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

Aware of criminal liability for making untrue statements I declare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

I describe how DragonFly BSD, a mature UNIX-derived operating system, can be modified to conform to the Multiboot Specification, which provides a uniform interface between a bootloader and an operating system. In particular, I present an implementation for the Intel 386 architecture and sketch a solution for the x86-64 architecture. I demonstrate the changes made to GRUB, a bootloader implementing the specification, in order to allow it to boot DragonFly BSD. I also pinpoint certain issues in GRUB with respect to the x86-64 architecture.

Contents

1 Introduction

The concept of *bootstrapping*, or *booting* for short, is as old as computing itself. Bootstrapping relies on the machine being hardwired to read a simple program from a location known beforehand (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.

Usually, a bootloader is crafted towards a particular operating system and hardware platform, as is the case with `loadlin`[1] or LILO[2] for Linux, or `loader` for systems from the BSD family.

The multitude of problems involved in implementing a bootloader for each combination of an operating system and hardware platform, as well as high coupling of the bootloader to the operating system, were the reasons which led Okuji et al. (2006) 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 – one of the most widely used bootloaders in the FOSS (Free and Open Source Software) world, which came to existence thanks to the effort of FSF[3]. Other motives behind the Multiboot specification are:

- 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).

GNU Hurd[5], NetBSD[6], NOVA[7] and Xen hypervisors, L4 family microkernels, and VMware ESXi are examples of kernels supporting the Multiboot specification.

---

[1] `loadlin` was a bootloader common in the early days of Linux, when it was used to load the kernel from a DOS or pre-NT Windows environment; see Wikipedia (2014b) article on `loadlin`.

[2] LILO was another Linux-only bootloader, used in many distributions before GRUB came into popularity; see Wikipedia (2014a) article on LILO.

[3] Free Software Foundation (http://www.fsf.org/) is the entity behind GNU, the userland part of the majority of Linux distributions.

[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).

[5] See GNU Hurd contributors (2010).

[6] See Vidal (2007).

[7] See Steinberg and Kauer (2010).

DragonFly BSD is an operating system based on FreeBSD, but with a number of different architectural and design decisions, scaling onto multi-core CPUs being one of them. GRUB is able to boot the system via *chain loading*, i.e. by loading the bootloader of DragonFly BSD. The same technique is used to make GRUB load a Microsoft Windows system.

This work describes the implementation of the Multiboot specification in DragonFly BSD, so that chain loading it is no longer necessary[8]. The implementation was performed for x86 architecture. A sketch of how to perform a similar implementation for x86-64 is also presented. Special care had to be taken to ensure compatibility with the existing booting strategy, i.e. all changes done to the kernel had to be backwards compatible with `dloader` (the native DragonFly BSD bootloader), so as not to break the already existing boot path. The work also demonstrates changes needed in GRUB to make it load the DragonFly BSD kernel. These changes are now part of the official GRUB code[9]. The following topics are elaborated on in the next sections:

- *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 obvious the need for simplification of the boot process.

- *3. The Multiboot Specification and GRUB* provides a rationale for 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.

- *4. Booting DragonFly BSD with GRUB* provides a description of changes necessary to make the system conform to the Multiboot specification. This is the core part of this work.

  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 on the 32bit Intel architecture. 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.

---

[8]The repository with modified DragonFly BSD source code is available at https://github.com/lavrin/DragonFlyBSD

[9]Code committed to the GRUB project is available at http://git.savannah.gnu.org/cgit/grub.git/commit/?id=1e908b34a6c13ac04362a1c71799f2bf31908760

- *5. Booting DragonFly BSD with 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 (The FreeBSD Documentation Project 2014), 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 bits 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*.

Just after the boot up the value of the *instruction pointer* register 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 – the beginning of the hard drive) and running the code stored therein.

2.2 First stage: `boot0`

The code contained in MBR is `boot0` - the first stage of the BSD bootloader. `boot0` knows nothing 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 this choice is done it just loads the first sector of the chosen partition and runs it, i.e. it runs `boot2`[10].

2.3 Second stage: `boot2`

`boot2` is aware of the possibility of multiple hard drives in the PC and it also understands 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. `boot2` also switches the CPU to the aforementioned protected

---

[10]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.

mode. The main characteristics of this mode are 32 bit memory addresses and a *flat memory model*[11].

One more thing `boot2` does before running `loader` is initializing the first few fields of the `struct bootinfo` structure. The structure is passed to the `loader` and later to the kernel and is the main interface between the BSD bootloader and the kernel. It 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` (known as `dloader` in DragonFly BSD) is already running in protected mode and 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, and 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*[12], 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 x86 line. 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 then enables paging. After 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

---

[11] Flat memory model essentially 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.

