The very basics of a terminal emulator

2018-02-24

XTerm, st, libvte, and others. Terminal emulators. Pretty much anyone uses them – some people all day long. The big question remains, what do they do, how do they work?

Let's write a very basic terminal emulator. This has been on my TODO list for a long time. You can find my example program over here:

eduterm

Where to start?

Let's assume we're on Linux. You have probably run the "`w`" command at some point. Its output looks like this:

```
$ w
 16:43:05 up  8:24,  2 users,  load average: 0.58, 1.06, 1.26
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
void     tty1      09:31    7:11m 14:12   0.00s xinit /home/...
void     pts/10    16:42    0.00s  0.02s  0.00s w
$
```

"`tty1`" is the Linux text console where I logged in. The second line shows "`pts/10`" and this is the XTerm that I'm running "`w`" in. (It doesn't show all

the other terminals that are currently open because VTE doesn't update utmp anymore.)

The term "pts" points us into the right direction. It even has a manpage:

```
$ man pts
```

That manpage tells you almost everything you need to know. In particular:

- We're talking about "pseudoterminals". "Pseudo" because we won't be dealing with actual terminal hardware.
- There is a master and a slave file descriptor.
- The master will be used by your emulator and the slave by the program running "inside" of your emulator.

How do you get a master slave pair? That's a bit tricky. The BSDs have "`openpty()`" which does all the work for you. suckless st uses this. The manpage says that this function is "not standardized in POSIX", so, yeah, maybe you shouldn't use it? Maybe it's fine? Nobody really knows. Even musl libc has this function (and note the comment on this function being "vastly superior").

For my little example program, I went with "`posix_openpt()`". It has a different interface and requires you to do more work. It goes like this:

- `posix_openpt()` returns a file descriptor to a master device.
- `grantpt(master)` adjusts the file system permissions of the corresponding slave device, such that you will be able to open that device.
- `unlockpt(master)` "unlocks" your slave. This basically means that you are now allowed to actually open the slave device.

I'm not 100% sure why we need this little dance. The interface of "`openpty()`" feels a lot better.

Okay then. Run these three calls and you have a pair of master and slave file descriptors. It's done in the "`pt_pair()`" function in my example program.

Connecting a child process to your terminal

You basically have to do this:

```
p = fork();
if (p == 0)
{
    close(pty->master);

    setsid();
```

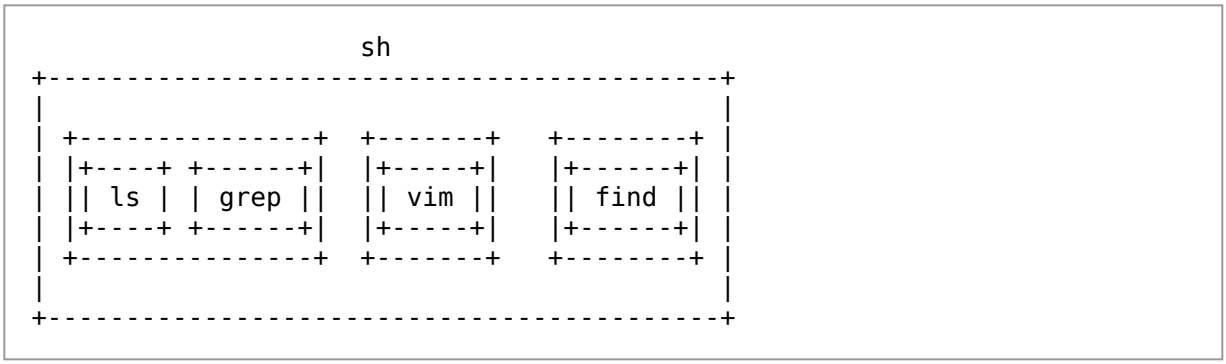
```
    if (ioctl(pty->slave, TIOCSCTTY, NULL) == -1)
    {
        perror("ioctl(TIOCSCTTY)");
        return false;
    }

    dup2(pty->slave, 0);
    dup2(pty->slave, 1);
    dup2(pty->slave, 2);
    close(pty->slave);

    execle("/bin/sh", "/bin/sh", NULL, env);
}
```

You do a fork to get a new process that inherits virtually all properties of the parent. In our child, we close the master file descriptor. The child will operate only with the slave, so we “copy” the slave file descriptor to stdin, stdout and stderr. This allows the shell to talk to us.

We also create a new session group using `setsid()`, which implicitly creates a new process group as well. These two concepts have become somewhat “arcane” these days. It’s explained over here:

https://www.win.tue.nl/~aeb/linux/lk/lk-10.html

And it looks like this (I hope this helps):



`ls`, `grep`, `vim`, and `find` are all individual processes. `ls` and `grep` are put together in a process group because let’s assume you ran “`ls | grep foo`”. `vim` and `find` are in their own process group. All of them, including `sh`, are in a session and `sh` is the leader of that session.

Let’s assume that “`ls | grep foo`” is the foreground process group. This means you have started `vim` and `find` earlier and put both of them “in the background” using ^Z. When you now hit ^C, SIGINT will be delived to the foreground process group, i.e. our pipe.

