Kernel Stdio Theory

From OSDev Wiki

Contents

Standard Input/Output Theory

What is Standard Input/Output?

Standard Input and Output, and Standard Error, are streams which are implemented as part of the C Standard Library. A stream is a read/write interface for access to a file, hardware resource, or another process.

On inclusion of the `stdio.h` header, three streams are auto created, and associated with the environment's Standard Input, standard Output and standard Error streams. It is conventional, and default in most cases, for the process's Standard Output and Standard Error to be tied to the terminal which opened it. And unless StdIn is redirected, the default source of Standard Input is taken to be the keyboard.

These effects are handled by the C Library, and they interface with an underlying OS to provide access to the stream resource. Stream resources may have the following properties:

- Read/Write
- Text/Binary
- Buffered/Unbuffered

In most cases, by default, StdOut is buffered, while StdErr is unbuffered. This allows for data output on StdErr to be seen by the user immediately. Buffering may be either *line* buffered or *fully* buffered. The C Standard Library's intricacies are not the focus of this article, however.

A kernel must provide the underlying devices' API and present them to a running application. All applications, for purposes of this writing, are by default bound to

Keyboard for StdIn, and the Console for StdOut.

Since C was used to develop the Unix kernel, and Unix provided abstraction of devices as files, which were no different from any other, a process could easily open a device as a resource stream, and connect its StdIO to the device for reading or writing.

A classic example of some basic know-how of how the Unix's StdIO works is the terminal itself. Note well that provision of such abstraction does not need to be done via files. The Windows API provides abstraction via its API function calls. Programs may access StdIn, StdOut, and StdErr, but these are really only relevant when working in console mode. For GUI applications, StdIn, StdOut, and StdErr are substituted by the application developer with relevant API functionality. E.g: Alert Boxes for errors and warnings, etc.

In the Unix kernel, the default Standard Output is connected to the user's terminal device. (e.g: /dev/tty0). Try going into your Unix/Linux command prompt and typing `who am i`. This is the device that represents your current terminal's standard Output. On microcomputers (the one you're using right now) there are no terminals connected to huge mainframes, so the modern Linux/Unix microcomputers just create a terminal device anyway, and associate it with the Kernel's Standard Output.

Now: Try `echo Hello from Std Output! >file`

This prints the string "Hello from Std Output" to a file called 'file', by redirecting the Standard Output of the program to the file. The program is given the handle of the file for its stdout device, and it writes, for all intents and purposes, to the file.

Go back and find out which terminal you are, by running either `who am i` or simply typing `tty`. Take the output from `tty` and place it at the end of the line, where 'file' was last time. My kernel reports my terminal as /dev/pts/1.

Now try: `echo Hello from Std Output! >/dev/pts/1`

This time, you see the out put from 'echo' on the terminal. Why? You explicitly redirected the output of Stdout to your Stdout. /dev/pts/1 in my case, and whatever you got in yours, is synonymous with the standard output for the current user.

Interestingly, though, this file *also* acts as the Stdin device: reading from it causes all the characters you type at the keyboard to be reported to the program currently reading from it.

The unix `cat` command reads and outputs what it sees in a file. When you type at the keyboard, this is directed to Stdin, which applications may read from. To get a better understanding more quickly, try:

```
cat /dev/pts/1
```

Now keep typing words and characters, and then press enter when you feel like it. The characters you typed at the terminal are sent to Stdin, which `cat` is reading from. When you press 'enter', the kernel stops reading from the keyboard, having reached the end of line for the current bit of input, and sends this line of keyboard input to the Standard Input, which, currently, `cat` is reading from. Cat, thinking that it is reading from an ordinary file, does what it's supposed to do: echo the contents of the file onto the terminal.

When you're done typing lines, press *Ctl+d*, which is a synonym in *nix for "No more input/End of input". When dealing with files, this is synonymous for "End of file". The kernel receives the indication that there is no more input, and forwards this to the Stdin, which gives it to `cat`. `cat` interprets the "no more data" signal to mean that the end of the file it is currently reading has been reached. Just like any other file, it stops reading from the kernel. This is something common to all Unices, in that they treat everything as if it's nothing more than a normal file.

But we can understand that the dynamic nature of this file the terminal StdIO file indicates that it is obviously not a normal file on disk, but a link to a kernel API which manages Stdin and Stdout.