[12] 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 bus are actually used as there is simply no need to use more with the amount of memory available in a contemporary computer.

addresses and continue running in the virtualized address space. Two more functions are called afterwards: `init386` and `mi_startup`. `init386` does further platform dependent initialization of the chip. `mi_startup` is the machine independent startup routine of the kernel. This routine never returns – it finalizes the boot process.

2.6 x86-64 kernel

On x86-64 initialization of the kernel is performed a bit 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 boot procedure and software described in section 2 is battle proven, works well for FreeBSD and, 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, for that matter, any other operating system which significantly differs from FreeBSD.

The bootloader implementing the Multiboot Specification is GNU GRUB. Apart from the points already mentioned in section *1. Introduction*, GRUB, thanks to the modular architecture and clever design, is able to run on multiple hardware platforms, support a range of devices, partition table schemes and file systems and load a number of operating systems. It is also possible to use it as a Coreboot payload[13]. GRUB also sports a modern graphical user interface, for a seamless user experience from as early as the boot menu to as far as 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.

3.1 State of the Art

The focus of this work is to describe all changes necessary to make both the DragonFly BSD kernel and the GRUB bootloader interoperate as described in the Multiboot specification. In other words, all the changes needed 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.

Besides loading and booting operating system kernels, GRUB is capable of so called *chain loading*[14]. 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.

---

[13]Coreboot is a BIOS firmware replacement. Payloads are standalone ELF executables that Coreboot can use in order to support some desired functionality.

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

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 Booting DragonFly BSD with GRUB

In this section I will describe what was necessary to make GRUB understand `disklabel64`, the DragonFly partition table, and what steps allowed GRUB to load the DragonFly kernel. This section contains details of the actual implementation carried out as part of the project, where applicable referring to the source code in question.

4.1 Making GRUB understand `disklabel64`

The module for reading `disklabel64` described in this section is already included in GRUB[15]. That is one of the main contributions of this work.

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

Traditionally, systems of the BSD family (along 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),
- fields describing sizes and offsets are 64 bit wide to accommodate the respective parameters of modern hardware.

GRUB is extensible with regard to the partition tables and file systems it is able to understand. Although variants of `disklabel` used by existing BSD flavours differ, all of them had already been supported by 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.

---

[15]Code committed to the GRUB project is available at http://git.savannah.gnu.org/cgit/grub.git/commit/?id=1e908b34a6c13ac04362a1c71799f2bf31908760

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 from a preconfigured location during the boot up sequence.

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 on the behaviour of the userspace utility program `disklabel64`.

4.1.1 GRUB source code organization

During this project, the revision control system of GNU GRUB changed from Bazaar to Git. As of writing this work, the code is located at http://git.savannah.gnu.org/grub.git.

GRUB uses, as one might expect, GNU Autotools as its build system. Since the source tree, while well organized, might be intimidating at first glance, a short overview of its contents follows.

The project comes with a set of helper utilities meant to be run from a fully functional operating system. These are, among others, `grub-file`, `grub-install`, `grub-mkrescue`. The last one is particularly useful for creating file system images containing a configured and installed GRUB with a set of arbitrary files, e.g. a development kernel. Use of this command greatly simplifies and speeds up the development process. All the code meant to be run from an operating system is located in the main project directory.

Code intended to be run at boot time is located under `grub-core/`. In general, the structure follows the `grub-core/SUBSYSTEM/ARCH/` pattern, where `grub-core/SUBSYSTEM/` contains generic code of a given subsystem, while each `grub-core/SUBSYSTEM/ARCH/` subdirectory contains platform specific details. Some of the subsystems are:

- `boot/` – boot support of GRUB itself, e.g. the code which runs just after GRUB is loaded by BIOS on an x86 machine,
- `commands/` – implementation of commands available in the GRUB shell,
- `fs/` – file system support, e.g. `btrfs`, `ext2`, `fat`, `ufs`, `xfs`,
- `gfxmenu/` – graphical menu for choosing from the available boot options,
- `loader/` – loaders for different kernels and boot protocols, e.g. Mach, Multiboot, XNU (i.e. the Darwin / MacOS X kernel),
- `partmap/` – partition table support, e.g. `apple`, `bsdlabel`, `gpt`, `msdos`,
- `term/` – support for a textual interface through a serial line or in a graphical mode,

- `video/` – graphical mode support for different platforms and devices.

It is worth noting that GRUB deliberately contains almost no code for writing data to file systems – that's a guarantee that it can't be responsible for any file system corruption.

