AGH
University of Science and Technology in Kraków

Faculty of Computer Science, Electronics and Telecommunications
Department of Computer Science

Master of Science Thesis

Radosław Szymczyszyn

Making DragonFly BSD operating system compliant with the Multiboot specification

Supervisor:
Marcin Kurdziel Ph.D.

Kraków 2014

Abstract

The evolutionary development of processor architectures, requirement of maintaining backwards compatibility and design errors lead to a lot of complications for the operating system developer. Abstracting away the boot time peculiarities and legacy cruft allows for simple and fast implementation of novel operating system concept prototypes, new hypervisors for the purpose of virtualization and improvement of the structure and maintainability of existing systems. I describe the Multiboot Specification which provides such an abstraction and how DragonFly BSD, a mature UNIX-derived operating system, can be modified to conform to that specification along with an implementation for the Intel 386 architecture. I present the changes made to GRUB, a bootloader implementing the specification, in order to allow it to boot DragonFly BSD. I also pinpoint an issue with the modern x86-64 architecture and the negative impact a wrong CPU design decision may have on the whole boot process.

Contents

1 The Boot Process

Wikipedia states that *an operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs.* In other words, an operating system is a computer program which allows other programs to run. What, then, allows the operating system to run if itself it cannot rely on an operating system? Especially, what does start the operating system?

The problem at hand is as old as computing itself. So is the concept of *bootstrapping* or, to put it simply, starting the computer. Bootstrapping, *booting* for short, relies on the machine being hardwired to read a simple program from a known beforehand location (e.g. a hard drive or a network interface) and running it. That program is called *the bootloader* and is responsible for loading the operating system. A more detailed description of the process is given in *2 Booting a BSD system*.

The multitude of problems involved in implementing a bootloader for each combination of an operating system and hardware platform in use led Bryan Ford and Erich Stefan Boleyn to devise the Multiboot Specification. The specification defines an interface between a universal bootloader and an operating system. One implementation of the specification is GRUB – the bootloader which came to existence thanks to the effort of GNU and is one of the most widely used bootloaders in the FOSS (Free and Open Source Software) world. More on the specification and GRUB is available in *3 The Multiboot Specification and GRUB* section.

The contributions of this paper are following:

- *2 Booting a BSD system* gives an approachable introduction to the boot process of a BSD-like operating system on the Intel x86 architecture. This section should also make the need for simplification of the boot process obvious.
- The rationale behind *the Multiboot Specification and GRUB* (section 3) which provide an abstraction over the hardware specifics an operating system programmer must overcome to bootstrap the system.

  This section shows how GRUB makes the boot process seem simpler than it really is.
- Section *4.3 DragonFly BSD and GRUB on x86* provides a description of changes necessary to make the system conform to the version of the specification targeted at the 32bit Intel architecture.

  In fact, this section describes all the changes which were applied to the DragonFly BSD kernel in order to make it fully functional when booted by GRUB. These changes include, but are not limited to, adjusting the kernel linker script, modifying the existing entry point written in assembly language and finally enabling the system to interpret boot information passed by GRUB in order to mount the root file system.

  This also includes extending the GRUB bootloader by writing a module for recognizing a new partition table type.

This is the core part of this paper.

- *4.4 DragonFly BSD and GRUB on x86-64* covers why the same approach can't be taken on the x86-64 architecture and how, even in the light of these differences, the system could be modified to work with GRUB on this architecture.

2 Booting a BSD system

The contents of this section are heavily (though not entirely) based on the outstanding work of the authors of the FreeBSD Architecture Handbook; namely on chapter 1. Bootstrapping and Kernel Initialization and on the analysis of FreeBSD and DragonFly BSD source code.

Though the description in this section is far from being simple, its every paragraph is only a simplification of what actually happens – a lot of details are omitted.

2.1 BIOS

The BIOS Boot Specification defines the behaviour of a PC (personal computer) just after power on when it is running in *real mode*. The real mode means that memory addresses are 20 bit wide which allows for addressing up to 1MiB of memory. This must be enough for all software running prior to the processor being switched to *protected mode*.

