3 Pseudoterminals

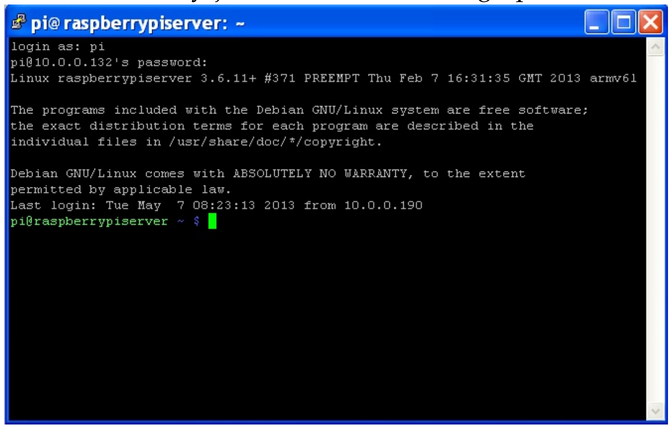
This chapter explains how to set up a pseudoterminal to control an interactive program.

3.1 Terminals

A terminal is a device for providing input to a program and printing output from the same program. The original terminals used paper. This device is called a *teletype*.

At some point, "terminals" became CRT screens,

and then eventually, just windows inside a graphical user interface.



```
login as: pi
pi@10.0.0.132's password:
Linux raspberrypiserver 3.6.11+ #371 PREEMPT Thu Feb 7 16:31:35 GMT 2013 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue May  7 08:23:13 2013 from 10.0.0.190
pi@raspberrypiserver ~ $
```

This is where we are now. Terminals still behave almost exactly the same way they did when they were invented in the 1960s.

Whenever you start a program on your computer, the program is attached to a terminal. By default, the program is attached to the terminal in which you started it. Your operating system knows how to route inputs and outputs to your program, and not to some other program or device, because *your* program is attached to *your* terminal. In fact, there are likely hundreds of terminals in use in your operating system, attached to various devices and programs. Go ahead, have a look.

Type at the command prompt:

```
$ ls /dev/tty*
```

Here are the terminals I see on my Mac.

