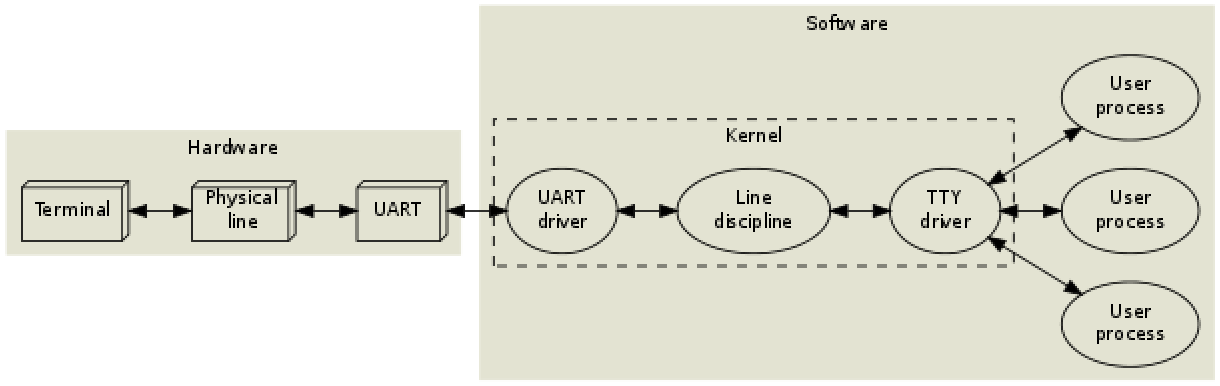
In 1869, the *stock ticker* was invented. It was an electro-mechanical machine consisting of a typewriter, a long pair of wires and a ticker tape printer, and its purpose was to distribute stock prices over long distances in realtime. This concept gradually evolved into the faster, ASCII-based *teletype*. Teletypes were once connected across the world in a large network, called *Telex*, which was used for transferring commercial telegrams, but the teletypes weren't connected to any computers yet.

Meanwhile, however, the computers — still quite large and primitive, but able to multitask — were becoming powerful enough to be able to interact with users in realtime. When the command line eventually replaced the old batch processing model, teletypes were used as input and output devices, because they were readily available on the market.

There was a plethora of teletype models around, all slightly different, so some kind of software compatibility layer was called for. In the UNIX world, the approach was to let the operating system kernel handle all the low-level details, such as word length, baud rate, flow control, parity, control codes for rudimentary line editing and so on. Fancy cursor movements, colour output and other advanced features made possible in the late 1970s by solid state *video terminals* such as the VT-100, were left to the applications.

In present time, we find ourselves in a world where physical teletypes and video terminals are practically extinct. Unless you visit a museum or a hardware enthusiast, all the TTYs you're likely to see will be emulated video terminals — software simulations of the real thing. But as we shall see, the legacy from the old cast-iron beasts is still lurking beneath the surface.

The use cases



A user types at a terminal (a physical teletype). This terminal is connected through a pair of wires to a *UART* (Universal Asynchronous Receiver and Transmitter) on the computer. The operating system contains a *UART driver* which manages the physical transmission of bytes, including parity checks and flow control. In a naïve system, the UART driver would then deliver the incoming bytes directly to some application process. But such an approach would lack the following essential features:

Line editing. Most users make mistakes while typing, so a backspace key is often useful. This could of course be implemented by the applications themselves, but in accordance with the UNIX design philosophy, applications should be kept as simple as possible. So as a convenience, the operating system provides an editing buffer and some rudimentary editing commands (backspace, erase word, clear line, reprint), which are enabled by default inside the *line discipline*. Advanced applications may disable these features by putting the line discipline in *raw* mode instead of the default *cooked* (or *canonical*) mode. Most interactive applications (editors, mail user agents, shells, all programs relying on `curses` or `readline`) run in raw mode, and handle all the line editing commands themselves. The line discipline also contains options for character echoing and automatic conversion between carriage returns and linefeeds. Think of it as a primitive kernel-level `sed(1)`, if you like.