4.1.2 `part_dfly` GRUB module implementation

The newly added support for `disklabel64` partitioning scheme was located in `grub-core/partmap/dfly.c` file. In order to make the new module build along with the rest of GRUB a few changes had to be introduced:

- modification of `Makefile.util.def` to include a reference to `grub-core/partmap/dfly.c`,
- modification of `grub-core/Makefile.core.def` to include a reference to `grub-core/partmap/dfly.c` and indicate that the name of the loadable GRUB module is `part_dfly` (this is the name usable from GRUB shell),
- addition of `grub-core/partmap/dfly.c` with `disklabel64` read support,
- addition of automatic tests of the new code and auxiliary files in `tests/`.

`grub-core/partmap/dfly.c` contains the definitions of `disklabel64` on-disk structures, a single callback function called by GRUB from outside the module and some initialization and finalization boilerplate code.

The first structure actually is *the disklabel*, i.e. a header containing some meta information about the disk, along with a table of entries describing the consecutive partitions:

```
/* Full entry is 200 bytes however we really care only
   about magic and number of partitions which are in first 16 bytes.
   Avoid using too much stack.  */
struct grub_partition_disklabel64
{
  grub_uint32_t   magic;
#define GRUB_DISKLABEL64_MAGIC        ((grub_uint32_t)0xc4464c59)
  grub_uint32_t   crc;
  grub_uint32_t   unused;
  grub_uint32_t   npartitions;
};
```

As can be seen in the above listing, only fields strictly necessary to enable read support of the disklabel are included in the structure definition. This is due to two principles – the limitations of the embedded environment that GRUB is running in, and the design decision that for safety and security reasons GRUB ought not to have write support for any on-disk data.

The second structure is a disklabel entry, i.e. a description of a single partition:

```
/* Full entry is 64 bytes however we really care only
   about offset and size which are in first 16 bytes.
   Avoid using too much stack.  */
#define GRUB_PARTITION_DISKLABEL64_ENTRY_SIZE 64
struct grub_partition_disklabel64_entry
{
  grub_uint64_t boffset;
  grub_uint64_t bsize;
};
```

Again, full read-write support would require full details of the structure to be present.

Signature of the callback function defined in `dfly.c` is as follows:

```
static grub_err_t
dfly_partition_map_iterate (grub_disk_t disk,
                            grub_partition_iterate_hook_t hook,
                            void *hook_data)
```

This function is called in a loop implemented in GRUB framework code.

In general, GRUB is able to handle nested partition tables, which are quite common on the personal computer (PC) x86 architecture. It's customary that a PC drive is partitioned using an MS-DOS partition table, which supports up to 4 primary partitions and significantly more logical partitions on an extended partition.

A bare disk would be referred to as `(hd0)` by GRUB. The second MS-DOS partition on a disk as `(hd0,msdos2)` (please note that disks are counted from 0 while partitions from 1), while the first DragonFly BSD subpartition of that MS-DOS partition as `(hd0,msdos2,dfly1)`.

In this light, it should be clearer how the partition recognition loop of GRUB (mentioned above) works. First, each partition type callback (e.g. `dfly_partition_map_iterate`, `grub_partition_msdos_iterate`, ...) is run on the raw device to find a partition table. Then, it is run on each partition found in the table (by using the `grub_partition_iterate_hook_t hook` parameter). Such a discovery procedure leads to at most two-level partition nesting, as in the `(hd0,msdos2,dfly1)` example.

The tests for automatic partition table and file system discovery, located in `tests/` directory of GRUB source tree, rely on GNU Parted. Being a partitioning utility Parted, unlike GRUB, supports full read and write access to a number of partition table and file system formats. Unfortunately, `disklabel64` is not one of them. In order to enable automatic tests of the `part_dfly` module it was necessary to supply a predefined disk image

containing the relevant disklabel. Two such images were prepared: one with an MS-DOS partition table and one with a DragonFly BSD `disklabel64`.

4.2 Booting DragonFly BSD with GRUB on x86

Conceptually, enabling GRUB to boot DragonFly BSD is relatively simple and involves the following steps:

- enabling GRUB to identify the kernel image as Multiboot compliant,
- adding an entry point to which GRUB will perform a jump when the kernel image is loaded and the environment is set up,
- once in the kernel, interpreting the information passed in by GRUB and performing any relevant setup to successfully start up the system.

However, things get difficult when we get to the details. The foremost issue is compatibility with the existing booting strategy. In other words, all changes done to the kernel must be backwards compatible with `dloader`, so as not to break the already existing boot path.

The following sections describe in detail how the steps listed above were implemented for the `pc32` variant of DragonFly BSD kernel, i.e. for the Intel x86 platform.

4.2.1 How does GRUB identify the kernel image?

The Multiboot specification (Okuji et al. 2006, sec. 3.1) states that besides the usual headers of the format used by an OS image, the image must contain a 32-bit aligned Multiboot header. This header must not exceed past 8KiB from the beginning of the image and generally should come as early as possible.