The value of *instruction pointer* register just after the boot up points to a memory region where BIOS and its POST (Power On Self Test) code is located. The last thing done by this code is the loading of 512 bytes from the MBR (Master Boot Record – usually the hard drive) and running the code contained within them.

2.2 First stage: `boot0`

The code contained in MBR is `boot0` - the first stage of the BSD bootloader. `boot0` hardly knows more than absolutely necessary to boot the next stage; it understands the partition table and can choose one of the four primary partitions to boot the later stage from. After the choice is done it just loads the first sector of that partition and runs it, i.e. it runs `boot2`[1].

2.3 Second stage: `boot2`

`boot2` is aware of the possibility of multiple hard drives in the PC; it also understands the file system structures. Its aim is to locate and run the `loader` – the third stage of the bootloader which is responsible for loading the kernel.

[1] Why not `boot1`? `boot1` actually exists. It is used when booting from a floppy disk, which is rarely the case nowadays. It performs the role equivalent to `boot0`, i.e. finding and loading `boot2`, with the difference of being run from a floppy.

`boot2` also switches the CPU to the aforementioned protected mode. The main characteristics of this mode are 32 bit memory addresses and a *flat memory model*[2].

One more thing `boot2` does before running `loader` is initializing the first few fields of the `struct bootinfo` structure which is passed to the `loader` and later to the kernel. This structure is the main interface between the BSD bootloader and the kernel and contains basic information about the kernel (its location), the hardware (disk geometry as detected by the BIOS), available memory, preloaded modules and the environment (variables configuring the kernel and the modules).

2.4 Third stage: `loader`

The `loader`, already running in the protected mode, is actually quite a capable piece of software. It allows for choosing which kernel to boot, whether to load any extra modules, configuring the environment, booting from encrypted disks. It is an ELF binary as is the kernel itself. In case of a 64 bit system, the `loader` is capable of entering the *long mode*[3], turning on paging and setting up the initial page tables (the complete setup of the virtual memory management is done later in the kernel).

2.5 x86 kernel

Once loaded, the kernel must perform some initialization. After the basic register setup there are three operations it performs: `recover_bootinfo`, `identify_cpu`, `create_pagetables`.

On the Intel x86 architecture the `identify_cpu` procedure is especially hairy as it must differentiate between all the possible CPUs in the line from 8086. The first processors did not support precise self-identification instructions so the identification is a trial and error process. Discriminating between versions of the same chip manufactured by different vendors is in fact done by checking for known vendor-specific defects in the chip.

`create_pagetables` sets up the page table and after that enables paging. After doing that a fake return address is pushed onto the stack and return to that address is performed – this is done to switch from running at low linear addresses and continue running in the virtualized address space. Then, two more functions are called: `init386` and `mi_startup`. `init386` does further platform dependent initialization of the chip. `mi_startup` is the machine independent startup routine of the kernel which never returns – it finalizes the boot process.

[2]Flat memory model essentialy means that the whole available memory is addressable in a linear space. All segment registers may be reset to 0 – the whole memory may be viewed as one huge segment.

[3]Long mode is the mode of execution where the processor and the programmer are at last allowed to use the full width of the address bus. In theory. In practice, at most 48 bits of the address are actually used as there is simply no need to use more with today's amounts of available memory.

2.6 x86-64 kernel

On x86-64 initialization of the kernel is performed slightly differently and in fact is less of a hassle. As `loader` has already enabled paging as a requirement to enter the long mode the CPU is already running in that mode and the jump to the kernel code is performed in the virtual address space. The kernel does the platform dependent setup and calls `mi_startup` (the machine independent startup).

3 The Multiboot Specification and GRUB

The described boot procedure and software is battle proven and works well for FreeBSD as well as, with minor changes, DragonFly BSD. However, no matter how fantastic software the BSD bootloader is there is one problem with it – it is **the BSD** bootloader.

In other words, the bootloader is crafted towards a particular operating system and hardware platform. It will not boot Linux or any other operating system which significantly differs from FreeBSD.

