The TTY demystified

The TTY demystified

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.

Real teletypes in the 1940s.

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

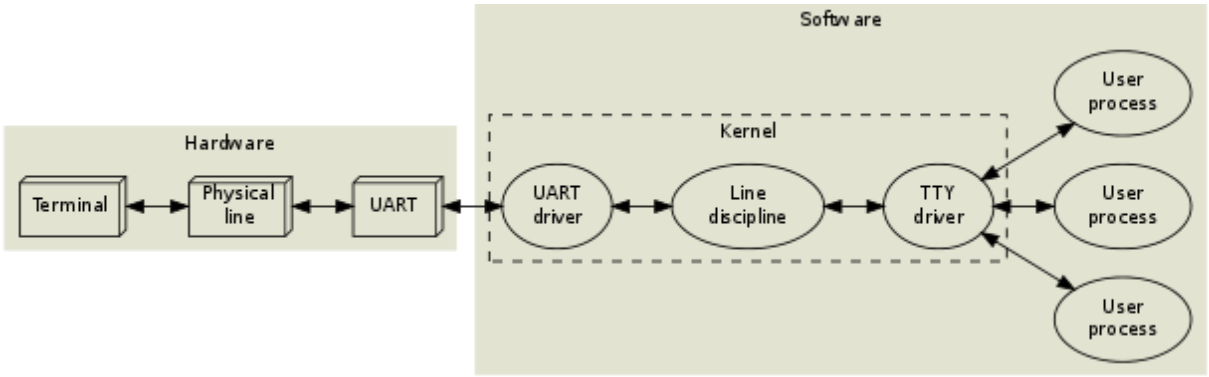
In 1869, the stock ticker was invented. It was an electro-mechanical machine consisting of a typewriter, a long pair of wires and a ticker tape printer, and its purpose was to distribute stock prices over long distances in realtime. This concept gradually evolved into the faster, ASCII-based teletype. 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.

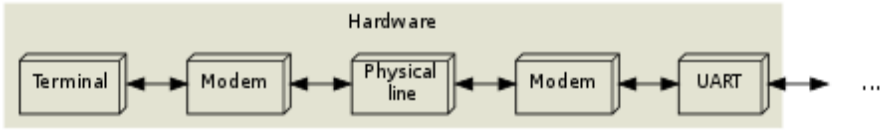
Session management. The user probably wants to run several programs simultaneously, and interact with them one at a time. If a program goes into an endless loop, the user may want to kill it or suspend it. Programs that are started in the background should be able to execute until they try to write to 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 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,

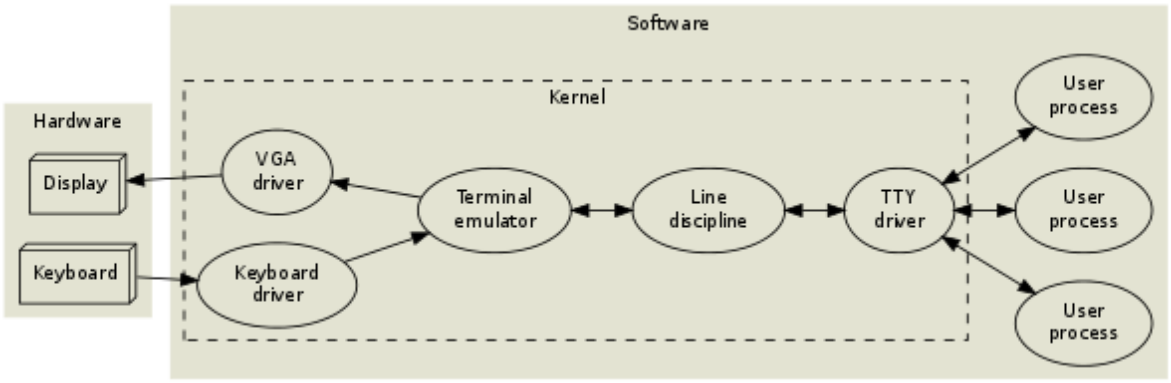
which runs with root privileges.

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



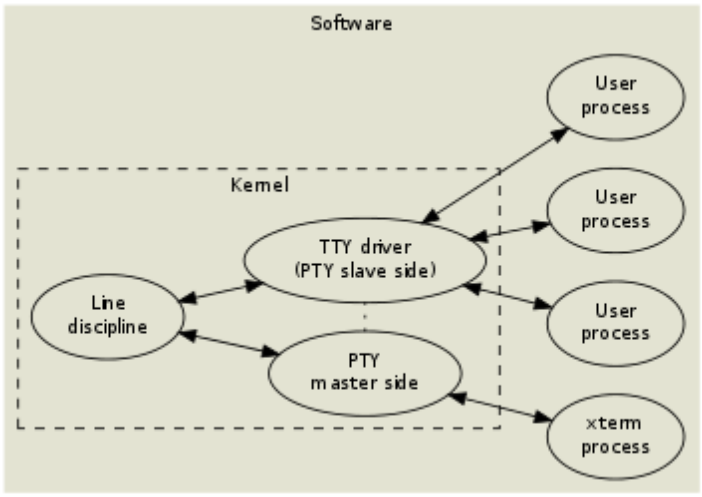
This does not change much, except that the system now has to handle a modem hangup situation as well.

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:

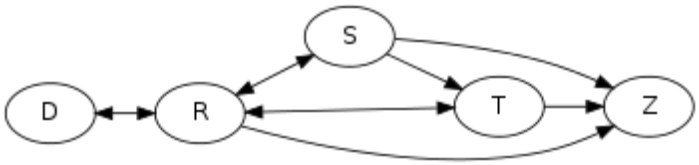


To facilitate moving the terminal emulation into userland, while still keeping the TTY subsystem (session management and line discipline) intact, the pseudo terminal or pty was invented. And as you may have guessed, things get even more complicated when you start running pseudo terminals inside pseudo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

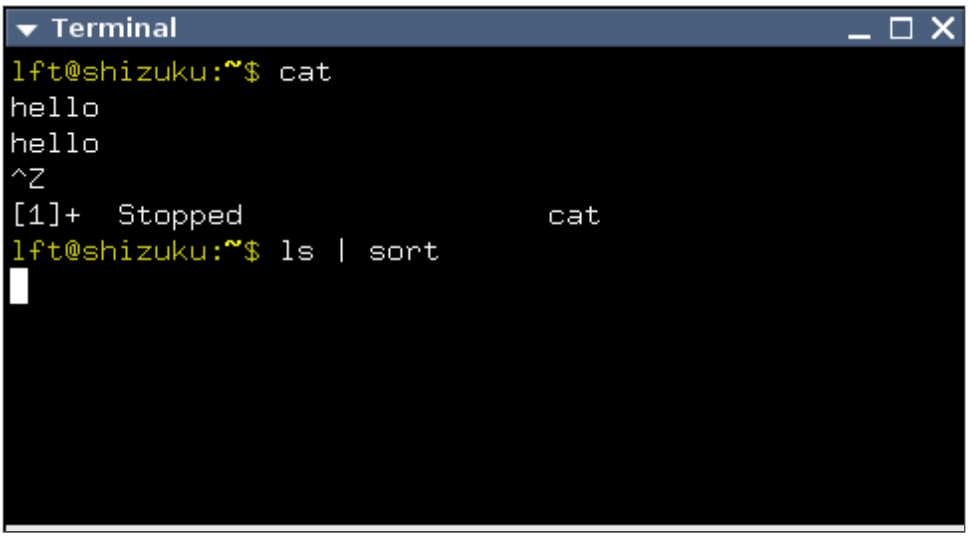
These attributes are used for job control.