Except this requirement, there are really no constraints put onto the format of the kernel image file. Specifically, the kernel may be stored in ELF format, which is a widely accepted standard for object file storage. However, ELF requires the ELF header to be placed at the immediate beginning of a file (see TIS Committee and others 1995, sec. 1–2).

If not for the aforementioned flexibility of Multiboot, this ELF requirement would lead to a serious problem for booting DragonFly BSD with GRUB, because of the kernel being natively stored as an ELF file.

The most natural way of embedding the Multiboot header into the kernel is defining it in the assembly language. Given the heritage of DragonFly BSD it should not make one wonder that the kernel uses the AT&T assembler syntax[16].

---

[16]Ritchie and Thompson (1974) developed UNIX when working at AT&T Bell Laboratories.

The low level, architecture dependent parts of the kernel are in sys/platform/pc32 and sys/platform/pc64 subdirectories of the system source tree.

The entry point to the x86 kernel is defined in assembly in file sys/platform/pc32/i386/locore.s. It's format may seem a bit strange at first, since it's the AT&T syntax assembly mixed with C preprocessor directives. Deciphering some definitions and rules from sys/conf/kern.pre.mk leads to the general command such files are processed with:

```
gcc -x assembler-with-cpp -c sys/platform/pc32/i386/locore.s
```

The command handles generating an object file from such a C preprocessed assembly file. Thanks to this setup, the Multiboot header definition can use meaningfully named macros instead of obscure numbers defined by the specification:

```
/*
 * Multiboot definitions
 */
#define MULTIBOOT_HEADER_MAGIC      0x1BADB002
#define MULTIBOOT_BOOTLOADER_MAGIC  0x2BADB002
#define MULTIBOOT_PAGE_ALIGN        0x00000001
#define MULTIBOOT_MEMORY_INFO       0x00000002
#define MULTIBOOT_HEADER_FLAGS      MULTIBOOT_PAGE_ALIGN \
                                    | MULTIBOOT_MEMORY_INFO
#define MULTIBOOT_CMDLINE_MAX       0x1000
#define MI_CMDLINE_FLAG             (1 << 2)
#define MI_CMDLINE                  0x10

/* ... snip ... */

    .section .mbheader
    .align  4
multiboot_header:
    .long   MULTIBOOT_HEADER_MAGIC
    .long   MULTIBOOT_HEADER_FLAGS
    .long   -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)
```

The header itself consists of 3 fields, each 4 bytes wide:

- the Multiboot header magic number: 0x1BADB002,
- flags which state what the kernel expects from the bootloader,
- a checksum which when added to the other header fields must have a 32-bit unsigned sum of zero.

These values are sufficient for GRUB to recognize the kernel image as Multiboot compliant. Please refer to Okuji et al. (2006) for more details.

However, it's not sufficient just to place the header in locore.s. Please note that the header is declared in its own .mbheader section of the object file.

This is necessary, but not enough, to comply with the requirement of placing it in the first 8KiB of the kernel image.

4.2.2 Modifying the linker script

Executable and Linking Format Specification (TIS Committee and others 1995) describes the format of an object file – e.g. the DragonFly BSD kernel executable. While ELF and its specification is a product of the post-UNIX age of computing history, the process of composing programs from reusable parts in its rawest form, or *linking*, is probably as old as computing itself.

What is the responsibility of a linker? Levine (1999) in *Linkers & Loaders* gives an answer that a linker assigns numeric locations to symbols, determines the sizes and location of the segments in the output address space, and, what is crucial in our case, figures out where everything goes in the output file. Chamberlain and Taylor (2003) in *Using ld, the GNU linker* point at a different aspect – that a linker maps sections of the input files into the output file and that most linker scripts do nothing more than this. The same authors also inform that GNU `ld` accepts Linker Command Language files written in a superset of AT&T's Link Editor Command Language syntax, which ought to provide explicit and total control over the linking process.

In the light of this stated *total control*, which decides *where everything goes in the output file*, one might think that making the linker place the Multiboot header at the beginning of the output file is a simple matter. The experience gained from this project is exactly the opposite.

The behaviour of GNU `ld`, the linker used by DragonFly BSD, may either be driven by an explicitly specified linker script or by one embedded in the linker executable itself.

In case of userspace applications, the programmer doesn't have to worry about specifying the linker script, as `ld` will in almost all cases *just do the right thing*. Unfortunately, that's not the case for kernel development. There are several reasons for using a custom linker script for the kernel.