Such high coupling of the bootloader to the operating system was one of the reasons behind the Multiboot Specification. Other motives behind the specification are:

- the reduction of effort put into crafting bootloaders for disparate hardware platforms (much of the code for advanced features[4] will be platform independent),
- simplifying the boot process from the point of view of the OS programmer (who is not interested in the low level post-8086 cruft),
- introducing a well defined interface between a bootloader and an operating system (allowing the same bootloader to load different OSes).

The bootloader implementing the Multiboot Specification is GNU GRUB. Thanks to the modular architecture and clever design, GRUB is able to run on multiple hardware platforms, support a range of devices, partition table schemes and file systems and load different operating systems. Is is also possible to use it as a Coreboot payload. GRUB also sports a modern graphical user interface for a seamless user experience from the boot to the desktop environment login screen.

The availability of GRUB is a major step towards simplification of the boot process from the OS programmer point of view.

[4] E.g. a graphical user interface implementation most probably will not require platform specific adjustments; same goes for booting over a network (with the exception of a network interface driver, of course).

4 DragonFly BSD and GRUB

The focus of this paper is to describe all changes necessary to the DragonFly BSD kernel and the GRUB bootloader to make both interoperate as described in the Multiboot specification. In other words, all the changes necessary to make GRUB load and boot DragonFly BSD using the Multiboot protocol. This might lead to the question – can GRUB boot DragonFly BSD using any other protocol? It turns out that it can.

4.1 State of the Art

Besides[5] loading and booting operating system kernels, GRUB is capable of so called *chain loading*. This is a technique of substituting the memory contents of the running program with a new program and passing it the control flow. The UNIX `exec(2)` system call is an application of the same technique.

By chain loading GRUB is able to load other bootloaders which in turn might boot operating systems that GRUB itself can't (e.g. Microsoft Windows) or perform specific configuration of the environment before running some exotic kernel with unusual requirements.

That is the approach commonly used with many BSD flavours (of which only NetBSD supports the Multiboot protocol) and DragonFly BSD is not an exception. That is, the only way to boot DragonFly BSD using GRUB before starting this project was to make it load `dloader` and delegate to it the rest of the boot process.

However, this defeats the purpose of Multiboot and a uniform OS-kernel interface. The hypothetical gain of decreased maintenance effort thanks to one universal bootloader doesn't apply anymore due to the reliance on `dloader`. Neither the seamless graphical transition from power-on to useful desktop is achievable when relying on chain loading.

Therefore, chain loading is unsatisfactory.

4.2 Making GRUB understand `disklabel64`

One of the things a bootloader does is understanding the disk layout of the machine it is written for – the partition table and file system the files of the operating system are stored on.

Traditionally, systems of the BSD family (among with Solaris back in the day called SunOS) used the `disklabel` partitioning layout. Unfortunately, DragonFly BSD has diverged in this area from the main tree. It introduced a new partition table layout called `disklabel64` which shares the basic concepts of `disklabel` but also introduces some incompatibilities:

- partitions are identified using Globally Unique Identifiers (GUIDs),

[5] In fact it's not true that GRUB can chain load besides booting OS kernels. Chain loading is *how* it boots both those kernels and loads other bootloaders.

- fields describing sizes and offsets are 64 bit wide to accommodate the respective parameters of modern hardware.

TODO: there could be an appendix on BSD partitioning-related parlance: slices, partitions, labels.

GRUB is extensible with regard to the partition tables and file systems it is able to understand. Although the variants of `disklabel` used by disparate BSD flavours differ, all of them have already been supported bu GRUB before this project was started. Unfortunately, that wasn't the case for DragonFly BSD's `disklabel64`. It is very important, because one piece of the puzzle is booting the kernel but another one is finding and loading it from the disk.

Fortunately, the main file system used by DragonFly BSD is UFS (the Unix file system) which is one of the core traditional Unix technologies and is already supported by GRUB.

Alas, to get to the file system we must first understand the partition table.

Extending GRUB to support a new partition table or file system type is essentially a matter of writing a module in the C language. Depending on the module type (file system, partition table, system loader, etc) it must implement a specific interface.

The module is compiled as a standalone object (`.o`) file and depending on the build configuration options either statically linked with the GRUB image or loaded on demand during the boot up sequence from a preconfigured location.