Jobs and sessions

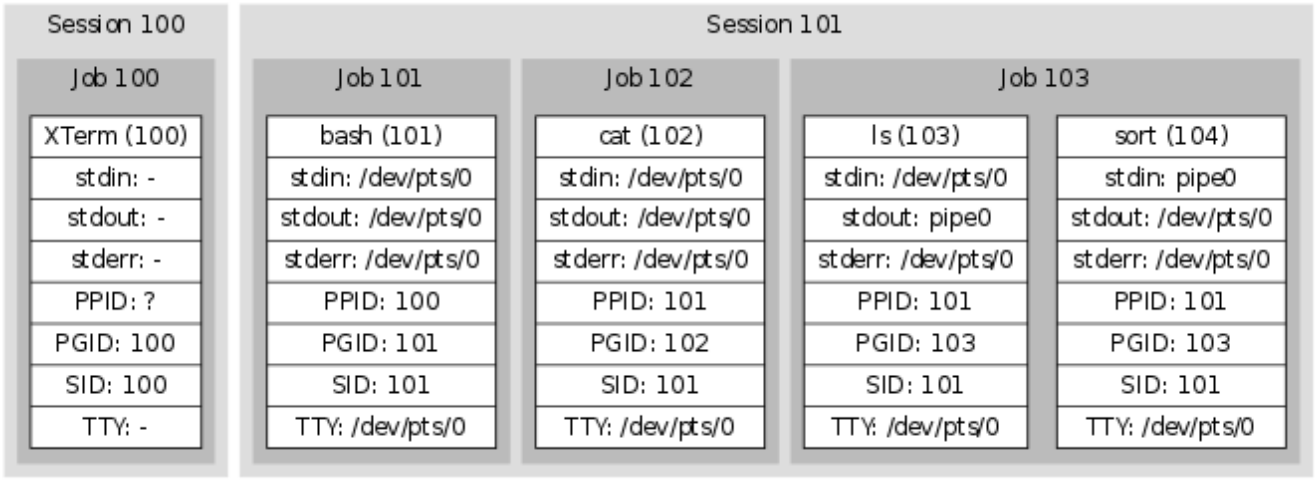
Job control is what happens when you press `^Z` to suspend a program, or when you start a program in the background using `&`. A job is the same as a process group. Internal shell commands like `jobs`, `fg` and `bg` can be used to manipulate the existing jobs within a session. Each session is managed 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...`



...correspond to these processes...



...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 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

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

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

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 `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-

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.
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

Now ^C, 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 19:46

Discuss this page

Anonymous
Sun 24-Aug-2008
23:36

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.

'stty -icanon' still interprets control characters such as Ctrl-C whereas 'stty raw' disables even this and is the real raw mode.

Greetings,
-Andreas.

lft
Linus Åkesson
Fri 29-Aug-2008
18:42

> 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.
>
> 'stty -icanon' still interprets control characters such as Ctrl-C whereas 'stty raw' disables even this and is the real raw mode.

Thanks!

Yes, you're quite right. I've fixed it.

Anonymous
Wed 26-Nov-2008
08:13

I have been trying to chew through documents in every unix book possible to explain how the TTY system really works. Your article has been more informative than all of these books. Without a programmers knowledge of how a unix kernel works, it's quite difficult for a System Administrator to understand concepts that are so crucial to their jobs. Thank you for your writing such a great article. Your help is truly appreciated.

Anonymous
Wed 10-Dec-2008
13:27

I really admire you!! I'm not use to post, but this article was by far the best I ever read about tty, i read many articles and even books but nothing so clear like this.
Keep doing things like this please...

Excelente!
Muchas Gracias
Fede Tula

Anonymous
Sat 20-Dec-2008
06:20

Good article. A few years ago, after some experimentation, I wrote up some TTY stuff for myself, because I couldn't find any good documentation. For example, I never understood why process group leaders (and consequently also session leaders) are prevented from calling setsid(2). (See the manual for what setsid(2) does.) I think I can explain it now.

The process P is a pg leader if P.PID = P.PGID. In the example of the article, "Job" means process group, and ls (103) is a process group leader:

ls.PID = 103
ls.PGID = 103
ls.SID = 101
ls.CTTY = /dev/pts/0

Suppose we allow ls to call setsid(2). This would have the following consequences:

ls.PID = 103 # unchanged
ls.PGID = 103 # set to ls.PID, but in fact this is no change!
ls.SID = 103 # set to ls.PID
ls.CTTY = <none>

Now ls is session leader (ls.SID = ls.PID), and ls is process group leader (ls.PGID = ls.PID).

At this point, however, sort (104) would belong to a process group (103) whose leader's (ls's) SID (103) doesn't match sort's SID (101)!

sort.PID = 104
sort.PGID = 103
sort.SID = 101

the pg leader for pg 103 is ls (103):
ls.PID = ls.PGID = 103 = sort.PGID
however
ls.SID = 103 != 101 = sort.SID

We have two processes in the same process group belonging to different sessions!

ls is prevented from calling setsid() because as current process leader its PGID doesn't change when it is set to its PID, while its SID changes. Thus it would leave the session while staying in the process group.

Sort, OTOH, can call setsid(), becuase it also leaves the process group:
sort.PID=104
sort.PGID=104 # leaves process group too
sort.SID=104
sort.CTTY=<none>

fork(2)-ing and calling setsid(2) in the child helps, because the child gets a new PID, which will be different from any PGID of the parent (as that PGID was the PID of some process), and so when the child calls setsid(2), the child.PGID := child.PID operation will actually change the child's inherited PGID and so the child will be able to leave the process group.

Right after fork():
parent.PGID = some_old_PID
child.PID = new_PID
child.PGID = parent.PGID = some_old_PID

The child calls setsid():
child.PGID = child.PID = new_PID != some_old_PID = parent.PGID

A session leader *could* call setsid(), despite also being a process group leader, since neither its PGID nor its SID would change. However, its CTTY would be set to <none>, and this would result in a situation where the original controlling process (= a session leader with a CTTY), for example, your shell, has no more access to the terminal!

Furthermore, there is the rule that when a controlling process dies, each session member (each process P with P.SID = SL.SID) loses access to the terminal (and possibly get a SIGHUP on the next read/write). This clearly shows the intent that no session member shall have access to the terminal when the session leader has none. This principle would be violated if the

when the session leader has none. This principle would be violated if the current session leader could detach from the CTTY by calling setsid(). (Or all session members would have to lose access to the CTTY, just as if the session leader died.)

Anonymous
Thu 25-Dec-2008
20:19

good job & best introductory for TTY

Anonymous
Mon 20-Apr-2009
10:28

amazing man , this is amazing !!! best tty article ever

Anonymous
Tue 2-Jun-2009
21:27

very good article!

Anonymous
Mon 15-Jun-2009
18:10

Holy Smokes dude, those were the days!

Riff
www.absolute-anonymity.us.tc

Anonymous
Mon 15-Jun-2009
21:59

I rarely comment on Blogs as I usually feel my input would not be necessary, but I must say this has been one of the best written and descriptive documents I have had the pleasure of learning from

Anonymous
Tue 16-Jun-2009
00:41

ZZZZZZZZZZZZZZZZZZZZZZZZZZ to long, jesus christ write up a summery for us lazy people

Anonymous
Tue 16-Jun-2009
02:22

I've been looking for an article like this for a very long time... thanks very much for putting this together!

Anonymous
Tue 16-Jun-2009
04:38

Great summary, and very useful. Thanks!

Anonymous
Tue 16-Jun-2009
08:57

Just freaking great!

script to make commands like less (more) adapt to changed screen size.
It even tries to exit from the command leaving the cursor on the "correct" place.
The kludge should work well with anything, that ought to be updated because of
a change in terminal window size.

CAVEATS
It's written for Unix under Mac OsX, doesn't really know if tput are implemented under Linux.

