How different is a POS terminal from a regular computer terminal? This latter kind of terminal has two devices in it: an input device (keyboard) and an output device (screen). Such terminals generally communicate with a computer via a serial data connection, so in addition to logic to operate the separate devices in the box, there is a device to handle serial transmitting/reception (the UART: universal asynchronous receiver transmitter).

How does this look to the OS? what happens when someone types in a character from the terminal?
Terminals: OS–eye view

- user types key
- bits arrive serially at UART in computer’s serial port
- when all bits received, generate interrupt
- kernel picks up byte of data from serial port and sends it...

when all, say, 8 bits arrive from the terminal via the serial port, the serial port hardware and logic generate an interrupt to indicate that there is data that needs to be attended to. The interrupt handler, running in kernel mode (suppose we’re not using a microkernel), picks up the byte of data and now needs to figure out who it belongs to: for whom is this byte destined?
UNIX Processes and ttys

- each process that is started at a terminal has associated with it a **controlling tty**
  - i.e., UNIX treats tty as a single device that is read/write capable

- each process has 3 files open for it:
  - stdin: where input is expected
  - stdout: where output is expected to go
  - stderr: where error output is expected to go

In UNIX, each process started at a terminal keeps an association with that terminal. So, the kernel has only to determine which process is associated with the terminal (tty) the data just arrived from, and deliver that character to that process.

UNIX treats the tty device as a single device that supports reading and writing. Now, let’s jump a few levels in the hierarchy of things: from the process’ point of view, UNIX provides, for each process, three open files: stdin for input, and stdout + stderr for output.
UNIX Processes and ttys

- each process that is started at a terminal has associated with it a controlling tty
- each process has 3 files open for it
- normally, a process begun from a controlling tty has:
  - stdin = keyboard
  - stdout, stderr = display

Each process has these three files open for it. For processes bound to terminals, stdin is normally the keyboard of the tty, stdout + stderr are normally the display.

A process needs only read from stdin to get input. But stdin is not a device, it is a file that, most of the time, is associated with a device. In the case of something like assignment 2, your program is reading input from stdin, but needs to be able to control the underlying device as well…
UNIX ttys

- ttys normally operate in full duplex mode:
  - consequence: received characters must be echoed back to display in order for user to see them
- \( \Rightarrow \) tty driver program must perform echo operation

- other things this program does?

It needs to control the tty because normally the OS layers between stdin and the device are doing quite a lot of processing on your behalf. For instance, terminals operate in full-duplex (fdx) mode: any character you type in, does not automatically appear on your screen unless the computer receiving it sends it back to the display. The OS software operating the tty performs this “echo” operation automatically, so you don’t have to in every program you write.

In the case of assignment 2, though, you need to be able to get it to stop doing this echo operation.

What other processing does the tty driver do for you?
UNIX ttys

- line buffering: programs reading stdin only get complete line-at-a-time,
  - e.g., fread(mybuffer,1,256,stdin) returns only when a complete line (256 chars) is read
  - or EOF, or error

One significant thing is line buffering of input: a program doing a read() request on a tty (reading stdin when stdin is bound to a tty) blocks until an entire line has been read in (indicated by a return or newline character, i.e., the user pressed return or enter on the keyboard). So, the example fread() above will not return until 256 characters have been read (or until some unusual condition arises like an error, or an end-of-file).

Anything else the tty driver does for you?
UNIX ttys

- ever use DEL or BS to correct a typing mistake?
  - tty driver understands these special characters and performs the ‘edit’ operation
- ever use ctrl-C to stop a program?
  - tty driver understands that character to mean send a SIGINT signal to the process

⇒ tty driver processes ‘special’ characters and takes action

It also *interprets* certain characters to mean “perform an action.” That is, instead of taking every character typed in by the user and just sending each and every one on to the program, the tty driver inspects characters one at a time as they arrive. Most characters simply go on into the line buffer awaiting the newline input character that will send the entire line buffer on to the program awaiting their arrival. But some are intercepted and acted upon, and do not go into the line buffer.

One obvious character that is intercepted locally (i.e., in the tty driver) is the backspace or delete key that is used to remove the last character put into the buffer.

It’s much more convenient for programmers to have the tty driver handle this because then the programmer doesn’t have to meddle with correcting characters in the buffer it receives, it doesn’t have to worry about buffer underflow if a user types in more deletes than there were characters in the buffer, etc.