One of them is portability among different architectures and compiler vendors, which is needed for bootstrapping. In order to be able to build the kernel on a different OS, the linker script must be compatible with the platform's compiler which might emit different output than the native compiler of DragonFly BSD. In other words, before we have *any* running DragonFly BSD system, we can't build DragonFly BSD on DragonFly BSD, hence the need of supporting different architectures.

Another reason is that by using the linker script it's possible to define symbols in the program namespace, whose values couldn't be otherwise determined in the code. Examples of such symbols are `edata` or `end` meaning, respectively,

the end of the `.data` section and of the kernel binary. In a userspace program there's no (or little) use for such symbols, but the kernel uses them for calculating the offset and size of the `.bss`[17] section which it must initialize itself.

The language accepted by `ld` has constructs for setting symbol/section virtual addresses, overriding their linear (or load) addresses, specifying regions of accessible memory and their access modes and assigning input sections to output sections placed in particular program segments. None of these constructs could be used to precisely specify the place of the Multiboot header in the output kernel binary.

Below is a description of the attempts to enforce the correct offset of the Multiboot header in the kernel image:

1. Placing the `.mbheader` section before the `.text` section:

   ```
   diff --git a/sys/platform/pc32/conf/ldscript.i386 \
             b/sys/platform/pc32/conf/ldscript.i386
   index a0db817..1068ba5 100644
   --- a/sys/platform/pc32/conf/ldscript.i386
   +++ b/sys/platform/pc32/conf/ldscript.i386
   @@ -21,6 +21,7 @@ SECTIONS
      kernmxps = CONSTANT (MAXPAGESIZE);
      kernpage = CONSTANT (COMMONPAGESIZE);
      . = kernbase + kernphys + SIZEOF_HEADERS;
   +  .mbheader       : { *(.mbheader) }
      .interp         : { *(.interp) } :text :interp
      .note.gnu.build-id : { *(.note.gnu.build-id) } :text
      .hash           : { *(.hash) }
   ```

   On the contrary to what one might expect, this had the effect of placing the `.mbheader` input section in the `.data` output section, which is placed *after* the `.text` output section. That's definitely far from the initial 8KiB of the kernel binary.

2. Inserting the `.mbheader` section at the beginning of the `.text` section. This succeeded in the sense that the header was in fact placed before all other program code. Alas, the `.text` section itself begins far in the file, after the program headers, the `.interp` section and a number of sections containing relocation information.

3. Analysis of `readelf` and `objdump` output showed that symbol offsets from the beginning of the kernel image were equal to the virtual addresses decreased by a constant value. In fact, the binary was linked to use the same values for virtual and load addresses. The constant

---

[17] `.bss` is a traditional name of a section in the object file containing uninitialized data; since this data has no explicit value, there's no point in assigning storage space for this section in an object file. The name stands for *block started by symbol*.

offset was equal to the value of symbol `kernbase`, which specifies a page aligned base address at which the kernel is loaded.

This led to an attempt at forcing the position in the output binary by modifying the load address of the whole `text` output segment through adjusting it in the `PHDRS` section of the linker script. This segment contained all consecutive sections up to the `.data` section, i.e. `.interp`, sections with relocation and symbol information and finally the `.text` section. An arbitrary address close to the beginning of the file was picked, but to no avail. As it became obvious later (when figuring out how to make GRUB understand what address to load the kernel at) this couldn't have worked.

4. Yet another unsuccessful approach was putting the `.mbheader` section into the `headers` output segment.

5. Introducing a completely new program header (a.k.a. segment) was also tried with no success. This is probably explained by the message of commit e19c755[18] from the DragonFly BSD repository, from which we can infer that `dloader` expects a kernel with exactly 2 loadable program segments. An image with more can be generated easily, but it won't be bootable by `dloader`.

6. Finally, the trial and error process led to inserting the `.mbheader` section at the end of the `.interp` section. Section `.interp` contains a path to the program interpreter, i.e. a program which is run in place of the loaded binary and prepares it for execution[19]. In case of a kernel, the path to the interpreter is just a stub (`/red/herring` to be exact). The `.interp` section is the first in the kernel image. Placing the `.mbheader` section after the null-terminated string in the `.interp` section causes no issues with accessing the interpreter path (if it is ever necessary), while still leads to placement of the Multiboot header in the first 8KiB of the kernel binary. This approach is illustrated below:

```
diff --git a/sys/platform/pc32/conf/ldscript.i386 \
          b/sys/platform/pc32/conf/ldscript.i386
index dc1242e..24081c9 100644
```

---

[18]The verbatim message is: *The gold linker changed its ELF program header handling defaults for version 2.22, and this resulted in an extra LOAD segment reserved only for the program headers. The DragonFly loader wasn't expecting that and instantly rebooted when trying to load a gold kernel.*