Well here we go. I'm sorry for the loss of tabs, it should have been indented.
I have used this for a year and a half and it really works.

```
#! /bin/bash
export LESS=" -I -r -f -J -S -g -M -x 4"
# -I ignore case when searching
# -r "raw" do not preparate ctrl-chars,
# -f force open special files (may be binary) BEWARE OF ANSISEQUENCES.
# -J show status column
# -S chop long lines.
# -g highlight on last hit in the search.
# -M Most Verbose status column...
# -x 4 tabspacing = 4
# ------------------------------------ the kludge starts here.....
ORIGLINES=$LINES

ESC=`printf "\e"`
ScreenRedraw_off=`echo -n "$ESC""[8m"`
ScreenRedraw_on=`echo -n "$ESC""[0m"`

function OkayScreen()
{
export PS1="" # Turns off the prompt to avoid cluttering..
echo -n ${ScreenRedraw_off}
CURLINES=`bash -i < ~/bin/kludge.bash `
# ^^^^^^^^^^^^ NB! the path where kludge.bash should be placed.
if [ $CURLINES -gt $ORIGLINES ] ; then
TO_SKIP="$(expr "$CURLINES" '-' "$ORIGLINES")"
if [ $TO_SKIP -lt 3 ] ; then
TO_SKIP="$(expr "$TO_SKIP" '-' '2')"
else
TO_SKIP="$(expr "$TO_SKIP" '-' '1')"
fi
```

```
''
tput cuu 1 #cursor up one line
echo -n ${ScreenRedraw_on}
echo -n "\$" #restores prompt
echo -n ${ScreenRedraw_off}
tput cud $TO_SKIP
echo -n ${ScreenRedraw_on}
echo # activate cli correct position.
else
tput cuu 2
echo ${ScreenRedraw_on}
fi
}
trap OkayScreen SIGWINCH
# if [ -t 0 ] ; then # /* this enables syntax highlighting */
# $VIMRUNTIME/macros/less.sh $@ /* After tinkering with vim */
# else
/usr/bin/less $@
# fi
trap '' SIGWINCH
# cp ./tmp/.vimrc~
```

Anonymous
Tue 16-Jun-2009
09:05

> Just freaking great!
>
> Forgot the innerpart, which makes it all work ....
> This is a second script called kludge.bash which I have in my ~/bin folder.
> Needs to execute this to get the changed winsize in a new process since at
> least bash 2.05a didn't update the LINE variable in active process in the terminal window.
>
> ```
> #! /bin/bash
> # ### SYS Finds the number of lines in a window after window rechange - less... !#
> # kludge.scr - to be placed in the ~/bin folder is the inner workings of the bash script named less
> PS1=""
> shopt -s checkwinsize
> echo $LINES
> ```

Anonymous
Wed 17-Jun-2009
01:58

Thanks for the great article!

Thanks for the great article!

I'd recommend adding some info about the *wide* spread myth of parent's death triggering SIGHUPs for all its children.

Recall that the related behavior only applies to session leaders, and is triggered in any of the two following cases:

#1:
IF session leader exiting
..IF it has ctty
....send SIGHUP to foreground PG
..ELSE
....send SIGHUP to foreground PG at last time it had ctty

#2:
IF session leader detaching (TIOCNOTTY)
..send SIGHUP to foreground PG

--JuanJo

Anonymous
Wed 17-Jun-2009
12:42

Actually, to be more precise:
#1:
IF session leader exiting
..IF it has ctty
....send SIGHUP to foreground PG
..ELSE
....send SIGHUP,SIGCONT to foreground PG at last time it had ctty

#2:
IF session leader detaching (TIOCNOTTY)
..send SIGHUP,SIGCONT to foreground PG

See: http://google.com/codesearch/p?hl=en&sa=N&cd=2&ct=rc#p4tPAkVsQ_c/linux-2.2.26/drivers/char/tty_io.c&l=537"

--JuanJo AKA jjo

lft
Linus Åkesson
Sun 21-Jun-2009
14:58

A reader pointed out that VT (in VT-100) stands for "video terminal", not "virtual terminal". This has been fixed.

Anonymous
Thu 30-Jul-2009
15:07

Thanks for the article, really helped !

Anonymous
Thu 30-Jul-2009
21:09

Thankyou, I found this to be an excellent TTY primer.

Anonymous
Sat 1-Aug-2009
00:14

Bookmarked.
Good tips =)

Anonymous
Mon 10-Aug-2009
09:21

Very nice. I'm currently (trying to) write a Unix-clone mostly from scratch, and this is a good resource on how TTYs are supposed to work.

Anonymous
Sun 16-Aug-2009
17:32

Hi--how are things in Sweden?

Every once in a while I get up the ambition to complain about the width of text on a web page, and you're the lucky winner today--sorry ;-)

This could be a good article--from the looks of it, it probably is--but why is it (and so many other web pages today) so wide?

Checking one line at random, it is 130 characters wide:

```
echo "Meanwhile, however, the computers — still quite large and
primitive, but able to multitask — were becoming powerful enough to" |
wc
1 20 130
```

Oh, and I'm ignoring the stuff in the left hand panel / column--I simply horizontally scroll so that panel is not visible.

I have three choices if I want to read your article:

* horizontally scroll on each line

* set the type size very small (or zoom out), so an entire line appears on the screen, then use a magnifying glass

* copy and paste the text to a file and read it in an editor--possibly deleting hard line breaks to let the text flow better

deleting hard line breaks to let the text flow better.

Ideally, and I've seen it done this way, so I believe it can be done:

* the text should be arranged to wrap to the width of the (reader's) window

* if there are long lines of code (pre-formatted text), or wide pictures, or something like that, the other text should still wrap to the width of the reader's window, although he'll have to horizontally scroll to see the full picture or code or whatever. (This is the part I'm specifically referring to as having seen done, but I can't remember any details (like an example, or how to do it--I'll try to pay attention and find some).

Anyway, sorry for the rant--thanks for making the effort to create and disseminate pages with information like this!

Randy Kramer

Anonymous
Fri 4-Sep-2009
11:24

Wonderful article.the explanation is meticulous and elegant. Thank you.

Anonymous
Sat 12-Sep-2009
16:09

GREAT ARTICLE!!! Read articles and books on Terminal I/O and none as clear as this summary.
Well Done!

Anonymous
Wed 23-Sep-2009
16:26

Thanks for a good article

Anonymous
Sat 17-Oct-2009
10:12

Thanks for this very good article

Anonymous
Tue 3-Nov-2009
18:53

Thanks for such a wonderful article. I am a beginner in tty and your article was of great help!!

Anonymous
Thu 5-Nov-2009
20:54

nice article..thanx for sharing ur knowledge :)

Anonymous
Tue 22-Dec-2009
21:58

Thanks a lot for this precise article ! This has been very useful to me...
(reading an external device on RS232 from bash...)

Anonymous
Tue 9-Feb-2010
08:34

Thank you for sharing such a great artical. -Hai

Anonymous
Thu 15-Apr-2010
11:08

Thanks a lot for this great article

Anonymous
Tue 4-May-2010
10:36

Wonderful article! I never learned much about unix process stuff (apart from little practical things like piping, detaching, killing, etc), but now I feel like I'm starting to see what's behind the magical terminals! Thank you :)

Anonymous
Wed 12-May-2010
03:00

"daemonizing" a process - detaching from the tty - would probably also fit here. -rurban

Anonymous
Thu 13-May-2010
03:17

What does TTY mean?

lft
Linus Åkesson
Thu 13-May-2010
17:33

> What does TTY mean?

TeleTYpe.

Anonymous
Wed 14-Jul-2010
05:21

What Randy Kramer said.

Anonymous excellent article, well done!

Anonymous excellent article, well done!
Sat 24-Jul-2010
16:28

Anonymous
Sun 25-Jul-2010
02:31

The process of daemonizing is covered elsewhere, easy to Google. It involved forking, killing the parent, then calling setsid() in the child, and optionally chdir()ing to / and closing stdin/stdout/stderr.

I am currently working on writing a toy OS, and this was very useful in its treatment of the basic structure of the TTY subsystem. Thanks.

ralph
Ralph Corderoy
Sun 25-Jul-2010
16:22

Nice article, various points...

The erase and kill characters used to be # and @, and as you were printing on paper there was no rubbing out, so you might see

$ ls @wc -l /etvc##c/passwd
42 /etc/passwd
$

where the `@' was killing the whole line entered so far and the `##' was erasing the preceding `vc'.

It's only modern shell that provide line editing, hence shell history substitutions like `!!' and `!$' existing. If /bin/sh is a plain old non-line-editing shell on your system then you can see the difference in tty settings by using `stty -a' from another terminal to capture the differences. Don't run stty(1) from, e.g., the bash shell since the shell will alter the tty settings before running stty. Here, bash has the literal next character, lnext, being undefined and turns off -icrnl, -icanon, and -echo.

The above example of # and @ was achieved by

$ sh
$ stty erase \# kill @ -crterase -echok
$ ls @wc -l /etvc##c/passwd
42 /etc/passwd
$ stty sane
$ exit
$

where /bin/sh is dash(1) on this Ubuntu system.

"W[illegible]

"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." I think it's read permission that's required to alter a tty's settings. It did used to be write, in the very early days, but since write(1) and mesg(1) meant users could write to one another's terminals it also meant they could alter their settings. Much fun could be had with changing erase to `e' for a second and back again at random intervals whilst the user was trying to type. So it was switched to require read permission which only the owner of tty normally has. This can be seen in stdin of stty needing to be re-directed to specify the terminal, and not stdout, e.g. `stty -a </dev/pts/1'.