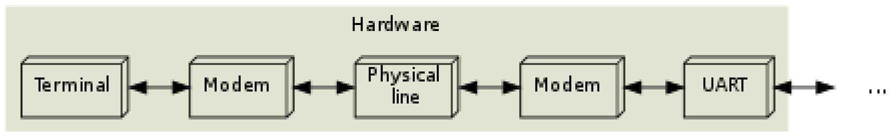
Incidentally, the kernel provides several different line disciplines. Only one of them is attached to a given serial device at a time. The default discipline, which provides line editing, is called `N_TTY` (`drivers/char/n_tty.c`, if you're feeling adventurous). Other disciplines are used for other purposes, such as managing packet switched data (ppp, IrDA, serial mice), but that is outside the scope of this article.

Session management. The user probably wants to run several programs simultaneously, and interact with them one at a time. If a program goes into an endless loop, the user may want to kill it or suspend it. Programs that are started in the background should be able to execute until they try to write to the terminal, at which point they should be suspended. Likewise, user input should be directed to the foreground program only. The operating system implements these features in the *TTY driver* (`drivers/char/tty_io.c`).

An operating system process is "alive" (has an *execution context*), which means that it can perform actions. The TTY driver is not alive; in object oriented terminology, the TTY driver is a passive object. It has some data fields and some methods, but the only way it can actually do something is when one of its methods gets called from the context of a process or a kernel interrupt handler. The line discipline is likewise a passive entity.

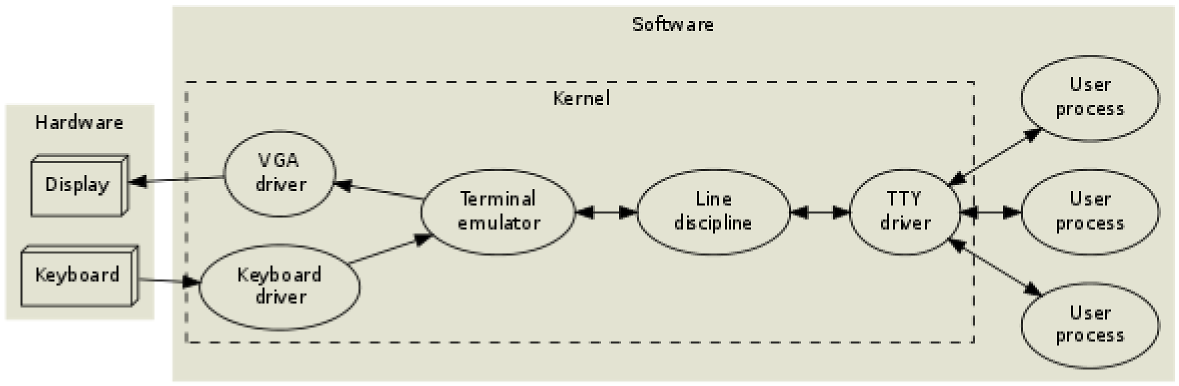
Together, a particular triplet of UART driver, line discipline instance and TTY driver may be referred to as a *TTY device*, or sometimes and TTY driver may be referred to as a *TTY device*, or sometimes just TTY. A user process can affect the behaviour of any TTY device by manipulating the corresponding device file under `/dev`. Write permissions to the device file are required, so when a user logs in on a particular TTY, that user must become the owner of the device file. This is traditionally done by the `login(1)` program, which runs with root privileges.

The physical line in the previous diagram could of course be a long-distance phone line:



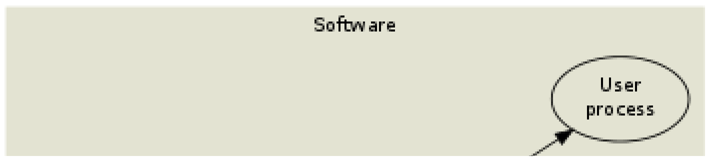
This does not change much, except that the system now has to handle a modem hangup situation as well.