[19]On x86-64 Linux this is typically `/lib64/ld-linux-x86-64.so.2` while on DragonFly BSD `/usr/libexec/ld-elf.so.2`. The majority of software nowadays is dynamically linked. This means that an executable doesn't contain all the code the program needs to run successfully and relies on shared libraries commonly available on the target system. In case of userspace programs, the interpreter handles symbol relocations and lazy loads these dynamically linked shared libraries finalizing the linking of the program during its execution.

```
--- a/sys/platform/pc32/conf/ldscript.i386
+++ b/sys/platform/pc32/conf/ldscript.i386
@@ -21,8 +21,8 @@ SECTIONS
    kernmxps = CONSTANT (MAXPAGESIZE);
    kernpage = CONSTANT (COMMONPAGESIZE);
    . = kernbase + kernphys + SIZEOF_HEADERS;
-   .interp         : { *(.interp) } :text :interp
+   .interp         : { *(.interp)
+                         *(.mbheader) } :text :interp
    .note.gnu.build-id : { *(.note.gnu.build-id) } :text
    .hash           : { *(.hash) }
    .gnu.hash       : { *(.gnu.hash) }
```

The approach hasn't caused any issues with the resulting binaries so far, but can't be considered *the proper way* of embedding the Multiboot header.

4.2.3 Booting the 32 bit kernel

Once the kernel is identifiable by GRUB what's left is making it bootable. This requires making GRUB load the image to a reasonable memory address, using a valid entry address and modifying the kernel entry point to handle entry from GRUB. All of that must be done in a way which maintains bootability by `dloader`.

The kernel, once booted by GRUB, instead of expecting the `struct bootinfo` structure must be able to interpret the structure passed from GRUB, which the Multiboot specification describes in detail. This is required at least for setting the root filesystem, but adjusting the kernel environment may be used for multiple other purposes.

No matter what bootloader booted the kernel, the entry will happen at the same address. However, just after entry from the bootloader the code must branch taking into account the points stated above. The branching 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.

4.2.4 Loading the image: `dloader`

By default, the DragonFly BSD kernel image is linked with virtual and physical addresses being the same. Moreover, the kernel has only two loadable program segments (marked `LOAD` in the listing below). This information can be read from the kernel image using `readelf`:

```
$ readelf -l /boot/kernel.generic/kernel
```

```
Elf file type is EXEC (Executable file)
Entry point 0xc016c450
There are 5 program headers, starting at offset 52

Program Headers:
  Type    Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR    0x000034 0xc0100034 0xc0100034 0x000a0 0x000a0 R   0x4
  INTERP  0x0000d4 0xc01000d4 0xc01000d4 0x0000d 0x0000d R   0x1
      [Requesting program interpreter: /red/herring]
  LOAD    0x000000 0xc0100000 0xc0100000 0x85a6ec 0x85a6ec R E 0x1000
  LOAD    0x85a6ec 0xc095b6ec 0xc095b6ec 0x9d313 0x50b454 RW  0x1000
  DYNAMIC 0x85a6ec 0xc095b6ec 0xc095b6ec 0x00068 0x00068 RW  0x4
...
```

The `-l` flag tells `readelf` to list just the program headers (a.k.a. segments).

`dloader` can load such kernel just fine, thanks to its algorithm for reading ELF images. First, it ignores the physical addresses embedded in the ELF image. Second, it calculates the load address based on the entry address stored in the image.

The entry address is read from the ELF header[20] and then passed to a function aware of the architecture word size that will read the ELF image. The signature of this function is automatically generated using the `__elfN(loadimage)`[21] macro, so that the same definition, written with appropriate care, can be reused for both ELF32 and ELF64 formats. For ELF32 a call to this function expands to `elf32_loadimage(fp, &ef, dest)`.[22] The load address is adjusted depending on the architecture[23]. For ELF32 and an initial entry address equal to `0xc016c450`, the offset equals `0xffffffff40000000`. An example program header virtual address of `0xc0100000` offset by `0xffffffff40000000` gives `0x100000`, i.e. 1MiB – a convenient low physical address to load the kernel at.

The insight from the above analysis is that the load addresses the kernel is linked with aren't important from the perspective of `dloader`. They can be freely adjusted in any way that is convenient.

4.2.5 Loading the image: GRUB

The ending remark of the previous section is very fortunate, as GRUB is much stricter than `dloader` in following the ELF specification. It relies solely on the segment load addresses for loading an ELF kernel. Thanks to `dloader`'s ambivalence as to the values of these addresses, we can safely adjust them in any way required by GRUB. One thing to keep in mind,

---

[20]See DragonFly BSD: `sys/boot/common/load_elf.c:174`
[21]See DragonFly BSD: `sys/boot/common/load_elf.c:258`
[22]See DragonFly BSD: `sys/boot/common/load_elf.c:233`
[23]See DragonFly BSD: `sys/boot/common/load_elf.c:289`

though, is not to introduce extra program headers, as `dloader` won't be able to handle it.