Flow control, e.g. ^S and ^Q, existed long before the signals for job control. IIRC, it was Berkeley that added all the ^Z stuff and related signals, it wasn't Bell Labs.

The Linux kernel doesn't bother to implement all of the normal control characters. Flush is one that's missing, IIRC, which is set with stty's `eol2'. It's a shame.

Cheers,
Ralph.

P.S. There's a typo, `1970:s'.

lft
Linus Åkesson
Sun 25-Jul-2010
18:14

> ralph wrote:
> Nice article, various points...

Thanks! That was very interesting. It hadn't occured to me that erase/kill would be usable without interactive line editing, but it makes sense.

> ralph wrote:
> P.S. There's a typo, `1970:s'.

Changed to 1970s.

Anonymous
Mon 2-Aug-2010
22:22

How do you restart bash in tty0? For some reason my bash has died in all tty's (Ctrl-Alt-Fn). I does not show the login prompt anymore.

How can I restart it? I cannot reboot the machine. I can ssh into it from another machine.

Ift
Linus Åkesson
Thu 5-Aug-2010
10:10

> How do you restart bash in tty0? For some reason my bash has died in all tty's (Ctrl-Alt-Fn). I does not show the login prompt anymore.
>
> How can I restart it? I cannot reboot the machine. I can ssh into it from another machine.

I don't know what's wrong in your particular case, but it's init (pid 1) that's supposed to (re-)start the login program in each terminal. You can modify init's configuration at runtime by editing /etc/inittab and then doing "kill -HUP 1". But it's probably not an error in the configuration file, so use ps(1) to investigate what processes are running in the terminals.

Anonymous
Tue 10-Aug-2010
01:41

Great article.
For another great source on TTY devices you can go to:
http://publib.boulder.ibm.com/infocenter/aix/v6r1/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/ttysys.htm

Enjoy,
Ori

Anonymous
Fri 13-Aug-2010
12:59

Very good article...
We want the part II were the concepts gets even closer to the Linux implementation...

just an example:
an xterm session under ssh when resized makes the TTY device to adjust the terminal size and generate SIGWINCH signal for the running app to know about the change...
but if the xterm is under a serial line when resize occurs NOTHING happens (not kernel side size update, not SIGWINCH signal)... I know under serial line xterm does not communicate the size change, well where is the the place to patch in order to solve this, from the xterm side it would be very easy to send a escape sequence telling the TTY driver the new size but this driver should be patched for catching it and react as in the ssh case....

Thanks for your out of ordinary (little or inexistant content + lot of google

Thanks for your out of ordinary (little or inexistant content + lot of google adds) article. I hope we can get a deeper version someday.

Pat

Anonymous
Sat 21-Aug-2010
05:00

Hi,

A very interesting article. I have just published a related article on the terminals in French : http://www.etud.insa-toulouse.fr/~mcheramy/wordpress/?p=198
(And if you don't understand french, there are few interesting links in english at the end)

Thanks.
Max.

Anonymous
Fri 27-Aug-2010
14:22

How to Modify the tty driver so that control-W erases the previous word typed by the user.

Anonymous
Mon 25-Oct-2010
15:14

Nithin: thanks for the page....i got a lot of information from ur web page...

Anonymous
Wed 1-Dec-2010
15:45

Great article, very imformative please explain how to prevent UART overrun

Anonymous
Thu 9-Dec-2010
19:19

Hi there ! Would it be possible to create a raw tty which redirects input to some fifo and another tty to read from that fifo ? I know that is extrange what I'm asking...I'm building a z80 emulator and if this is possible would help me to test the input/output routines of the emulator without writting code to read/display...

Anonymous very helpful indeed Thanks a ton.
Sun 26-Dec-2010
22:32

Anonymous Good article.Thanks you very much
Thu 6-Jan-2011
11:46 /Renjith G

Anonymous
Wed 12-Jan-2011
14:12

this article is very good .
now i have a question.
linux use /dev/console in booting before the init called . so i write a progrm that run after kernel booted and pass my progrm with option init in boot parameter and bypass init program so the first program that run is my program.
this is my question : how i correct my program to resive signels from /dev/consoole?

Anonymous
Tue 25-Jan-2011
10:19

Awesome article! Its been very helpful in understanding TTY layer as whole.

Thanks
Ambresh

Anonymous
Wed 9-Feb-2011
18:56

Hi, this seems to be an excellent technical article, however is there any chance you can provide a broad description of TTY for a non technical audience. e.g. explain in simple language the origin, function and implications of TTY.
You may not consider this your remit, which is fair enough.

Anonymous Great Job, learned much
Wed 16-Feb-2011
08:57

Anonymous Very good document, thanks!
Wed 23-Feb-2011
21:35

21:33

Anonymous
Fri 25-Feb-2011
23:10

great article. I have known the term 'TTY' for a long time and never knew what the history behind it. Seeing how even current day UNIX system still make use of TTY is quite interesting to me. Now I feel more 'connected' when I press a key on my keyboard and seeing a character appear on the monitor screen. A good read.

Anonymous
Tue 1-Mar-2011
22:42

Wow! The name of the article says it all, but for real this time! Thanks!

Regards,
Newman

Anonymous
Fri 18-Mar-2011
21:18

Amazing article. Thanks so much for putting this together. I've been using Linux for 10 years, and this taught me quite a bit.

Anonymous
Mon 11-Apr-2011
20:08

> ZZZZZZZZZZZZZZZZZZZZZZZZZZ to long, jesus christ write up a summery for us lazy people

```
if(easilyBored || notInterested){
goToAnotherSiteThen();
} else {
try {
readAndLearn();
}
catch(DontUnderstandException dde){
wiki(dde.getSubject());
}
finally {
enjoyArticle();
}
}
```

Anonymous
Wed 29-Jun-2011
14:19

Fantastic expalanation. Keep up the good work

Anonymous
Fri 1-Jul-2011 04:12
Greatly informative article. Of course, Bookmarked.

Anonymous
Mon 11-Jul-2011 16:18
Very informative article; Thanks for knowledge sharing ...

Anonymous
Wed 7-Sep-2011 15:48
Fucking awesome. Thanks a lot for the info!

Anonymous
Wed 7-Sep-2011 15:49
Fucking awesome. Thanks a lot for the info!

Anonymous
Fri 9-Sep-2011 16:25

Excelent article, I started to read it at work, now I'm printing it to read it more carefuly on my way home. Thanks for sharing all this!

Anonymous
Mon 19-Sep-2011 22:32
Thank you for sharing your knoledge so generously!

Anonymous
Tue 27-Sep-2011 08:01
thanks for the works!

Anonymous
Fri 30-Sep-2011 16:39

Hi,

I am currently working on Solaris sparc 10, where i am seeing below problem with default setting.

Shell is allowing me to type in only 256 characters. e.g.

bash-3.00$ cat SunStudio12u1-SunOS-SPARC-tar-MLSunStudio12u1-SunOS-SPARC-tar-MLSunStudio12u1-SunOS-SPARC-tar-MLSunStudio12u1-SunOS-SPARC-tar-MLSunStudio12u1-SunOS-SPARC-tar-MLSunStudio12u1-SunOS-SPARC-tar-MLSunStudio12u1-SunOS-SPARC-tar-MLS

If i attempt to input more data nothing happens. After going through your notes it seems like i am crossing line buff limit. If yes can you please suggest how i can increase this limit

Minal Patil

Anonymous
Mon 24-Oct-2011
21:11

can anybody give exact command for sending messages via linux?

Anonymous
Wed 2-Nov-2011
03:21

> Hi,
> can you please suggest how i can increase this limit

http://tinyurl.com/6yql6r8

Anonymous
Mon 14-Nov-2011
21:33

From your disclaimer: “Please report any abuse, such as insults, slander, spam and illegal material”

Just a note: “slander” only applies to verbal communication. For written, use the term “libel” instead. :-D

Anonymous
Tue 22-Nov-2011
16:04

Thanks for writing the article. It was a bit heavy going at times, but it's helped me understand some of what is going on with the whole terminal thing.