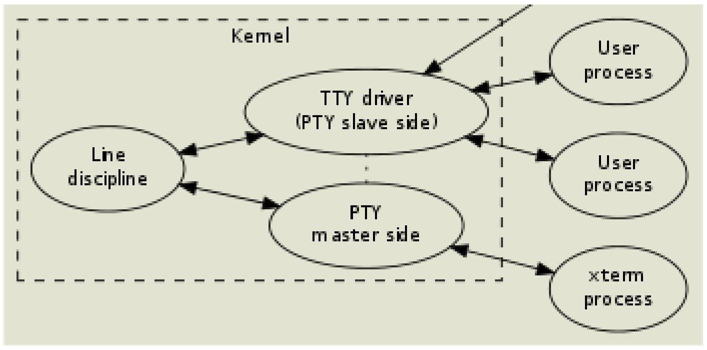
Let's move on to a typical desktop system. This is how the Linux console works:



The TTY driver and line discipline behave just like in the previous examples, but there is no UART or physical terminal involved anymore. Instead, a video terminal (a complex state machine including a *frame buffer* of characters and graphical character attributes) is emulated in software, and rendered to a VGA display.

The console subsystem is somewhat rigid. Things get more flexible (and abstract) if we move the terminal emulation into userland. This is how `xterm(1)` and its clones work:



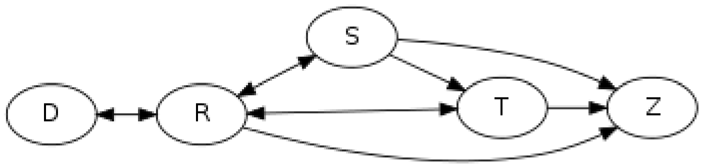


To facilitate moving the terminal emulation into userland, while still keeping the TTY subsystem (session management and line discipline) intact, the *pseudo terminal* or *pty* was invented. And as you may have guessed, things get even more complicated when you start running pseudo terminals inside pseudo terminals, à la `screen(1)` or `ssh(1)`.

Now let's take a step back and see how all of this fits into the process model.

Processes

A Linux process can be in one of the following states:



R	Running or runnable (on run queue)
D	Uninterruptible sleep (waiting for some event)
S	Interruptible sleep (waiting for some event or signal)
T	Stopped, either by a job control signal or because it is being traced by a debugger.
Z	Zombie process, terminated but not yet reaped by its parent.

By running `ps l`, you can see which processes are running, and which are sleeping. If a process is sleeping, the `WCHAN` column ("wait channel", the name of the wait queue) will tell you what kernel event the process is waiting for.

```
$ ps l
F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
0   500  5942  5928  15   0  12916  1460 wait   Ss   pts/14     0:00 -/bin/bash
0   500 12235  5942  15   0  21004  3572 wait   S+   pts/14     0:01 vim index.php
0   500 12580 12235  15   0   8080  1440 wait   S+   pts/14     0:00 /bin/bash -c (ps l) >/tmp/v727757/1 2>&1
0   500 12581 12580  15   0   4412   824 -      R+   pts/14     0:00 ps l
```

The "`wait`" wait queue corresponds to the `wait(2)` syscall, so these processes will be moved to the running state whenever there's a state change in one of their child processes. There are two sleeping states: Interruptible sleep and uninterruptible sleep. Interruptible sleep (the most common case) means that while the process is part of a wait queue, it may actually also be moved to the running state when a signal is sent to it. If you look inside the kernel source code, you will find that any kernel code which is waiting for an event must check if a signal is pending after schedule() returns, and abort the syscall in that case.

In the `ps` listing above, the `STAT` column displays the current state of each process. The same column may also contain one or more attributes, or flags:

s	This process is a session leader.
+	This process is part of a foreground process group.

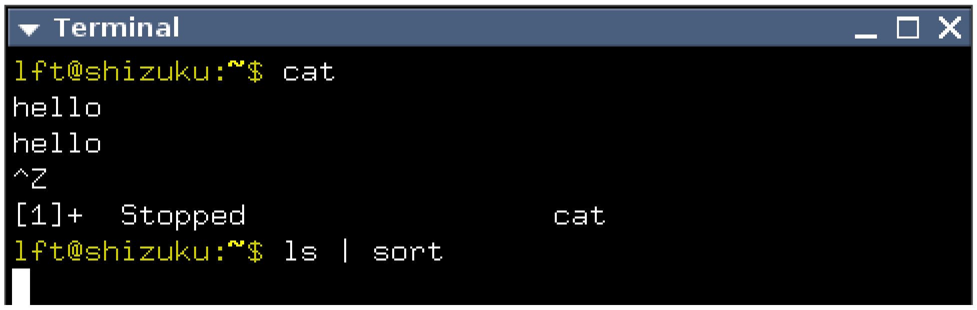
These attributes are used for job control.