The only thing required to adjust the load addresses is overriding the load address of the first output section in the linker script - the following sections will be just laid out linearly after the adjusted one. This first section is the already mentioned `.interp`. The introduced change then is as follows:

```
    kernmxps = CONSTANT (MAXPAGESIZE);
    kernpage = CONSTANT (COMMONPAGESIZE);
    . = kernbase + kernphys + SIZEOF_HEADERS;
-   .interp         : { *(.interp)
-                       *(.mbheader) } :text :interp
+   .interp         : AT (ADDR(.interp) - kernbase)
+   {
+     *(.interp)
+     *(.mbheader)
+   } :text :interp
    .note.gnu.build-id : { *(.note.gnu.build-id) } :text
    .hash           : { *(.hash) }
    .gnu.hash       : { *(.gnu.hash) }
```

The `AT (<expression>)` keyword sets the load address of an output section. As already mentioned, following sections will be laid out linearly after this one, so their load addresses will be modified accordingly. The `ADDR (<section>)` keyword returns the virtual address of `<section>`.

Inspecting a kernel built with the adjusted linker script shows that the segment load addresses are in fact offset from the virtual addresses by a value of `0xc0000000`, i.e. the `kernbase` location.

```
$ readelf -l /boot/kernel/kernel

Elf file type is EXEC (Executable file)
Entry point 0xc013b040
There are 5 program headers, starting at offset 52

Program Headers:
  Type    Offset    VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR    0x000034  0xc0100034 0x00100034 0x000a0 0x000a0 R   0x4
  INTERP  0x0000d4  0xc01000d4 0x001000d4 0x00019 0x00019 R   0x4
      [Requesting program interpreter: /red/herring]
  LOAD    0x000000  0xc0100000 0x00100000 0x310134 0x310134 R E 0x1000
  LOAD    0x310134  0xc0411134 0x00411134 0x2b875 0x4662ac RW  0x1000
  DYNAMIC 0x310134  0xc0411134 0x00411134 0x00068 0x00068 RW  0x4
...
```

Also, it is worth noting how GRUB treats the entry address present in the ELF headers of the kernel image:

```
$ readelf -h /boot/kernel/kernel
ELF Header:
  ...
```

```
    Entry point address:              0xc013b040
    ...
```

As can be seen above, the entry point (`0xc013b040`) is specified as a virtual address positioned high in the memory. Fortunately, GRUB is capable of adjusting this value by the difference between the virtual and physical addresses of the segment that the entry point is defined in (i.e. by `kernbase`)[24].

This is sufficient for GRUB to be able to properly load the kernel image and jump to the entry point address. The next step is to make the entry point expect entry from GRUB, i.e. to adjust the low level assembly routine controlling the kernel just after it's jumped to.

4.2.6 Adjusting the kernel entry point

Output of `readelf` in the previous section tells us that the entry address of the kernel is `0xc013b040` (more or less, depending on the exact version of the code). The top of the linker script, on the other hand, tells us what code actually resides at that address:

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(btext)
SEARCH_DIR(/usr/lib);
...
```

`btext` is an assembly procedure defined in `sys/platform/pc32/i386/locore.s`, the same file where the definition of the Multiboot header was placed.

There's no use quoting the verbatim contents of the file, but I'll describe the changes introduced in order to perform an entry from GRUB.

Just after entering the kernel we check whether the bootloader that booted us is Multiboot compliant. This is done by comparing the value stored in `eax` register with `MULTIBOOT_BOOTLOADER_MAGIC` macro, whose value is a magic number defined in the Multiboot specification. The comparison is illustrated in the following listing:

```
@@ -208,6 +232,17 @@ NON_GPROF_ENTRY(btext)
 /* Tell the bios to warmboot next time */
        movw    $0x1234,0x472

+/* Are we booted by a Multiboot compliant bootloader? */
+       cmpl    $MULTIBOOT_BOOTLOADER_MAGIC,%eax
+       jne     1f
+       /* We won't be using the traditional bootinfo,
```

---

[24]See GRUB: `grub-core/loader/multiboot_elfxx.c:131`

