Coyotos Bootstrap[†]

Jonathan Shapiro, Ph.D.
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

September 27, 2005

Abstract

A quick look at the Coyotos kernel bootstrap mechanism.

Contents

1 Overview

One of the annoyingly complicated parts of the EROS project was the kernel bootstrap mechanism. This was complicated for several reasons:

- The PC isn't exactly a friendly place for boot loader authors. Try writing 16-bit real mode code using a 32-bit C compiler for yourself. It's Fun!
- At the time, no general and flexible boot loader existed. Bryan Ford and Erich Stefan Boleyn were developing the grub boot loader at about the same time we did the EROS boot loader.
- EROS didn't have a file system, and all of the existing boot loaders (particularly LILO and Grub) were very filesystem oriented.
- On the PC, there is a good bit of information you need from the BIOS, and it is helpful to collect it in the bootstrap program and throw it over the wall to the kernel. How to do this was a moving target at the time, because the PC environment was changing quite rapidly (LBA, bigger drives, new and incompatible BIOS extensions, and so forth).

  Further, there were existing legacy BIOSes that we felt we wanted to continue to support, mainly because I owned a couple of machines that ran them.

As we begin the Coyotos project, matters have improved. Grub is well established, and it is fairly easy to extend. Grub2 is multiplatform, which is better still. Best of all, Grub is already designed to provide to the OS essentially all of the information that the EROS bootstrap was designed to provide (and then some).

It isn't clear yet whether Coyotos will be persistent. We have gone back and forth on this. At the moment we are leaning towards peristence. If we choose to do a conventional file system, then supporting Coyotos should simply be a matter of adding an appropriate filesystem support module to Grub. The issues I want to tackle here are:

- **Preload:** How should preloaded regions be handled?
- **Transition:** If we adopt a file system, how do we deal with the interim condition where the Coyotos file system module (if any) isn't yet integrated into the mainstream Grub distribution.
- **Compatibility:** What other boot environments (if any) need to be considered?
- **Persistence:** What should we do if we decide to go with persistence?

2 Preload

Both EROS and Coyotos will need to preload a binary image that is conceptually equivalent to an `initrd` image. The term `initrd` stands for "initial ram disk." Linux uses this mechanism to load an initial set of loadable device drivers into the kernel. In particular, the initial ram disk is used to load the *disk* drivers that are in turn used to load the rest of the system.

From the perspective of Grub, however, the `initrd` image is simply a file that is loaded into memory without interpretation. Grub handles these by providing a mechanism to load "modules" on behalf of the booted kernel, and we can use the same basic mechanism to load preloaded ranges. Grub also provides a mechanism for supplying module-specific parameters. We can hijack this mechanism to advise the kernel whether a given module is read-only, read-write, and so forth.

In EROS, and presumably in Coyotos if we go with a persistent design, the disk included a "range table." This table describes contiguous ranges on the disk, and the type of each. Some of these ranges are marked for preload. For bootstrap purposes, we can think of the disk image as a trivial file system wherein each file is a range and all files are contiguous.

In order to integrate this approach with Grub, we need to write a file system interpreter that views the range table as the top level directory of the file system. What we should probably do is add a "name" field to each range so that we can use a grub-like loading convention. In this approach, loading of named ranges corresponds to loading of named files. The names don't have to be very long or very sophisticated — the range table is not a user file system.

One very convenient aspect of this approach is that it lets us handle booting from a local floppy or file system for testing. In that scenario, each preloaded range is actually loaded from a file that is stored on some host file system (e.g. in your development `/boot` partition). This *might* turn out to be the easiest way to boot Coyotos in production. Equally important, it provides a means to deal with the second stage Grub files – we can simply stick a RAMFS image into a range somewhere.

One scenario that this *doesn't* address is ROM or network boot. More on this below.

3 Transitional Support

Whatever form the Coyotos on-disk image takes, it is not supported by the current Grub implementation, and it will not *be* supported during some transitional phase. While it is fairly simple to add the necessary support to Grub, there *may* be a delay in integration, and there *will* be a delay due to release propagation.

The workaround for this is fairly straightforward: as part of the Coyotos distribution, we supply a Coyotos-specific version of Grub. This version knows how to deal with the Coyotos disk image. When we need to boot up on systems that run earlier (non-compatible) versions of Grub, we can proceed in one of two ways:

1. Boot using the non-compatible grub, which boots us (i.e. *our* grub) as though we were a raw bootable partition.

2. Rearrange things to boot *our* grub as the primary boot loader.

Either approach will work. The pleasant thing about this is that we can distribute a Grub variant that understands Coyotos disk images and be able to use it immediately.

4 Compatibility

There are really two compatibility issues to consider, and they are independent:

- Are there environments where Grub cannot be used?

- If so, should we continue to use the multiboot mechanism for providing environment characterization to the booted kernel?

Regrettably, there *are* environments where booting with Grub is difficult:

- ROM-based systems, which provide no file system.

- Legacy network boot systems, notably NetBoot and PXE, where in some cases the network bootstrap code doesn't support the multiboot specification (e.g. `pxelinux`).

For such systems, I think the right approach is to build (or borrow) an appropriate bootstrap loader for that application, and modify it to pass the minimally required environment information to the booted kernel using the multiboot interface specification. Minimally required information includes the memory environment and the initial console graphics configuration (if any). The goal is to end up in a position where all versions of the kernel assume that they get a multiboot compliant handoff.

5 Persistence

If we adopt a persistent design, then the system must (in effect) boot from a database. As discussed above, we can view the range table as a degenerate non-hierarchical file system, and adding Grub support for this should not be difficult. The later stages of grub itself can be loaded from within a Coyotos/EROS disk range.

A Change History

This section is an attempt to track the changes to this document by hand. It may not always be accurate!

A.1 Changes in version 0.1

- First document version.