Jobs and sessions

Job control is what happens when you press `^Z` to suspend a program, or when you start a program in the background using `&`. A job is the same as a process group. Internal shell commands like `jobs`, `fg` and `bg` can be used to manipulate the existing jobs within a *session*. Each session is managed by a *session leader*, the shell, which is cooperating tightly with the kernel using a complex protocol of signals and system calls.

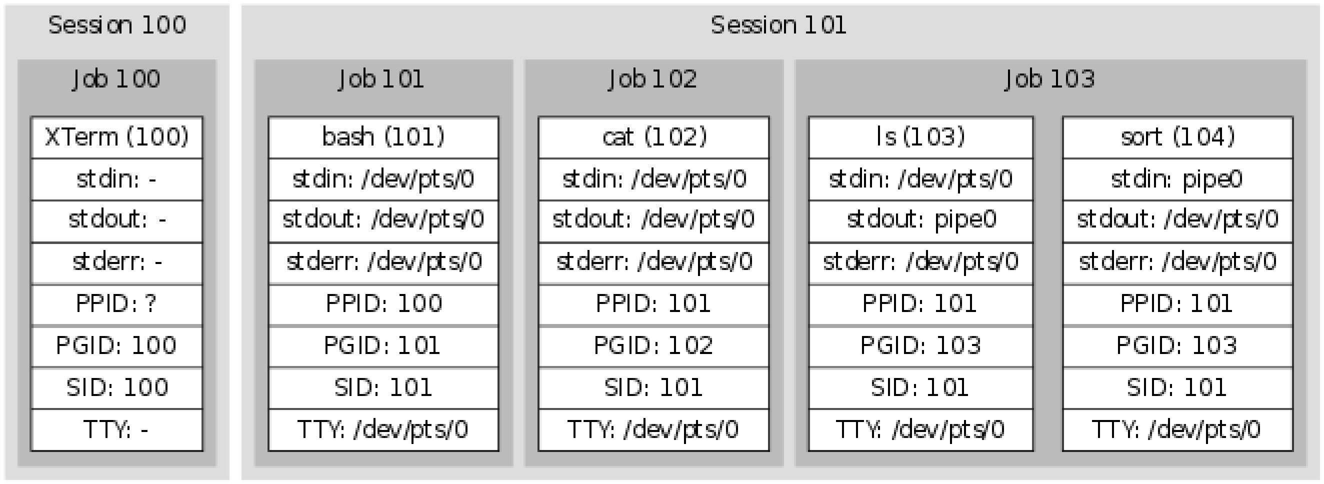
The following example illustrates the relationship between processes, jobs and sessions:

The following shell interactions...



```
lft@shizuku:~$ cat
hello
hello
^Z
[1]+  Stopped                 cat
lft@shizuku:~$ ls | sort
```

...correspond to these processes...



Session 100	Session 101			
Job 100	Job 101	Job 102	Job 103	
XTerm (100)	bash (101)	cat (102)	ls (103)	sort (104)
stdin: -	stdin: /dev/pts/0	stdin: /dev/pts/0	stdin: /dev/pts/0	stdin: pipe0
stdout: -	stdout: /dev/pts/0	stdout: /dev/pts/0	stdout: pipe0	stdout: /dev/pts/0
stderr: -	stderr: /dev/pts/0	stderr: /dev/pts/0	stderr: /dev/pts/0	stderr: /dev/pts/0
PPID: ?	PPID: 100	PPID: 101	PPID: 101	PPID: 101
PGID: 100	PGID: 101	PGID: 102	PGID: 103	PGID: 103
SID: 100	SID: 101	SID: 101	SID: 101	SID: 101
TTY: -	TTY: /dev/pts/0	TTY: /dev/pts/0	TTY: /dev/pts/0	TTY: /dev/pts/0

...and these kernel structures.