Another commonly intercepted character is ctrl-C (generated by holding down the control and C keys at the same time): normally it has the action of causing a SIGINT to be sent to the program. ctrl-D send EOF, so the reading program will see an end of file on its input stream.
So, in summary: all characters generated at the keyboard arrive at the tty-driver (where the kernel delivers them in response to the interrupt event that got the data noticed in the first place). Each character is inspected, nearly all simply go on into the buffer accumulating the current line of input (blue band in figure). But some characters are intercepted and cause actions to occur.
Normal tty settings

- terminal characteristics: (use “stty -a” command)
  - speed 38400 baud, rows = 67, columns = 108, ...

- special chars:
  - intr = ^c
  - quit = ^\n  - erase = ^h
  - kill = ^u
  - eof = ^d
  - eol = ^?
  - eol2 = ^?
  - swtch = ^
  - start = ^q
  - stop = ^s
  - susp = ^z
  - dsusp = ^y
  - rprnt = ^r
  - flush = ^o
  - werase = ^w
  - lnext = ^v

- modes:
  - parenb
  - -parodd
  - cs8
  - -cstopb
  - -hupcl
  - cread
  - -ignbrk
  - brkint
  - -ignpar
  - -parmrk
  - -inpck
  - -istrip
  - -inxan
  - -ixoff
  - -imaxbel
  - isig
  - icanon
  - -xichr
  - -echo
  - echok
  - -tostop
  - -echoctl
  - -echoprt
  - -echoe
  - -echoprt
  - -echoke
  - -echok
  - -onlcr
  - -ocrnl
  - -onocr
  - -onlret
  - -ofill
  - -ofdel

For a normal UNIX terminal (here, actually a Linux xterm), certain things are set by default (just the way signal handlers are set by default so there will be a known action for any signal).

Note in the “special chars” section the number of different characters that are intercepted to mean “do something”. As mentioned, ctrl-c (written ^C for short) is the character that causes the process to be interrupted (SIGINT), ^D sends EOF, ^\ sends a SIGQUIT to a process, ...

Note the modes: these represent the understanding the terminal and tty driver have about how to interact. Any of the modes can be on or off; if a mode is off, it shows here with a minus sign in front of it. So, odd parity is disabled: “-parodd”. You might be particularly interested in the “echo” mode which is enabled: this is the mode that controls whether the terminal driver echoes characters back or not.
Normal tty settings

- all aspects of tty operation are controllable
  - via appropriate system calls

As you would expect, all of these terminal modes can be controlled via software: they are protected operations – only the kernel can perform them – but users can, of course, ask the kernel to do the operation for them. The request succeeds if the user has permission to perform that operation (via the kernel).
UNIX devices

- All devices are represented as files
  - May be opened or closed
  - May be read and/or written
  - May be "controlled"
    - E.g., seek command on a disk drive

```c
int fd = open(filename, flags, mode);
```

- Full path to file to be opened, e.g., dev/hda1
- Read, write, r/w, etc.
- Setting access modes

One of the (at the time) innovative features of UNIX was its treatment of devices – all devices – as files. So, a terminal, a printer, a disk drive: all are handled the way a file would be, e.g., open for reading or writing (depending on what the device is capable of), etc.
UNIX device control

- basic system call:
  - `int ioctl(int fd, int req, ...);`

- valid requests depends on type of device (see, e.g., `man ioctl_list`)

- many categories of device have `ioctl()` wrappers
  - easier to work with

In addition, recognizing that devices need to be controlled to operate correctly, a system call, `ioctl()`, is provided to get current values from a device about its status and configuration, and to set device values. The argument for “req” is usually a bit mask which sets or clears certain features of the device. The actual allowed values depend on the type of device.

In many cases, programmers don’t have to use `ioctl()` directly as a series of higher-level functions is available. These functions are usually more intuitive to work with, better reflect the nature of the device being controlled from the programmer’s perspective.
UNIX device control

- for ttys use termios functions (man termios)
- e.g.,
  - int tcgetattr(int fd, struct termios *termios_p);
  - int tcsetattr(int fd, int optional_actions, struct termios *termios_p);
- use for, e.g., control of echoing, character interpretation, ...

For the tty family of devices, a family of wrapper functions makes programming easier: that family is the termios functions; see the man page for termios. You’ll see from the two examples above, that they behave much the way your signal handler functions did, i.e., there’s a way to get the current settings, and a way to set the settings to something new.

Typically for a program you want first to get the current settings before you set them to something new (so you can restore them later).