Anonymous
Mon 12-Dec-2011
04:22

This was enormously informative and helpful. I've bookmarked it for future use. Thank you!

Anonymous
Wed 21-Dec-2011
04:20

I am trying to solve a mystery with certain services started at reboot or via a root crontab.

These sercvices seem to fail, unless I login to a shell that has a proper TTY setup
(eg: PuTTY of any type of ssh session)
If I start up these services with a proper tty, they work and continue to run.
My theory is that on reboot or via root crontab there is no tty.

How can I create a psuedo tty master/slave pair underwhich to run these services so they work, with me doing a manual ssh login.....?

Larry Wichter

Anonymous
Thu 12-Jan-2012
15:37

Very informative article, thanks very much for sharing it with the web.

Anonymous
Wed 18-Jan-2012
01:00

Real great article thanks very much.

Anonymous
Mon 20-Feb-2012
05:35

Superb post. Thank you.

Anonymous
Mon 5-Mar-2012
18:15

Great article! I'm currently running a real Teletype ASR-33 on a serial port with Ubuntu. I have to use stty to set the proper parameters for an uppercase-only terminal, change tabs to spaces, etc. Is there a way to determine what port I'm logging into (ttyS0? S1? S2, S3 or S4?) Right now, I have to run stty on all ports and get an access denied error for any port I'm not connected to. I'd like to find out what port I'm on and only run the stty on that port. Thanks for your time!

Anonymous
Mon 2-Apr-2012
14:21

> Is there a way to determine what port I'm logging into (ttyS0? S1? S2, S3 or S4?)

Sounds like a job for the tty(1) command.

Anonymous
Mon 2-Apr-2012
19:14

I think the commentary about line editing being part of the operating system to simplify programs is potentially misleading - the simplification of applications is a side effect.

If you're connected something like a VT100, the terminal handles line editing, and programs send control codes to the terminal to switch between cooked mode and raw mode. When virtual terminals were implemented, this functionality became part of the operating system to maintain compatibility with existing software.

Anonymous
Thu 12-Apr-2012
01:15

This is a really great job! Really helped with background jobs writing to stdout and going to sleep (putting 'stty -tostop;' before background command). It takes much time to generate meaningful examples - much appreciated!

Anonymous Thanks !!
Sat 14-Apr-2012
13:54

Anonymous
Sat 14-Apr-2012
14:21

Great Article and good writing. I came looking for one answer (which I found) and found that I could not stop reading (normally a sign of a good article).

Having been around as the last of the Telex's died out, having designed UART circuits, worked on Xenix and Unix Systems, and naturally Linux systems, your article plugged some reasonable holes I had on the subject...

Thanks again...

Anonymous
Tue 17-Apr-2012
18:35

This is a very good article and I've read and enjoyed it multiple times (no, really!).

One thing I think it does lack is that when it's talking about signals, it should mention about Linux's new(ish) signalfd.

Anonymous Great and insightful article!!!

Anonymous
Sat 5-May-2012
16:56

Great and insightful article.
From this article I gain general knowledge about architecture of Linux devices(LowLevelDriver<->LineDiscipline<->HighLevelDriver). It's really a core of Linux I/O.

Thank you very much.

_____
Sichkar Dmytro from Ukraine
dmbios@mail.ru

Anonymous
Wed 16-May-2012
01:52

Thank you so much. Great article!

Anonymous
Thu 24-May-2012
17:22

Good stuff, thanks!

Anonymous
Mon 28-May-2012
04:48

Thanks.

whizziwig
David Blackman
Mon 4-Jun-2012
19:16

Can you explain why sometimes my terminal gets diseased, and the line I'm typing overwrites itself, or the cursor is in the wrong place, or similar symptoms? is there a way to fix it?

Anonymous
Mon 4-Jun-2012
21:11

Love the article, but I did have a comment -- namely that I wouldn't describe the DEC VT100 as solid state because of its CRT. Again, though, the article is great! Thanks!

Anonymous
Tue 5-Jun-2012
09:20

Absolutely terrific!

Anonymous
Tue 5-Jun-2012
21:07

That brings back old memories. The TTY shown is from the 1940's were 5 bit rather than the later 8 bit (really 7 bit plus parity) machines seen in the

bit rather than the later 8 bit (really 7 bit plus parity) machines seen in the late 60's and 70's. Next retro core memory come back.

RC Roeder

Anonymous
Wed 6-Jun-2012
05:02

Fantastic! awesome article you should think about to write a book or something, It clarified me some things, even having 5+ years of Unix experience.

Al Mejida.

Anonymous Thanks a lot for writing this!
Tue 12-Jun-2012
20:59

Anonymous
Fri 6-Jul-2012 17:46

"yes" program , produced in 2009 !

Thank you David McKenzie for your contribution to open source community !

I am just wondering what in your background that enabled the FSF to accept such worthless contribution ?

Member of what masonic lodge or what church or son of a war hero or billonarties you have to be so they accept that piece of crap ?

For reference:

yes command - otputs a line on tty until killed !

coded and added to Linux in 2009.

Anonymous
Fri 6-Jul-2012 17:48

> "yes" program , produced in 2009 !
>
> Thank you David McKenzie for your contribution to open source community !
>
> I am just wondering what in your background that enabled the FSF to accept such worthless contribution ?

> Member of what masonic lodge or what church or son of a war hero or billonarties you have to be so they accept that piece of crap ?
>
> For reference:
>
> yes command - otputs a line on tty until killed !
>
> coded and added to Linux in 2009.

interesing point although might be harsh on a boy who modified a hello world program and managed to add it to Linux.

Anonymous
Fri 6-Jul-2012 17:50

> > "yes" program , produced in 2009 !
>
> I disagree an entire source code of the yes in TARball may be a good example of how to add a new command to Linux, sort of like a new command template.
>
> www.LinuxCAD.com

lft
Linus Åkesson
Fri 6-Jul-2012 18:07

> "yes" program , produced in 2009 !
> ...
> Member of what masonic lodge or what church or son of a war hero or billonarties you have to be so they accept that piece of crap ?

Your angry ignorance is amusing. Do you also consider echo or /dev/zero crap? Unix commands are simple by design, so that they can be combined easily. For instance, off the top of my head, here's a way to list the first 100 powers of two: (echo 1; yes 'p2*')|dc|head -n 100

Anonymous
Fri 6-Jul-2012 23:17

```
int main( in argc, char** arg)
{
if ( argc < 2 )
{
printf( "\nNo program, prints line to tty until killed !... please put now my
```

```
name in Linux.\n" );
exit(1);
}
while(1)
{
printf( argv[1] );
};
}
```

Anonymous Thank you!!
Fri 13-Jul-2012
19:37

Anonymous
Sat 14-Jul-2012
05:33

> Shell is allowing me to type in only 256 characters. e.g.
>
> bash-3.00$ cat
> SunStudio12u1-SunOS-SPARC-
>
> Minal Patil

Can Web Master please delete this post so that the page would not be made so wide in some browser to be unreadable?
Thanks.

lft
Linus Åkesson
Sat 14-Jul-2012
13:39

> Can Web Master please delete this post so that the page would not be made so wide in some browser to be unreadable?
> Thanks.

I've added some "max-width" attributes that should fix the problem. What browser do you use?

Anonymous
Sun 15-Jul-2012
00:18

> lft wrote:
>
> > Can Web Master please delete this post

> I've added some "max-width" attributes that should fix the problem. What browser do you use?

Thanks for being so responsive.

I did realize, later, that it was just elinks that laid out the page to be annoyingly wide.

$ elinks http://www.linusakesson.net/programming/tty/index.php

(Use "]" or mouse to click near the right border of the terminal to scroll right.)

I also found, later, that removing the posting containing a unusually long single-word helps, but a little scrolling left and right is still needed in elinks.

Since this is restricted to an uncommon browser, don't worry about it.

(I started using elinks on one computer due to the recent development on Linux desktop that resulted in poor support for older video cards where Firefox would freeze the entire desktop when rendering some common web content. elinks worked great for text content.)

By the way, thanks for the page. It was a great read.

lft
Linus Åkesson
Sun 15-Jul-2012
22:29

> lft wrote:
>
> > Can Web Master please delete this post
>
> I've added some "max-width" attributes that should fix the problem. What browser do you use?
>
> Thanks for being so responsive.
>
> I did realize, later, that it was just elinks that laid out the page to be annoyingly wide.