How can I implement StdIO in my kernel?

Short Discourse on Design Considerations

Most early developers do not take into account the fact that their kernel should implement some form of Standard Input/Output for applications. Two of the most early drivers that most OSDev-ers (not a word...) write are actually two drivers which should be part of a Standard Input/Output setup: The console driver, and the Keyboard Driver.

Note very well that Stdin and Stdout are *not* necessarily tied to the Keyboard device, or to a console screen, or other screen output. A program may open a file handle, and write to it as if it were a Std Output stream.

Now in most kernels, including the author's, implementing any form of Stdout at boot time is relatively pointless: there are no applications to use it except for the kernel. Not only that, but Stdout must be able to be tied to several streams which need kernel support.

(Todo: give examples of devices/resources (e.g. files) that can be tied to StdIO).

While the screen isn't the only device or resource to which StdOut can be tied, it is necessary to note that from all indications, all OSDev-ers write a driver that directly references the VGA text mode Framerbuffer while booting. This is not

detrimental, or wrong. but when the kernel is able to run applications, there must be a resource available in the kernel to support writing to at least the console screen as a StdOut resource. Of course, after boot, when the Graphical side of the kernel must come into play, many *still* don't write a Stdout interface because of lack of proper design.

One can generally assume that Stdout is a read or write stream that an application may access.

Let's perform a neat experiment to understand what happens with the StdIO terminal file in *Nix: Open up two terminal windows under *Nix. On one terminal, type the command: `cat /dev/pts/1` (or whatever your current terminal device is. From now on when I type /dev/pts/1 that's what I'm referring to) and press enter. You will see that Cat has begun reading from the terminal.

On the second one, type `echo <Insert random phrase here>`. Note that when the shell is reading input from the keyboard, it allows you to send non-printable characters by escaping them. If you type

```
echo hello \<enter> Jane\<enter> I\'m glad to meet you<enter>
```

You'll find that the *escaped* <enter> keypresses do not cause the `echo` command to be processed, but that the shell interprets that as a literal LineFeed character and continues to read what characters to send to `echo`. The Linefeed will be sent to `echo` just as any other character would. So you can type stuff on multiple lines, too.

What we have here is a setup where `cat` is reading from the terminal, and `echo` is writing to it. Keep the window with cat open, and keep sending `echo` commands to the terminal.

You'll find that the window with `cat` is showing nothing. Now modify your `echo` commands to write directly to the terminal:

```
echo <insert random phrase here> >/dev/pts/1.
```

What happens now? Yes. `cat` reads the characters that `echo` sends. So why didn't cat get them before? Every application has its *own* Stdout, so `echo` was writing to *its* Stdout, and not to the Terminal/System Stdout.

Returning to where we were before, I was stating that every application should have its own Stdout. This article, as yet immature, will for now only take into account one of the basic StdIn devices: the keyboard. An efficient design for tying Stdin to the Keyboard would be:

- Execute a task. Have a global variable to indicate which process is the currently active process, and therefore would be reading from the user's

keyboard input. (Note that i didn't say which is the currently *executing* process, but the currently *active* process. I.e: the one the user has open in front of him, where the currently active cursor is.

- When the keyboard IRQ is raised, the Keyboard driver fetches the keystroke from the device, and appends it to the StdIn for the active process. The active process reads from StdIn, and gets the keys placed in *its* Stdin buffer. **Question**: But what about the other applications running in the background? Like if the user has a notepad open, and a Firefox where he is posting on a forum. The Firefox window is active.

Well, take some time to think about it a bit. The processor is constantly rotating timeslices between processes. Each process which needs to read from Stdin is, of course, reading from StdIn. The *default* Stdin resource is the keyboard, as long as the Process hasn't opened another handle, and specified another resource by its handle, all reads will be assumed to be from the keyboard. But currently Stdin is tied to the *Active* Application window process. So ONLY the *Active* process actually receives the keys entered at the keyboard, since the keyboard writes directly to the specific buffer of the *Active* process. Hmmm...well? What *about* the other process windows? Do they need to know what I'm typing in a textbox element on a webpage in Firefox? No. When I switch to *their* process, and *they* become the Active Process, then when they read from Stdin, then they'll receive the Keyboard's input, too.

Makes sense, right?

That's the basic rundown of a small part of the theory. Later additions should add more body, and even implementation details.

Category: OS theory