As the `disklabel64` format is not described anywhere in the form of written documentation the GRUB implementation was closely based on the original header file found in the DragonFly BSD source tree (`sys/disklabel64.h`) and the behaviour of the userspace utility program `disklabel64`.

The module responsible for reading `disklabel64` this section refers to is already included in GRUB. That is one of the main contributions of the project this paper is about.

4.3 DragonFly BSD and GRUB on x86

TODO: declare why grub can't read disklabel

The other aim of the paper is the description of enabling GRUB to read the custom DragonFly BSD partition table – `disklabel64`.

4.3.1 How does GRUB identify the kernel image?

TODO: describe embedding the multiboot header, linker script, asm declarations

Embedding the Multiboot header

Modifying the linker script

4.3.2 Booting the 32 bit kernel

In case of the x86 variant of DFly (DragonFly BSD) the solution is straightforward. Instead of expecting the `struct bootinfo` structure the kernel must be able to interpret the structures passed from GRUB which the Specification describes in detail. However, in order to maintain compatibility with the current bootloader a new entry point must be introduced into the kernel instead of simply changing the current one to support only the Multiboot approach. All in all the two entry points should converge before calling the platform dependent `init386` initialization procedure. The rest of the system should not need to be aware of what bootloader loaded the kernel.

Adjusting the entry point

Mounting the root file system

4.4 DragonFly BSD and GRUB on x86-64

In case of the x86-64 architecture the problem is more complicated. The Multiboot Specification defines an interface only for loading 32 bit operating systems due to two reasons.

Firstly, when the specification was defined in 1995, the x86-64 was still to be unknown for the next 5 years.[6]

Secondly, the AMD64 (the standard describing the x86-64 instruction set) requires the entry to the long mode be preceded by enabling paging and setting a logical to physical address mapping. Choosing any scheme for this mapping limits the freedom with respect to the operating system design. In other words, the mapping initialized by the bootloader would be forced onto the to-be-loaded kernel. The kernel programmer would have two choices: either leave the mapping as is or write some custom code to reinitialize the page table hierarchy upon entry into the kernel. The former is limiting. The latter would defeat the initial purpose of the specification, i.e. to make the OS startup procedure as simple as possible.

Given the above, from the point of view of creating a universal bootloader **the CPU design decision to require enabling of the virtual addressing before entering the long mode is a flaw.** The CPU should be able to enter the long mode with a simple one-to-one logical-to-physical address mapping; the bootloader would then be able to load the 64 bit kernel anywhere into the 64 bit addressable memory and run it; the kernel itself would be responsible for setting up the memory mapping scheme according to its own requirements.

[6] According to Wikipedia: AMD64 was *announced in 1999 with a full specification in August 2000.*

4.4.1 The workaround

Given the aforementioned limitations of GRUB and the CPU the cleanest possible way of loading the 64 bit kernel is out of reach. It does not mean, however, that adapting the x86-64 DragonFly BSD kernel to the Multiboot Specification is impossible.

The idea is to embed a portion of 32 bit code inside the 64 bit kernel executable and only for the sake of the bootloader pretend to be a 32 bit binary.

This code logic would be similar to the code found in the 64 bit extension of the BSD `loader`, i.e. it would set up paging, enter the long mode and jump to the 64 bit kernel entry point.

Implementation of this approach is yet to be carried out.

5 Related work

There is a number of projects revolving around the issue of bootstrapping.

Coreboot is a BIOS firmware replacement. It is based on the concept of *payloads* (standalone ELF executables) which it loads in order to offer a specific set of functionality required by the software which is to run later. The usual payload is Linux, but there is a number of others available: SeaBIOS (offering traditional BIOS services), iPXE/gPXE/Etherboot (for booting over a network) or GNU GRUB. Thanks to the number of payloads Coreboot is able to load most PC operating systems.

Coreboot has a broad range of capabilities but as a firmware replacement it is intended for use by hardware manufacturers in their products (motherboards or systems-on-chip) in contrast to GRUB which is installable on a personal computer by a power-user.

UEFI (Unified Extensible Firmware Interface) is a specification of an interface between an operating system and a platform firmware. The initial version was created in 1998 as *Intel Boot Initiative*, later renamed to *Extensible Firmware Interface*. Since 2005 the specification is officially owned by the *Unified EFI Forum* which leads its development. The latest version is 2.4 approved in July 2013.