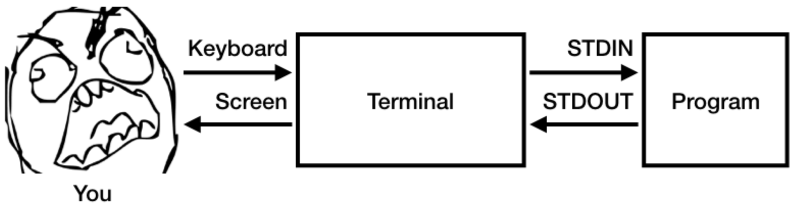
```
$ ls /dev/tty*
/dev/tty                         /dev/ttyr9                    /dev/ttyu3
/dev/tty.Bluetooth-Incoming-Port /dev/ttyra                    /dev/ttyu4
/dev/tty.MALS                    /dev/ttyrb                    /dev/ttyu5
/dev/tty.SOC                     /dev/ttyrc                    /dev/ttyu6
/dev/tty.iPhone-WirelessiAPv2    /dev/ttyrd                    /dev/ttyu7
/dev/ttyp0                       /dev/ttyre                    /dev/ttyu8
/dev/ttyp1                       /dev/ttyrf                    /dev/ttyu9
/dev/ttyp2                       /dev/ttys0                    /dev/ttyua
/dev/ttyp3                       /dev/ttys000                  /dev/ttyub
/dev/ttyp4                       /dev/ttys001                  /dev/ttyuc
/dev/ttyp5                       /dev/ttys003                  /dev/ttyud
/dev/ttyp6                       /dev/ttys004                  /dev/ttyue
/dev/ttyp7                       /dev/ttys1                    /dev/ttyuf
/dev/ttyp8                       /dev/ttys2                    /dev/ttyv0
/dev/ttyp9                       /dev/ttys3                    /dev/ttyv1
/dev/ttypa                       /dev/ttys4                    /dev/ttyv2
/dev/ttypb                       /dev/ttys5                    /dev/ttyv3
/dev/ttypc                       /dev/ttys6                    /dev/ttyv4
/dev/ttypd                       /dev/ttys7                    /dev/ttyv5
/dev/ttype                       /dev/ttys8                    /dev/ttyv6
/dev/ttypf                       /dev/ttys9                    /dev/ttyv7
/dev/ttyq0                       /dev/ttysa                    /dev/ttyv8
/dev/ttyq1                       /dev/ttysb                    /dev/ttyv9
/dev/ttyq2                       /dev/ttysc                    /dev/ttyva
/dev/ttyq3                       /dev/ttysd                    /dev/ttyvb
/dev/ttyq4                       /dev/ttyse                    /dev/ttyvc
/dev/ttyq5                       /dev/ttysf                    /dev/ttyvd
/dev/ttyq6                       /dev/ttyt0                    /dev/ttyve
/dev/ttyq7                       /dev/ttyt1                    /dev/ttyvf
/dev/ttyq8                       /dev/ttyt2                    /dev/ttyw0
/dev/ttyq9                       /dev/ttyt3                    /dev/ttyw1
/dev/ttyqa                       /dev/ttyt4                    /dev/ttyw2
/dev/ttyqb                       /dev/ttyt5                    /dev/ttyw3
/dev/ttyqc                       /dev/ttyt6                    /dev/ttyw4
/dev/ttyqd                       /dev/ttyt7                    /dev/ttyw5
/dev/ttyqe                       /dev/ttyt8                    /dev/ttyw6
/dev/ttyqf                       /dev/ttyt9                    /dev/ttyw7
/dev/ttyr0                       /dev/ttyta                    /dev/ttyw8
/dev/ttyr1                       /dev/ttytb                    /dev/ttyw9
/dev/ttyr2                       /dev/ttytc                    /dev/ttywa
/dev/ttyr3                       /dev/ttytd                    /dev/ttywb
/dev/ttyr4                       /dev/ttyte                    /dev/ttywc
/dev/ttyr5                       /dev/ttytf                    /dev/ttywd
/dev/ttyr6                       /dev/ttyu0                    /dev/ttywe
/dev/ttyr7                       /dev/ttyu1                    /dev/ttywf
/dev/ttyr8                       /dev/ttyu2
```

You can also find out to which terminal your current shell is attached.

```
$ tty
/dev/ttys000
```

A little more abstractly, you should think of a terminal as a thing with two ends. Typically, at one end of a terminal, a keyboard (input) and a screen (output) are attached, and at the other end, a program's input and output are attached.

Figure 3.1: This is a 007. My `tty` is a 000. Consider that the next time you take a late day on your homework.



3.1.1 *Controlling Programs That Attach To Terminals*

Every program designed for use on the UNIX command line interface attaches to a terminal. Because the designers of UNIX expected that users would want to control programs from other programs, command line programs frequently adhere to the following convention: input is fed to the program via a special file, called the *standard input stream*, or `stdin`, and output is printed to another special file, called the *standard output stream*, or `stdout`.[1] When you use the so-called UNIX "pipe" operators, `|`, `<`, or `>`, what you are doing is redirecting `stdin` or `stdout` to different programs or files. The ability to easily redirect inputs and outputs helps explain the relative popularity of UNIX over other operating systems among programmers and systems administrators.

[1] There is another standard output stream called the *standard error stream*, or `stderr`, that is also attached to your screen by default. `stderr` is useful for displaying diagnostic information, and since it is distinct from `stdout`, it can be silenced by redirecting it to the "system's trash can," `/dev/null`.