- TTY Driver (`/dev/pts/0`).

```
Size: 45x13
Controlling process group: (101)
Foreground process group: (103)
UART configuration (ignored, since this is an xterm):
  Baud rate, parity, word length and much more.
Line discipline configuration:
  cooked/raw mode, linefeed correction,
  meaning of interrupt characters etc.
Line discipline state:
  edit buffer (currently empty),
  cursor position within buffer etc.
```

- pipe0

```
Readable end (connected to PID 104 as file descriptor 0)
Writable end (connected to PID 103 as file descriptor 1)
Buffer
```

The basic idea is that every pipeline is a job, because every process in a pipeline should be manipulated (stopped, resumed, killed) simultaneously. That's why `kill(2)` allows you to send signals to entire process groups. By default, `fork(2)` places a newly created child process in the same process group as its parent, so that e.g. a

`^C` from the keyboard will affect both parent and child. But the shell, as part of its session leader duties, creates a new process group every time it launches a pipeline.

The TTY driver keeps track of the foreground process group id, but only in a passive way. The session leader has to update this information explicitly when necessary. Similarly, the TTY driver keeps track of the size of the connected terminal, but this information has to be updated explicitly, by the terminal emulator or even by the user.

As you can see in the diagram above, several processes have `/dev/pts/0` attached to their standard input. But only the foreground job (the `ls | sort` pipeline) will receive input from the TTY. Likewise, only the foreground job will be allowed to write to the TTY device (in the default configuration). If the cat process were to attempt to write to the TTY, the kernel would suspend it using a signal.

Signal madness

Now let's take a closer look at how the TTY drivers, the line disciplines and the UART drivers in the kernel communicate with the userland processes.

UNIX files, including the TTY device file, can of course be read from and written to, and further manipulated by means of the magic `ioctl(2)` call (the Swiss army-knife of UNIX) for which lots of TTY related operations have been defined. Still, `ioctl` requests have to be initiated from processes, so they can't be used when the kernel needs to communicate *asynchronously* with an application.

In *The Hitchhiker's Guide to the Galaxy*, Douglas Adams mentions an extremely dull planet, inhabited by a bunch of depressed humans and a certain breed of animals with sharp teeth which communicate with the humans by biting them very hard in the thighs. This is strikingly similar to UNIX, in which the kernel communicates with processes by sending paralyzing or deadly signals to them. Processes may intercept some of the signals, and try to adapt to the situation, but most of them don't.

So a signal is a crude mechanism that allows the kernel to communicate asynchronously with a process. Signals in UNIX aren't clean or general; rather, each signal is unique, and must be studied clean or general; rather, each signal is unique, and must be studied individually.

You can use the command `kill -l` to see which signals your system implements. This is what it may look like:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

As you can see, signals are numbered starting with 1. However, when they are used in bitmasks (e.g. in the output of `ps s`), the least significant bit corresponds to signal 1.

This article will focus on the following signals: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGPIPE`, `SIGCHLD`, `SIGSTOP`, `SIGCONT`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU` and `SIGWINCH`.

SIGHUP

- Default action: **Terminate**
- Possible actions: Terminate, Ignore, Function call

`SIGHUP` is sent by the UART driver to the entire session when a hangup condition has been detected. Normally, this will kill all the processes. Some programs, such as `nohup(1)` and `screen(1)`, detach from their session (and TTY), so that their child processes won't notice a hangup.

SIGINT

- Default action: **Terminate**
- Possible actions: Terminate, Ignore, Function call

`SIGINT` is sent by the TTY driver to the current foreground job when the *interactive attention* character (typically `^C`, which has ASCII code 3) appears in the input stream, unless this behaviour has been turned off. Anybody with access permissions to the TTY device can change the interactive attention character and toggle this feature; additionally, the session manager keeps track of the TTY configuration of each job, and updates the TTY whenever there is a job switch.

SIGQUIT

- Default action: **Core dump**
- Possible actions: Core dump, Ignore, Function call

`SIGQUIT` works just like `SIGINT`, but the quit character is typically `^\` and the default action is different.

SIGPIPE

- Default action: **Terminate**
- Possible actions: Terminate, Ignore, Function call

The kernel sends `SIGPIPE` to any process which tries to write to a pipe with no readers. This is useful, because otherwise jobs like `yes | head` would never terminate.

SIGCHLD

- Default action: **Ignore**
- Possible actions: Ignore, Function call

When a process dies or changes state (stop/continue), the kernel sends a `SIGCHLD` to its parent process. The `SIGCHLD` signal carries additional information, namely the process id, the user id, the exit status (or termination signal) of the terminated process and some execution time statistics. The session leader (shell) keeps track of its jobs using this signal.