UEFI introduces processor architecture independence, meaning that the firmware may run on a number of different processor types: 32 or 64 bit alike. However, the OS system must size-match the firmware of the platform, i.e. a 32 bit UEFI firmware can only load a 32 bit OS image.

GPT (GUID Partition Table) is the new partitioning scheme used by UEFI. GPT is free of the MBR limitations such as number of primary partitions or their sizes still maintaining backwards compatibility with legacy systems understanding only MBR. The maximum number of partitions on a GTP partitioned volume is 128 with the maximum size of a partition (and the whole disk) of 8ZiB (2^{70} bytes).

In essence, UEFI is similar to the Multiboot Specification addressing the same limitations of the BIOS and conventional bootloaders. However, the Multiboot Specification was intended to provide a solution which could be retrofitted onto already existent and commonly used hardware, while UEFI is aimed at deployment on newly manufactured hardware. The Multiboot Specification is also a product of the Free Software community in contrast to the UEFI which was commercially backed from the beginning. The earliest version of the Multiboot Specification also predates the earliest version of UEFI (then known as Intel Boot Initiative) by 3 years.

6 Conclusions

The evolutionary development of processor architectures, requirement of maintaining backwards compatibility and design errors lead to a lot of complications for the operating system developers.

Even the newest architecture designs are not free of flaws such as the x86-64 CPU's requirement of enabling virtual memory addressing before entering the long mode.

However, with clever software design it is possible to abstract away most of the boot time peculiarities and cruft from the OS while initiatives like the Multiboot Specification and UEFI provide a clean interface for new and existing OS implementations.

The extension of the Multiboot Specification to cover loading of 64 bit operating systems might be an interesting path of research. This might be achieved by constructing a generally acceptable logical to physical memory mapping for at least the size of the kernel (contained inside the ELF binary) and spanning the whole range of addresses the kernel is linked to use. However, the concept needs thorough evaluation.

7 Literature

TODO: embed bibtex or whatever makes sense, for now it's just copy-n-paste

7.1 Printed

1. The DragonFlyBSD Operating System, dragonflybsd.asiabsdcon04.pdf
2. The Design and Implementation of the 4.4BSD Operating System, design-44bsd-book.html
3. Intel 64 and IA-32 Architectures Software Developer's Manual, IA32-1.pdf and other IA32-XYZ.pdfs
4. Introduction to 64 Bit Intel Assembly Language Programming for Linux, Ray Seyfarth

5. Operating System Concepts, Silberschatz, Galvin, Gagne, silberschatz-operating-system-concepts.pdf
6. BIOS Boot Specification, Version 1.01, Jan 11, 1996, specs-bbs101.pdf
7. The UNIX Time-Sharing System, D. M. Ritchie and K. Thompson, 1978, ritchie78unix.pdf
8. Tool Interface Standard Executable and Linking Format Specification, Version 1.2, TIS Comitee, May 1995, elf.pdf
9. Intel 80386 Programmer's Reference Manual, 1986, i386.pdf
10. PC Assembly Language, Paul A. Carter, Nov 11, 2003, pcasm-book.pdf

7.2 Web

1. Multiboot Specification version 0.6.96, http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
2. FreeBSD Architecture Handbook, http://www.freebsd.org/doc/en/books/arch-handbook/
3. The new DragonFly BSD Handbook, http://www.dragonflybsd.org/docs/newhandbook/
4. GNU GRUB Manual 2.00, http://www.gnu.org/software/grub/manual/grub.html
5. MIT PDOS Course 6.828: Operating System Engineering Notes, Parallel and Distributed Operating Systems Group, MIT, http://pdos.csail.mit.edu/6.828/
6. Operating System Development Wiki, http://wiki.osdev.org/

7.3 More refs

1. asm64-handout.pdf - AT&T asm syntax examples, lot of refs

TODO: clean up the references/citations stuff up

(see Hsu)

Hsu states some stuff.

8 References

Hsu, Jeffrey M. The dragonFlyBSD operating system.