I use elinks quite a lot myself, actually, for a fast, nonsense-reduced browsing experience. The page is not wider than the screen on my setup (elinks version 0.12pre5 with CSS enabled).

Anonymous
Wed 19-Sep-2012
23:01

23:01

It's not to keep application simple that the line discipline is in the kernel. The unix philosophy is quite the contrary, to keep the KERNEL simple, and deport to user space the handing of complexities that they don't want inside the kernel. (See for example interrupted system calls and signal handling, but that's another topic).

The reason why the line discipline is inside the kernel, is to avoid context switches at the reception of each character (which in the early times of small core memories, would imply swap-outs and swap-ins!). So the line discipline keeps in a kernel buffer a line of input, and since it's simple enough to test for a specific byte and decrement a counter to implement the backspace "editing" (and a few other simple editing functions), it's done there.

The alternative, is to use the raw mode, where the characters are forwarded to the application as soon as they're received, which is needed for more sophisticated editors, like the (at the time) famously known Eight Megabytes And Constantly Swapping editor (emacs). And indeed, since emacs had to use this raw mode, which implies a context switch at the reception of each character typed, it was constantly swapping when the computers hadn't enough memory to keep emacs and all the other programs in core.

Anonymous
Thu 20-Sep-2012
04:54

Thanks a lot! Was looking for this kind of information all over the place, never found anything good. If you got more please keep it coming.

Anonymous
Fri 28-Sep-2012
00:03

The most concise/intuitive article I've read so far on the subject. Diagrams were especially helpful. If you manage to write a book on Linux is such manner, no doubt it's going to be a bestseller...

Thanks,
Val.

Anonymous
Fri 5-Oct-2012
19:38

> ZZZZZZZZZZZZZZZZZZZZZZZZZZ to long, jesus christ write up a summery for us lazy people

too many Zs. write a shorter insult with proper spelling for us non-moron

people

Anonymous
Sun 11-Nov-2012
21:09

Thanks for the page. You help me a lot.

Anonymous
Thu 29-Nov-2012
11:27

Great job!

I have just a suggestion.
This article can be used as a first step by people (like me...) who don't have a deep knowledge on the topic,
so why not to add a "References" section to help going into more depth?

Cheers!

Anonymous
Sun 30-Dec-2012
10:26

Thank you very much for this great article!

Anonymous
Fri 4-Jan-2013
16:38

Excellent!! Thank you very much for this article. Watching and trying to understand technological issues via their historical context and being showed how they evolved make it really a pleasure to follow your explanations and very easy to get them.

Anonymous
Wed 6-Feb-2013
02:27

Very useful article, thanks.

A couple of clarifications that would be great:

1. It seems that there is a 1:1 correspondence between a session and the associated tty (bearing in mind that the associated tty may be "none") - is that true? In other words, can one session include processes with different ttys, and can processes in two different sessions be associated with the same tty? (and if so... what does that mean?!)

2. What are the exact rules for automatic raising of SIGHUP? It seems that this is raised by the TTY driver, right? Does the POSIX specification specify when this should happen and who should receive it, and does

Linux follow that? I have a confusing situation involving ssh -t raising SIGHUP on exit, while logging out of an interactive ssh login apparently does not.

Thanks
D

Anonymous
Mon 11-Mar-2013
18:54

Wow! Thanks Linus for the article.
This is how I learned about things in the past, from people who knew their craft well. Compare this with how we do it today - don't think, google first, rummage through incoherent posts, forums, mailing lists and if you are patient enough might be able to put pieces together to get just the clues/pointers to the information you want! So much for the 'age of information'.
Keep up the good work!
Thanks.

- VJ

Anonymous
Tue 2-Apr-2013
13:01

Real very informative! Kudos to your effort and thanks for hard work!

ulzha
ulzha
Sun 7-Apr-2013
16:59

Fantastic, tack så mycket!

I'm not writing an OS, but writing an improved screen or GUI terminal I do consider.

In particular I was interested in whether there was an API to tell foreground job's output and background job's output apart (e.g. associating a PID with each chunk output) to highlight them understandably or something. Now I figure that I just might be able to implement that by wisely trapping SIGTTOUs perhaps...

Anonymous
Sat 18-May-2013
13:57

> So much for the 'age of information'.

Well, it is the age of information. Noone said it was the age of wisdom.

Anonymous
Sat 18-May-2013

Sat 18-May-2013
14:08

> "yes" program , produced in 2009 !
>
> Thank you David McKenzie for your contribution to open source community !
>
> I am just wondering what in your background that enabled the FSF to accept such worthless contribution ?
>
> Member of what masonic lodge or what church or son of a war hero or billonarties you have to be so they accept that piece of crap ?
>
> For reference:
>
> yes command - otputs a line on tty until killed !
>
> coded and added to Linux in 2009.

Actually, "yes", which is part of GNU coreutils, has been around since, like forever, being an implementation of the same-named Unix command.

David MacKenzie is the author of many of coreutils' commands, including chgrp, chmod, chown, date, dirname, expand, fold, ginstall, groups, head, mkdir, mkfifo, mknod, nice, printenv, printf, rmdir, stty, su, tty, uname, unexpand, and obviously yes; and is co-author of many others. But he is probably best known for autotools, which is one of the most central pieces of free software, as any distribution maintainer could tell you.

Now, what have YOU done for GNU or Linux or free software?

Anonymous
Sat 24-Aug-2013
23:48

So I was searching for information about the TTY system, and where did I, by chance, end up? Thanks for the great article. :-)

/radiantx

Anonymous
Fri 6-Sep-2013
14:10

really great post ! Thanks a lot for the general understanding of the how, instead of the usual "to do this, do that" ! And I'll stop here because having three exclamation mark in one sentence is a clear sign of mental disorder, but thanks again.

Anonymous Excellent post on TTY!! Thank you -- Aananth C N

Anonymous Excellent post on TTY!! Thank you -- Aananth C N
Sun 15-Sep-2013
22:58

Anonymous thank you. very interesting.
Thu 10-Oct-2013
07:42

Anonymous
Wed 11-Dec-2013
15:19

I'll join the crowd here to say thank you for the time you spent on this article, I've appreciated it a lot.

Anonymous Thank you. I don't think this article will EVER be outdated :P
Fri 13-Dec-2013
16:13

Anonymous
Fri 27-Dec-2013
16:55

> Great article! I'm currently running a real Teletype ASR-33 on a serial port with Ubuntu. I have to use stty to set the proper parameters for an uppercase-only terminal, change tabs to spaces, etc. Is there a way to determine what port I'm logging into (ttyS0? S1? S2, S3 or S4?) Right now, I have to run stty on all ports and get an access denied error for any port I'm not connected to. I'd like to find out what port I'm on and only run the stty on that port. Thanks for your time!

Have you tried ls /dev/tty.* to get a listing of connected devices? On my Mac OS X this cues me into which tty device to use. I'm running a DEC Writer III. :)

Anonymous
Tue 11-Feb-2014
20:38

Thanks for this enlightening article! It is a bit wordy, but I completely expected that.

> I am currently working on writing a toy OS

You work for Microsoft?

Anonymous
Wed 12-Feb-2014
03:14

First teletype was made in Germany around the 30s or 40s. One of the many types of teletypes was the Hellschreiber. There were also other types but all worked the same: you punch letters and they are remote printed to one or more receiving stations. After the war in the operations of confiscating the german patents, the US developed the teletype further. That's missing from your article.

Anonymous
Fri 14-Feb-2014
09:32

> First teletype was made in Germany around the 30s or 40s. One of the many types of teletypes was the Hellschreiber. There were also other types but all worked the same: you punch letters and they are remote printed to one or more receiving stations. After the war in the operations of confiscating the german patents, the US developed the teletype further. That's missing from your article.

The Hellschreiber device is not a teletype, it's more of a facsimile machine. You pressed a letter, and a _bitmap_ of the letter was transmitted (twice to account for mechanical asynchrony). On the receiving side, marks and spaces were literally penned onto the paper, creating two copies of the text, one atop the other. Due to said asynchrony, it was often skewed, but because two copies were printed, nonetheless legible.

This is a very, very different mode of operation from the teletypes described above.