SIGSTOP

- Default action: **Suspend**
- Possible actions: Suspend

This signal will unconditionally suspend the recipient, i.e. its signal action can't be reconfigured. Please note, however, that `SIGSTOP` isn't sent by the kernel during job control. Instead, `^Z` typically triggers a `SIGTSTP`, which can be intercepted by the application. The application may then e.g. move the cursor to the bottom of the screen or otherwise put the terminal in a known state, and subsequently put itself to sleep using `SIGSTOP`.

SIGCONT

- Default action: **Wake up**
- Possible actions: Wake up, Wake up + Function call

SIGTSTP

- Default action: **Suspend**
- Possible actions: Suspend, Ignore, Function call

`SIGCONT` will un-suspend a stopped process. It is sent explicitly by the shell when the user invokes the `fg` command. Since `SIGSTOP` can't be intercepted by an application, an unexpected `SIGCONT` signal might indicate that the process was suspended some time ago, and then un-suspended.

`SIGTSTP` works just like `SIGINT` and `SIGQUIT`, but the magic character is typically `^Z` and the default action is to suspend the process.

SIGTTIN

- Default action: **Suspend**
- Possible actions: Suspend, Ignore, Function call

If a process within a background job tries to read from a TTY device, the TTY sends a `SIGTTIN` signal to the entire job. This will normally suspend the job.

SIGTTOU

- Default action: **Suspend**
- Possible actions: Suspend, Ignore, Function call

If a process within a background job tries to write to a TTY device, the TTY sends a `SIGTTOU` signal to the entire job. This will normally suspend the job. It is possible to turn off this feature on a per-TTY basis.

SIGWINCH

- Default action: **Ignore**
- Possible actions: Ignore, Function call

As mentioned, the TTY device keeps track of the terminal size, but this information needs to be updated manually. Whenever that happens, the TTY device sends `SIGWINCH` to the foreground job. Well-behaving interactive applications, such as editors, react upon this, fetch the new terminal size from the TTY device and redraw themselves accordingly.

An example

Suppose that you are editing a file in your (terminal based) editor of choice. The cursor is somewhere in the middle of the screen, and the editor is busy executing some processor intensive task, such as a search and replace operation on a large file. Now you press `^Z`. Since the line discipline has been configured to intercept this character (`^Z` is a single byte, with ASCII code 26), you don't have to wait for the editor to complete its task and start reading from the TTY device. Instead, the line discipline subsystem instantly sends `SIGTSTP` to the foreground process group. This process group contains the editor, as well as any child processes created by it.

The editor has installed a signal handler for `SIGTSTP`, so the kernel diverts the process into executing the signal handler code. This code moves the cursor to the last line on the screen, by writing the corresponding control sequences to the TTY device. Since the editor is still in the foreground, the control sequences are transmitted as requested. But then the editor sends a `SIGSTOP` to its own process group.

The editor has now been stopped. This fact is reported to the session leader using a `SIGCHLD` signal, which includes the id of the suspended process. When all processes in the foreground job have been suspended, the session leader reads the current configuration from the TTY device, and stores it for later retrieval. The session leader goes on to install itself as the current foreground process group for the TTY using an `ioctl` call. Then, it prints something like " [1]+ Stopped" to inform the user that a job was just suspended.

At this point, `ps(1)` will tell you that the editor process is in the stopped state ("`T`"). If we try to wake it up, either by using the `bg` built-in shell command, or by using `kill(1)` to send `SIGCONT` to the process(es), the editor will start executing its `SIGCONT` signal handler. This signal handler will probably attempt to redraw the editor GUI by writing to the TTY device. But since the editor is now a background job, the TTY device will not allow it. Instead, the TTY will send job, the TTY device will not allow it. Instead, the TTY will send `SIGTTOU` to the editor, stopping it again. This fact will be communicated to the session leader using `SIGCHLD`, and the shell will once again write "[1]+ Stopped" to the terminal.