For example, I can use the `du` command ("disk usage") to find out the sizes of the files and folders in a directory, and then sort them, in reverse order, by their size. Pipes make doing this easy. I sometimes run these commands in order to find ways to cleanup my hard disk.

```
$ du -sk * | sort -rn
377728  install65.iso
207340  notes
120820  save
114752  customMap.pdf
32832   customMap.jpg
11072   selenium-server-standalone-3.141.59.jar
8320    swell
4672    businessCards.zip
3656    papers
3040    Feds Say That Banned Researcher Commandeered a Plane.pdf
424     version_dependencies.key
300     aslr_entropy.pdf
256     Stream under rocks Great Gulf.m4a
196     18F-CSCI-331_Intro_to_Computer_Security.pdf
164     me.png
132     two-stage-workflow
108     334_Extra.zip
100     debug.html
28      csci331_assn1_startercode
8       main.tex
8       csci331_assn1_startercode.zip
8       assignment3_code.zip
4       todos infrastructor.txt
4       meeting with david.txt
4       331 todos.txt
```

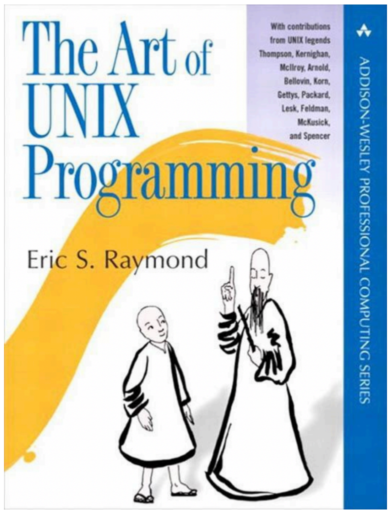

The Art of UNIX Programming
Eric S. Raymond
With contributions from UNIX legends Thompson, Kernighan, McIlroy, Arnold, Bellovin, Korn, Gettys, Packard, Lesk, Feldman, McKusick, and Spencer
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Figure 3.2: If you want to learn more about the elegant UNIX design philosophy, I recommend *The Art of UNIX Programming*, 1st edition, by Eric S. Raymond, Addison-Wesley Professional Computing, 2003. ISBN: 0131429019. This book is an easy read, but informative. Personally, it influenced me to go graduate school to study programming languages.

The things that take up the most space are at the top. Oh, it looks like I have a big ISO file that I should probably get rid of.

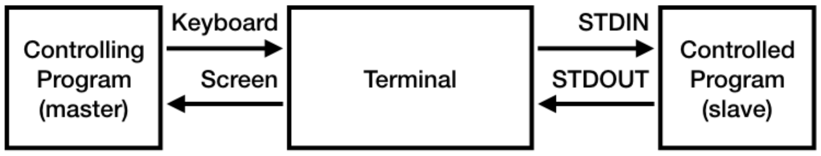
Unfortunately, the above scheme only works for so-called *batch programs*. A batch program is one that reads **all** of its input and produces **all** of its output without requiring any additional input along the way. A batch program typically reads all of its data from `stdin` and then prints everything to `stdout` and terminates. These programs are easy to redirect because they don't do anything sophisticated. Both `du` and `sort` are batch programs.

Other programs fundamentally require additional input *while* they run. This latter kind of program is called an *interactive program*. For instance, a login program may do different things depending on what you type; it may terminate immediately, or it may prompt you for additional input. If you type in the correct username and password, it grants you access. If you type in the wrong username and password, it prompts you again, or it may ask you for additional information. If you want to control one of these interactive programs with another program—for instance, a program that tries to guess passwords—this interactivity means that you can't just redirect inputs and outputs using UNIX pipe commands.

Controlling interactive programs are why we have pseudo terminals.

3.2 *Pseudo Terminals*

A pseudo terminal is what it sounds like: a fake terminal. Unlike a real terminal, a pseudo terminal lets you attach programs at both ends. You can attach a program you want to control at one end, and at the other end, instead of attaching a human, you attach a *controlling program*. The best part about pseudo terminals is that they are available via a set of standard POSIX calls,[2] so you can write code in your favorite programming language to use them.



[2] POSIX, short for "portable operating systems interface," is a mostly successful attempt to define what it is about UNIX that makes it UNIX. Programmers who write "POSIX-compatible" programs usually find it easier to get their software running on different UNIX-like operating systems, like Linux and the macOS.