Anonymous
Fri 14-Feb-2014
09:45

> First teletype was made in Germany around the 30s or 40s. One of the many types of teletypes was the Hellschreiber. There were also other types but all worked the same: you punch letters and they are remote printed to one or more receiving stations. After the war in the operations of confiscating the german patents, the US developed the teletype further. That's missing from your article.

I don't know of the history of teletypes in Germany, but the Teletype Model 15 was first produced in 1930 and was in wide use before and after World War II. Both in military and civilian contexts.

The Model 15 wasn't the first model produced by the Teletype corporation, but it was probably the most widely used pre-war model. It

might have been the most widely used model, period.

Like I said, I don't really know anything about German teletypes, but looking at the Wikipedia page for Hellschreiber, that device is quite different than the Teletype corporation's devices. It looks like the Hellschrieber sends pixels and might actually be more similar to FAX machines than Teletype devices.

Anonymous man! you made me see the light!
Mon 17-Feb-2014
08:42

Anonymous
Thu 27-Feb-2014
08:55

> Hi--how are things in Sweden?
>
> Every once in a while I get up the ambition to complain about the width of text on a web page, and you're the lucky winner today--sorry ;-)
>
> This could be a good article--from the looks of it, it probably is--but why is it (and so many other web pages today) so wide?
>
> Checking one line at random, it is 130 characters wide:
>
> echo "Meanwhile, however, the computers — still quite large and primitive, but able to multitask — were becoming powerful enough to" | wc
> 1 20 130
>
> Oh, and I'm ignoring the stuff in the left hand panel / column--I simply horizontally scroll so that panel is not visible.
>
> .....

You mean you aren't browsing this page with a teletype?

Anonymous
Thu 10-Apr-2014
14:05

> Writing to a TTY which is stopped due to flow control, ... will block your process

I can't understand? The process will not blocked when TTY is stopped by flow control(ctrl+S), the foreground process will continue running. The only difference is I can't see the display until I type ctrl+Q again.

Thanks for you article
nyu

Anonymous
Thu 10-Jul-2014
13:03

> Writing to a TTY which is stopped due to flow control, ... will block your process
> I can't understand? The process will not blocked when TTY is stopped by flow control(ctrl+S), the foreground process will continue running. The only difference is I can't see the display until I type ctrl+Q again.

You're right Nyu... and this running process may be blocked when the TTY kernel buffer is full of non-displayed characters, if it outputs too much on stdout/stderr.
I guess the author has taken a shortcut when writing this, as the flow control stop is often used to block a too verbose process and to be able to read few lines before let it go on again.

Yves

Anonymous
Mon 14-Jul-2014
13:16

great post!!

Anonymous
Thu 24-Jul-2014
11:39

Hi!
I'm struggling with RS485 communication:
Is it possible to configure a tty to automatically raise the RTS line before sending and lower the RTS line after sending?
Thank you for your input,
Helmut

Anonymous
Mon 28-Jul-2014
13:13

the VT100 was not a colour terminal as the text suggests, though.

Anonymous
Tue 29-Jul-2014
00:07

Unix purist <- I WAS HERE +1 great article

Anonymous
Tue 5 Aug 2014

Great article - I will recommend it to my network.

Tue 5-Aug-2014
11:29

Kind regards,
Christian from Germany

Anonymous
Tue 2-Sep-2014
18:01

> the VT100 was not a colour terminal as the text suggests, though.

seems so, just checked the manuals online. http://www.vt100.net/docs/

do you perhaps know which terminal introduced color attributes for escape sequences?

Anonymous
Fri 5-Sep-2014
01:13

The Plan 9 operating system completely does away with TTY, signals, and ioctl. Everything in Plan 9 is either a file or represented as a file.

Anonymous
Sat 13-Sep-2014
14:33

The basic idea is that every pipeline is a job, because every process in a pipeline should be manipulated (stopped, resumed, killed) simultaneously.

Not just pipelines, every command is a job in shell's parlance. Job is a userspace thing, only maintained by shell, not kernel.
http://www.gnu.org/software/bash/manual/bashref.html#Shell-Commands

Niz

Anonymous
Wed 24-Sep-2014
14:34

By far the most amazing article about TTY I have ever read, thanks!

Anonymous
Sun 12-Oct-2014
19:04

Great article. Thank you! After 15 years of using unix and linux i finally got to the description of ttys :)

Anonymous

Fantastic article. Thanks for doing all of the legwork. :)

Fri 14-Nov-2014
02:31 -pf

Anonymous
Sun 16-Nov-2014
03:37
Thank you so for the article!
And now, I know the meaning of "tty" and "pty"!

Maya2003

Anonymous
Wed 3-Dec-2014
16:42
Great article, thanks a lot!

Anonymous
Wed 14-Jan-2015
21:09
This is the best article I have ever read about tty.

Thanks a lot!

Anonymous
Mon 9-Feb-2015
03:38
Thank you !!! :)

Anonymous
Sun 15-Feb-2015
23:47

Yup, interesting intro to the TTY area, I think, and fun trying out the terminal tweaks..but any ideas of where I can find a good intro to using TTY in Linux, without the writer assuming I know all about networking/the Linux kernel in advance??? Please and thank-you.

Anonymous
Sun 15-Feb-2015
23:47

Yup, interesting intro to the TTY area, I think, and fun trying out the terminal tweaks..but any ideas of where I can find a good intro to using TTY in Linux, without the writer assuming I know all about networking/the Linux kernel in advance??? Please and thank-you.

zdennis
Zach Dennis
Mon 9-Mar-2015
00:07

> Yup, interesting intro to the TTY area, I think, and fun trying out the terminal tweaks..but any ideas of where I can find a good intro to using TTY in Linux, without the writer assuming I know all about networking/the Linux kernel in advance??? Please and thank-you.

The O'Reilly book titled "Termcap & TermInfo" should get you started: http://www.amazon.com/termcap-terminfo-OReilly-Nutshell-Linda/dp/0937175226

Anonymous Sun 12-Apr-2015 00:14 — best tty explanation ever written, hands down. thank you so very much.

Anonymous
Wed 15-Apr-2015
01:21

I'm not especially new to bash programming or serial ports in general, but have never delved into stty and echo.

So I have a simple script (and I'm not sure I have the stty settings right), that should send "tx c" to the serial port. However, the best I can tell using minicom, if I perform the following from the command line:

echo tx c > /dev/ttyO1

the serial device receives:

tx.{

echo aaa > /dev/ttyO1 echos aay
echo bbbbb > /dev/ttyO1 echos bbbbz
echo abcdefghijklmnop > /dev/ttyO1 echos abeefgkijkmmnoz

WHAT am I missing?

Anonymous
Tue 21-Apr-2015
18:45

This is best ever article on TTY. What an awesome job. Please keep writing such articles.
Chakri

Anonymous Fri 8-May-2015 14:00 — The best article about tty ever! no kidding.

Anonymous Sat 25-Jul-2015 20:08 — very useful! thanks~

Anonymous
Thu 30-Jul-2015
06:41

I am new to Linux but this article is so clear even i can taste 60% of them. I will read it again a couple days later because this is the best article i ever read about linux.

Anonymous
Sat 15-Aug-2015
21:45

amazingly clear on a very fundamental piece of technology that I never really knew about. Just the build up of the subject is a great way to approach any kind of technical writing. I'll definitely keep this post in mind when I try to tackle writing about a complicated technical subject.

Anonymous
Sun 16-Aug-2015
05:57

> "yes" program , produced in 2009 !
>
> Thank you David McKenzie for your contribution to open source community !
>
> I am just wondering what in your background that enabled the FSF to accept such worthless contribution ?
>
> Member of what masonic lodge or what church or son of a war hero or billonarties you have to be so they accept that piece of crap ?
>
> For reference:
>
> yes command - otputs a line on tty until killed !
>
> coded and added to Linux in 2009.

> HISTORY
> The yes command appeared in Version 7 AT&T UNIX.