When we type `fg`, however, the shell first restores the line discipline configuration that was saved earlier. It informs the TTY driver that the editor job should be treated as the foreground job from now on. And finally, it sends a `SIGCONT` signal to the process group. The editor process attempts to redraw its GUI, and this time it will not be interrupted by `SIGTTOU` since it is now a part of the foreground job.

Flow control and blocking I/O

Run `yes` in an `xterm`, and you will see a lot of "`y`" lines swooshing past your eyes. Naturally, the `yes` process is able to generate "`y`" lines much faster than the `xterm` application is able to parse them, update its frame buffer, communicate with the X server in order to scroll the window and so on. How is it possible for these programs to cooperate?

The answer lies in *blocking I/O*. The pseudo terminal can only keep a certain amount of data inside its kernel buffer, and when that buffer is full and `yes` tries to call `write(2)`, then `write(2)` will *block*, moving the `yes` process into the interruptible sleep state where it remains until the `xterm` process has had a chance to read off some of the buffered bytes.

The same thing happens if the TTY is connected to a serial port. `yes` would be able to transmit data at a much higher rate than, say, 9600 baud, but if the serial port is limited to that speed, the kernel buffer soon fills up and any subsequent `write(2)` calls block the process (or fail with the error code `EAGAIN` if the process has requested non-blocking I/O).

What if I told you, that it is possible to explicitly put the TTY in a blocking state even though there is space left in the kernel buffer? That until further notice, every process trying to `write(2)` to the TTY automatically blocks. What would be the use of such a feature?

Suppose we're talking to some old VT-100 hardware at 9600 baud. We just sent a complex control sequence asking the terminal to scroll the display. At this point, the terminal gets so bogged down with the scrolling operation, that it's unable to receive new data at the full rate of 9600 baud. Well, physically, the terminal UART still runs at 9600 baud, but there won't be enough buffer space in the terminal to keep a backlog of received characters. This is when it would be a good time to put the TTY in a blocking state. But how do we do that from the terminal?

We have already seen that a TTY device may be configured to give certain data bytes a special treatment. In the default configuration, for instance, a received `^C` byte won't be handed off to the application through `read(2)`, but will instead cause a `SIGINT` to be delivered to the foreground job. In a similar way, it is possible to configure the TTY to react on a *stop flow* byte and a *start flow* byte. These are typically `^S` (ASCII code 19) and `^Q` (ASCII code 17) respectively. Old hardware terminals transmit these bytes automatically, and expect the operating system to regulate its flow of data accordingly. This is called flow control, and it's the reason why your `xterm` sometimes appears to lock up when you accidentally press `^S`.

There's an important difference here: Writing to a TTY which is stopped due to flow control, or due to lack of kernel buffer space, will *block* your process, whereas writing to a TTY from a background job will cause a `SIGTTOU` to suspend the entire process group. I don't know why the designers of UNIX had to go all the way to invent `SIGTTOU` and `SIGTTIN` instead of relying on blocking I/O, but my best guess is that the TTY driver, being in charge of job control, was designed to monitor and manipulate whole jobs; never the individual processes within them.

Configuring the TTY device

To find out what the controlling TTY for your shell is called, you could refer to the `ps l` listing as described earlier, or you could simply run the `tty(1)` command.

A process may read or modify the configuration of an open TTY device using `ioctl(2)`. The API is described in `tty_ioctl(4)`. Since it's part of the binary interface between Linux applications and the kernel, it will remain stable across Linux versions. However, the interface is non-portable, and applications should rather use the POSIX wrappers described in the `termios(3)` man page.

I won't go into the details of the `termios(3)` interface, but if you're writing a C program and would like to intercept `^C` before it becomes a `SIGINT`, disable line editing or character echoing, change the baud rate of a serial port, turn off flow control etc. then you will find what you need in the aforementioned man page.

There is also a command line tool, called `stty(1)`, to manipulate TTY devices. It uses the `termios(3)` API.

Let's try it!

```
$ stty -a
speed 38400 baud; rows 73; columns 238; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>; swtch = <undef>; start = ^Q;
stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff -iuclc -ixany imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt echoctl echoke
```

The `-a` flag tells stty to display **all** settings. By default, it will look at the TTY device attached to your shell, but you can specify another device with `-F`.