The phrases "pseudo terminal" or "pseudoterminal" are long, so people often shorten this to `pty`.[3]

[3] The UNIX world is filled with these little gems of jargon, and I think it is annoying. You just have to learn the jargon if you want to play along.

Unfortunately, in POSIX, setting up a pseudoterminal is a bit of a hassle.

Because programs attached to terminals are often thought of as being controlled by humans, someone at some point thought it would be a good idea to call the human side the "master" and the controlled program side the "slave." You are likely to encounter this terminology when reading `man` pages. I am quite aware that these terms come off as tone deaf nowadays, so I will avoid using them myself. Fortunately, many in our community are aware of the problem, and we're working on it.

3.2.1 *A Helper Function That Makes Things Easier*

For the purposes of this class, I have created library called `ptyhelper` to make working with pseudo terminals straightforward. `ptyhelper` includes a function called `exec_on_pty` that you'll use to set up a `pty`. The `exec_on_pty` function calls the system's lower-level pseudo terminal functions `openpty` and `login_tty` for you. `exec_on_pty` has the following function declaration:

```
int exec_on_pty(char **argv);
```

This is a function called `exec_on_pty`, and it has one argument, `argv`. `exec_on_pty` returns a file descriptor referring to the new pseudo terminal. Here's what the argument `argv` means.

`argv` is an array of strings (i.e., command-line options) to be given to the program when `exec_on_pty` starts it up. By convention, `argv[0]` contains the name of the program to control. `argv[1]` through `argv[`*n*`]` are whatever arguments you need to pass to the program. Note that `argv[0]` should be the *full path* of the program you want to control. Also note that the last element, `argv[`*n*`]`, should be `NULL`.[4]

[4] In other words, `argv` is null-terminated.

When run, `exec_on_pty` does the following. It

1. sets up a pseudo terminal;
2. starts a *child* process for your target program and attaches one end of the pseudo terminal to it; and
3. returns a file descriptor for the *parent* side of the pseudo terminal.

Now, by manipulating the file descriptor returned by `exec_on_pty`in your parent program, you can control the child program.

Have a look at the `exec_on_pty` code when you have a minute, which is distributed as a part of your starter code in the file `ptyhelper.c`. The function is not complicated, and you'll probably learn a thing or two about systems programming.

3.3 How to Write a Control Program

Using `exec_on_pty` to control a program is easy! Here's an example.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "ptyhelper.h"

#define RESPONSE_LEN 500
#define PATH_TO_PROGRAM "./login0"

int main() {
        // to store a response from the child
        char buf[RESPONSE_LEN];

        // set up the argument array
        char* args[] = { PATH_TO_PROGRAM, NULL };

        // start child in a pty and get the fd of the pty
        int fd = exec_on_pty(args);

        // do some stuff, like
        //   read(fd, buf, RESPONSE_LEN);
        //   write(fd, ..., ...);

        return 0; // assuming all went well
}
```

3.4 Development Tips

Multiplexed file descriptor. One quirk about the pseudoterminal facility in UNIX is that it returns a *single* file descriptor over which one sends input and receives output for the child process. The significance of this fact is that you'll need to remember that the parent and child *take turns* communicating over this single file descriptor. The first trip-up that people run into is that input and output are *buffered*.

Buffered input and output. In UNIX, a stream will not usually be written out, unless

- the buffer is full, or
- a newline character, `\n`, is encountered.

A symptom of this problem is that when you send input to a controlled program, it does not respond. Often, when this happens, it's because the controlled program is waiting for you to tell it that you're done giving it input. In other words, it's waiting for you to signal that it's time for the child process to take its turn. Appending a newline character or explicitly flushing the output stream signals to the child process to proceed.

Timing. Another common issue is timing. Data flows through a pseudo terminal quickly but not instantly. If you find that your controlling program is not reading all the input sent from the program, you may want to try making it *wait*. For example, the pseudoterminal might still writing data to the child program when the controlling program attempts to read. Inserting a delay can help you work out if that's what's going on. Two library calls that can help with this are `sleep` and `usleep` (see their `man` pages) which make a program wait for seconds and microseconds, respectively.