The code for this in GNU is ridiculously long, but in other systems, the code is shorter than the license or the above comment

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
if (argc > 1)
for (;;)
puts(argv[1]);
```

```
puts(argv[1]);
else
for (;;)
puts("y");
}
```

i336_
David Lindsay
Tue 18-Aug-2015
18:55

This is really cool. I've just discovered this; I might have more to add after I've properly read through it. While I remember, though, I want to mention one of the better resources I've found for terminal escape sequences: an old online manpage for dtterm(5), at

http://www2.phys.canterbury.ac.nz/dept/docs/manuals/unix/DEC_4.0e_Docs/HTML/MAN/MAN5/0036____.HTM

Also, the following Github Gist is updated extremely frequently (6 days ago, as I type this); it tracks terminal support for 24-bit color, a fairly new capability that's slowly gaining traction. You should be able to just expect it to be available 3-5 years from now; terminals in hyper-current distros like Arch likely have the support already.

https://gist.github.com/XVilka/8346728

Anonymous
Tue 22-Sep-2015
06:41

Thanks for a great explanation of infrequently discussed, arcane, but essential part of the Unix/Linux "reptile brain". Required knowledge for anyone who hopes to use Unix/Linux beyond a novice level.

Anonymous Amazing. Very very needed. Thank you very much.
Mon 30-Nov-2015
10:35

Anonymous
Fri 4-Dec-2015
09:14

Very nice summary for those born too late :-)

It might be worth mentioning that UARTs/serial ports have control lines to block transmission when the received data can't be processed as fast as it's received.
This way the UART itself would stop transmitting and the whole process was handled in hardware.
The sending process would have to wait (be blocked) for the UART to

The sending process would have to wait (be blocked) for the UART to transmit it's data anyway, as the hardware would mostly be slower than the software anyway.

However when modems came in to play, the terminals UART and the hosts UART were no longer directly wired to each other and modems would only transmit data and no control lines between them. This situation required another method of flow control that would have to be transmitted in-band with the data. So software flow control via device control characters was invented.

Anonymous
Thu 7-Jan-2016
13:45

5

Anonymous
Tue 12-Jan-2016
17:07

Really good document,thanks for your great efforts

Anonymous
Fri 5-Feb-2016
15:34

I am learning linux... and it is best article to understand what is going under the hood:) thanks for your time

Anonymous
Sat 20-Feb-2016
02:55

Good job demystifying TTY for me. Keep it up.

Cheers

Anonymous
Thu 10-Mar-2016
02:06

This and all OK.. But.. I have one doubt for long time... In case of terminal emulator we can navigate/scroll using mouse.. How do we scroll/navigate inside TTY (I am a Linux beginner.. So sorry if it's a stupid question)

lft
Linus Åkesson
Wed 30-Mar-2016
00:35

> This and all OK.. But.. I have one doubt for long time... In case of terminal emulator we can navigate/scroll using mouse.. How do we scroll/navigate inside TTY (I am a Linux beginner.. So sorry if it's a stupid question)

Hi! That is not handled by the TTY layer, it is done by the terminal emulator itself. Whenever something scrolls off-screen, it is appended to a scrollback buffer, which you can then view using GUI controls or keyboard shortcuts.

Anonymous
Thu 31-Mar-2016
00:02

Yes! More than I was looking for, but I'm not complaining. Brilliant article, and now I know enough to stay the fuck away until I must.

Anonymous really cool thanx to share your knowledge
Sat 9-Apr-2016
03:04

Anonymous
Sun 10-Apr-2016
05:34

Hello Linus! I have a question about what happens when you kill a process in raw mode. If you launch, say, vim in a terminal and then run `killall -9 vim` from a separate terminal, you would expect the original terminal to be left in raw mode after vim exits (as if you had run `stty raw` or something). However, surprisingly, it is left in canonical mode. I have tried this with multiple shells, terminal emulators, and OS's, and only urxvt behaves differently from this. My question is, what mechanism resets the terminal in this case?

-- kc

lcu
Luis Colorado
Thu 14-Apr-2016
16:19

> Hello Linus! I have a question about what happens when you kill a process in raw mode. If you launch, say, vim in a terminal and then run `killall -9 vim` from a separate terminal, you would expect the original terminal to be left in raw mode after vim exits (as if you had run `stty raw` or something). However, surprisingly, it is left in canonical mode. I have tried this with multiple shells, terminal emulators, and OS's, and only urxvt behaves differently from this. My question is, what mechanism resets the terminal in this case?
>
> -- kc

Historically, the UNIX tty driver used to reset terminal settings to known values (300bps, some parity, cooked mode, etc.) in close() callback

(which was called by the kernel only on last close to the device only) but this was found nonsense, because normal initial session programs like old /etc/getty (yes, it was stored there) were to initialize it fully. This had some drawbacks, because this last close could happen when not needed and the reasons to conserve settings between close where significant.

This requirement has been dropped from linux kernel and now linux allows you to fix terminal settings from a previous shell command without losing those settings because of a last close issue.

Anonymous
Fri 22-Apr-2016
06:21

Thank you sir. You have been helpful.

Anonymous
Mon 16-May-2016
23:57

Thank you. I don't think this article will EVER be outdated :P

The images of it already are on high resolution monitors.

Anonymous
Thu 23-Jun-2016
22:00

Best article regarding this mess!

Anonymous
Fri 15-Jul-2016
23:14

This is just awesome! I have old phone co UNIX books that don't go into this depth! Thx for this info

Anonymous
Sat 16-Jul-2016
21:19

Nice one. Thanks.

Anonymous
Sat 23-Jul-2016
16:39

I'm using python on Debian os.
I've interfaced bluetooth to uart.
The data coming from bluetooth is received and can be monitored on ttyS0.
Can someone tell me how to copy that data.
I want to paste it in a text file

Anonymous
Tue 30-Aug-2016
00:34

> "yes" program , produced in 2009 !
>
> .... cut ....
>
> For reference:
>
> yes command - otputs a line on tty until killed !
>
> coded and added to Linux in 2009.

Interesting! I remember I made a joke with yes program to my friend in 1996 by using Slackware Linux I.e. Linux kernel so new.

Anonymous
Tue 30-Aug-2016
01:03

> I'm not especially new to bash programming or serial ports in general, but have never delved into stty and echo.
>
> So I have a simple script (and I'm not sure I have the stty settings right), that should send "tx c" to the serial port. However, the best I can tell using minicom, if I perform the following from the command line:
>
> echo tx c > /dev/ttyO1
>
> the serial device receives:
>
> tx.{
>
> echo aaa > /dev/ttyO1 echos aay
> echo bbbbb > /dev/ttyO1 echos bbbbz
> echo abcdefghijklmnop > /dev/ttyO1 echos abeefgkijkmmnoz
>
> WHAT am I missing?

Not sure about your system but in my old linux systems with serial port, device was ttyS01 but not tty01.

Anonymous
Tue 30-Aug-2016
01:11

> This and all OK.. But.. I have one doubt for long time... In case of terminal emulator we can navigate/scroll using mouse.. How do we scroll/navigate inside TTY (I am a Linux beginner.. So sorry if it's a stupid question)

Shift+page up/down

This will do the job both in fyi terminal emulators and the ones on TTYs (ctrl+alt+F1, etc.).

Anonymous
Tue 30-Aug-2016
01:23

> > I'm not especially new to bash programming or serial ports in general, but have never delved into stty and echo.
> >
> > So I have a simple script (and I'm not sure I have the stty settings right), that should send "tx c" to the serial port. However, the best I can tell using minicom, if I perform the following from the command line:
> >
> > echo tx c > /dev/ttyO1
> >
> > the serial device receives:
> >
> > tx.{
> >
> > echo aaa > /dev/ttyO1 echos aay
> > echo bbbbb > /dev/ttyO1 echos bbbbz
> > echo abcdefghijklmnop > /dev/ttyO1 echos abeefgkijkmmnoz
> >
> > WHAT am I missing?
>
> Not sure about your system but in my old linux systems with serial port, device was ttyS01 but not tty01.

It seems ttyO1 works for you.
Maybe outrun a CR and LF at the end of the echoed words will help.
Indeed CR may not be needed.

Try this:
echo bbbbb\n\r > /dev/ttyO1

Hope it works *fingers crossed* :)

Anonymous
Tue 30-Aug-2016
01:29

I have been with Linux and UNIX systems since 1994. They were good old days and I am learning these amazing knowledge just now from this article.
Fantastic job Linus. Thank you so much!

Bedri Özgür Güler

Anonymous
Thu 6-Oct-2016
21:39

> * the text should be arranged to wrap to the width of the (reader's) window

Whilst I agree with you entirely, this article wraps just as you request.

Of course, it may have been altered in the many years since you wrote your comment :-) but credit where credit is due!