You can inspect these groups using `ps` (which does not show session IDs per default):

```
$ ps -o pid,pgid,sess,comm
  PID  PGID  SESS COMMAND
 3301  3301  3301 bash
22474 22474  3301 vim
```

```
22547 22547  3301 find
23059 23059  3301 bash
23060 23059  3301 cat
23126 23059  3301 sleep
23291 23291  3301 ps
```

Some of these processes share a process group ID and they all share the same session ID. PID and PGID are the same for most processes, which means that they are also the process group leader – so, `vim`, `find`, and `ps` are the only processes in their respective process groups. Both `cat` and `sleep` have the same PGID: I ran "`while sleep 1; do date; done | cat`" and then hit ^Z. That made the shell fork (bash with PID 23059 becomes the process group leader) and then fork again a few times to exec new programs.

Okay, enough of that. Back to the code.

We have made our terminal the controlling terminal for the entire session using the `ioctl(pty->slave, TIOCSCTTY, NULL)` call. This is probably the most interesting part in this chapter because it informs the kernel that a SIGINT shall be delivered to the foreground process group when you hit ^C in your terminal emulator.

Pseudoterminals do a lot of the work

Okay, we now have a child process running and it's connected to our slave device. You can now write to your master device and it will be delivered as input to your child.

This means that the other part of your terminal emulator can be a regular X11 client. All you have to do is convert X11 key events to "bytes" and write them to the master device. In X11 land, you get an event and a `XKeyEvent` struct with a keycode field and other fields indicating modifiers like "Shift". You have to figure out which ASCII character corresponds to this struct. Luckily, `XLookupString()` is good enough for my very simple example.

And, of course, you have to read from the master device and display any characters in an X11 window.

```
+------------+      +================+      +----------------------+
| X11 client | <--> | pseudoterminal | <--> | terminal application |
+------------+      +================+      +----------------------+
```

The "X11 client" is your terminal emulator. It doesn't have to be an X11 client, but let's not get distracted.

The "pseudoterminal" thingy is **not** part of your code. It's a driver that lives in the kernel. And it performs a surprisingly large part of the job.

For example, you don't have to deal with signals. You literally write a "^C" (a byte with value 0x03) to your master device and that's it. You don't have to keep track of "foreground process groups" and what not.

You don't even have to implement "terminal modes". Have you ever used "`read -s`" in a shell script? Try it:

```
read -sp 'Type something and hit Enter: '
echo "You said: '$REPLY'"
```

You will notice that you don't see what you type. The shell (or any other program) accomplishes this by running `ioctl(stdin, TCSETSW, ...)` to instruct "the terminal" to not display any character entered by the user. But as it turns out, this is handled by the pseudoterminal driver and not the actual terminal emulator.

At the end of the day, a terminal emulator like XTerm or my example program is nothing but a converter from X11 events to simple bytes.

Okay, what does your terminal emulator have to emulate then?

Except that it isn't. I lied.

There are some things that you don't have to implement. On the other hand, the pseudoterminal driver is ignorant of any escape sequences. And this is where the fun starts.

```
printf '\033[1mThis is bold text.\n'
```

If you run this in my example program, you won't see bold text. You won't see any kind of color. You won't be able to run curses applications. Not even `vi` works (but `ed` does, so you should be fine :-)).

It's all those escape sequences that you have to interpret. Are you compatible with a VT100? Do you understand the "non-standard" escape sequences of XTerm that allow you to use 256 colors? Do you support a secondary screen buffer or special line drawing characters?

If your user presses "cursor key up", which bytes do you write to your master device? ^C was easy because it has been mapped to 0x03 a long time ago. For cursor keys, there is no such mapping. Ever seen a stray `^[[A` in your terminal? That might have been the escape sequence generated by your terminal emulator and written to its master device to indicate "cursor up" – the terminal application has to read and interpret this sequence.

Oh and speaking of “sequences”: Since UTF-8 is a multibyte encoding, have fun dealing with that. :-) My simple example program ignores all that and just expects ASCII.

Even plain ASCII can be challenging (or “annoying”). There’s the newline character. When you read it, you have to move the cursor one line downwards. Now, what do you do when your terminal is 80 cells wide and you read “a” 80 times followed by a newline? The 80th “a” has pushed your cursor to the 81st column which does not exist, so you probably wrap to the next line. Do you then ignore the following newline? Do you process it and create two line feeds?

Even simple stuff like that can cause a minor headache. Let alone atrocities like this:

```
printf 'ä\033[120b'
```

This prints an “a-umlaut” (UTF-8!) and then prints an escape sequence which instructs the terminal to repeat the previous character 120 times.

Conclusion

The basics of a terminal emulator are relatively easy to understand. You can tell by the short Git history of my program that it didn’t take long to write. I still learned a great deal.

I’ll stop at this point. Maybe I’ll implement some more interesting features, but I will not write a new terminal emulator from scratch any time soon. There’s a reason it’s called “curses”. And reinventing the wheel really doesn’t help this time – if you want to hack on a terminal, go hack on st.

It could be interesting, though, to look at the code that implements pseudoterminals. And/or to have a look at the BSD details.

I wonder what real hardware terminals were like. I assume that “pseudoterminals” didn’t exist back then. This means that an actual VT100 had to implement stuff like “cooked mode”, “echo mode”, and all that by itself. Right?

Comments?