Some of these settings refer to UART parameters, some affect the line discipline and some are for job control. *All mixed up in a bucket for monsieur.* Let's have a look at the first line:

speed	UART	The baud rate. Ignored for pseudo terminals

		terminals.
rows, columns	TTY driver	Somebody's idea of the size, in characters, of the terminal attached to this TTY device. Basically, it's just a pair of variables within kernel space, that you may freely set and get. Setting them will cause the TTY driver to dispatch a `SIGWINCH` to the foreground job.
line	Line discipline	The line discipline attached to the TTY device. 0 is `N_TTY`. All valid numbers are listed in `/proc/tty/ldiscs`. Unlisted numbers appear to be aliases for `N_TTY`, but don't rely on it.

Try the following: Start an `xterm`. Make a note of its TTY device (as reported by `tty`) and its size (as reported by `stty -a`). Start `vim` (or some other full-screen terminal application) in the `xterm`. The editor queries the TTY device for the current terminal size in order to fill the entire window. Now, from a different shell window, type:

```
stty -F X rows Y
```

where *X* is the TTY device, and *Y* is half the terminal height. This will update the TTY data structure in kernel memory, and send a `SIGWINCH` to the editor, which will promptly redraw itself using only the upper half of the available window area.

The second line of `stty -a` output lists all the special characters. Start a new `xterm` and try this:

```
stty intr o
```

Now "`o`", rather than `^C`, will send a `SIGINT` to the foreground job. Try starting something, such as `cat`, and verify that you can't kill it using `^C`. Then, try typing "`hello`" into it.

Occasionally, you may come across a UNIX system where the backspace key doesn't work. This happens when the terminal emulator transmits a backspace code (either ASCII 8 or ASCII 127) which doesn't match the `erase` setting in the TTY device. To remedy the problem, one usually types `stty erase ^H` (for ASCII 8) or `stty` erase ^? (for ASCII 127). But please remember that many terminal applications use readline, which puts the line discipline in raw mode. Those applications aren't affected.

Finally, stty -a lists a bunch of switches. As expected, they are listed in no particular order. Some of them are UART-related, some affect the line discipline behaviour, some are for flow control and some are for job control. A dash (-) indicates that the switch is off; otherwise it is on. All of the switches are explained in the stty(1) man page, so I'll just briefly mention a few:

icanon toggles the canonical (line-based) mode. Try this in a new xterm:

```
stty -icanon; cat
```

Note how all the line editing characters, such as backspace and ^U, have stopped working. Also note that cat is receiving (and consequently outputting) one character at a time, rather than one line at a time.

echo enables character echoing, and is on by default. Re-enable canonical mode (stty icanon), and then try:

```
stty -echo; cat
```

As you type, your terminal emulator transmits information to the kernel. Usually, the kernel echoes the same information back to the terminal emulator, allowing you to see what you type. Without character echoing, you can't see what you type, but we're in cooked mode so the line editing facilities are still working. Once you press enter, the line discipline will transmit the edit buffer to cat, which will reveal what your wrote.

tostop controls whether background jobs are allowed to write to the terminal. First try this:

```
stty tostop; (sleep 5; echo hello, world) &
```

The & causes the command to run as a background job. After five seconds, the job will attempt to write to the TTY. The TTY driver will suspend it using SIGTTOU, and your shell will probably report this fact, either immediately, or when it's about to issue a new prompt to you. Now kill the background job, and try the following instead:

```
stty -tostop; (sleep 5; echo hello, world) &
```

You will get your prompt back, but after five seconds, the background job transmits `hello, world` to the terminal, in the middle of whatever you were typing.

Finally, `stty sane` will restore your TTY device configuration to something reasonable.

Conclusion

I hope this article has provided you with enough information to get to terms with TTY drivers and line disciplines, and how they are related to terminals, line editing and job control. Further details can be found in the various man pages I've mentioned, as well as in the glibc manual (`info libc`, "Job Control").

Finally, while I don't have enough time to answer all the questions I get, I do welcome feedback on this and other pages on the site. Thanks for reading!

Posted Friday 25-Jul-2008 17:46

Discuss this page

Anonymous
Sun 24-Aug-2008
21:36

Very good and informative article for a complex topic
- the tty system really gets demystified here.

Only a small correction:
Your statement "icanon switches between raw and cooked mode"
is not completely true.