```
+            * so mark the relevant fields as undefined. */
+        movl    $0, R(bootinfo+BI_ESYMTAB)
+        movl    $0, R(bootinfo+BI_KERNEND)
+        movl    %ebx, R(multiboot_info)
+        jmp     2f
+1:
```

Unless the kernel is booted by a Multiboot compliant bootloader we skip the newly added code block and continue booting along the old path (i.e. `jne 1f` – *if not equal jump forward to local label 1*).

If it is Multiboot compliant, we mark the relevant fields of the `bootinfo` global variable as unused by assigning to them the value of zero. There are two interesting constructs used in these assignments. The first is a C structure field access done in the assembly language:

```
bootinfo+BI_ESYMTAB
```

`bootinfo` is, of course, the address of the structure itself, while `bi_esymtab` is one of the structure fields[25]. `BI_ESYMTAB` is a macro defined in `assym.s`, equal to the offset of field `bi_esymtab` in structure `struct bootinfo`. `assym.s` is a file automatically generated by `genassym.sh`. `genassym.sh(8)` FreeBSD 9.2 manpage says that the generated file is used by kernel sources written in assembler to gain access to information (e.g. structure offsets and sizes) normally only known to the C compiler. The second interesting thing in the above diff is the `R` macro:

```
movl    $0, R(bootinfo+BI_ESYMTAB)
```

whose definition is:

```
#define R(foo) ((foo)-KERNBASE)
```

The above definition tells us that each memory access via a virtual address should be done to a location specified by that virtual address decreased by the value of `KERNBASE`. It makes perfect sense when we realize that:

- the kernel is linked to run at a high virtual address (`0xc0000000`),
- the kernel is loaded at a low physical address (`0x100000`).

In fact, most of code in `locore.s` is running from a low memory location because that's where the bootloader puts the kernel. The MMU (memory management unit) is not setup yet to perform translation from low physical to high virtual addresses. Appropriately setting up the page tables (which drive the MMU translation) is one of the tasks performed in `locore.s`. Since the MMU isn't ready yet, some other mechanism of translation must be used to access the memory at the proper physical location – that's the purpose

---

[25]See DragonFly BSD: `sys/platform/pc32/include/bootinfo.h:48`

of the `R` macro. Code intended to run after relocation[26] does not use the macro anymore.

We're done with `struct bootinfo` once its fields are marked as unused. The next line:

```
movl    %ebx, R(multiboot_info)
```

saves the pointer to the Multiboot info structure passed in by GRUB. Later, the kernel command line is extracted from the structure by the `recover_multiboot_info` procedure and stored in a preallocated static page of kernel memory. Copying the whole command line buffer is done because of two reasons:

- the buffer is allocated by GRUB and its location is arbitrary; sooner or later the operating system will reuse the memory where it's stored, so it's safer to copy the contents to kernel memory,

- since the location of the buffer is not known beforehand (i.e. can't be foreseen at kernel compile time) it's not obvious how to access it after relocation.

Once the changes of the entry point described in this section were carried out the system was able to boot. The boot process at this moment was interactive, i.e. since the `struct bootinfo` fields were zeroed, the kernel was unable to find a root device and mount the root filesystem at `/`. This led to a prompt being displayed at boot time, asking for the device to be mounted as `/`. Except for that, the system turned out to be fully functional.

4.2.7 Mounting the root file system

`dloader` is capable of preparing the kernel environment before booting the kernel. In fact, it has a Forth interpreter built in, what makes it capable of performing some complicated tasks. Most of the time, though, `dloader` is simply used for parameterizing the functioning of some kernel subsystem, device driver or network stack. Setup of the kernel might also involve basic things, like choosing the root device. However, the kernel environment is essentially just a simple key-value storage space, where keys and values are zero-terminated strings whose meaning is left for interpretation to particular kernel subsystems.

---

[26]Relocation is the switch from executing code at physical addresses to executing at virtual addresses. It's done by setting up a virtual to physical memory mapping through appropriately populating the pagle tables. Once the MMU uses the proper page tables, a "fake return" virtual address is pushed onto the stack and a "return" to that address is performed using the `ret` instruction. See `sys/platform/pc32/i386/locore.s:335` for an example.

GRUB is neither aware nor capable of adjusting this kind of BSD specific kernel environment. However, the Multiboot specification defines that the kernel may be passed a command line. It's simple to simulate a key-value kernel environment with a command line formatted according to the below scheme:

```
kernel key1=val1 key2=val2 ...
```

Interpreting such a command line, however, requires the kernel to have some logic aware of the convention. This logic could populate the kernel environment just as `dloader` would do.

This code, though still coupled to the particulars of the bootloader which booted the kernel, is definitely out of scope of `locore.s`. Moreover, it would be unwise to write it in assembly, when C is easily available just a little later in the boot process.

As described in the previous section, the contents of the command line passed by GRUB are stored safely into kernel memory in `locore.s`. However, the command line contents aren't interpreted until later in the boot process. Choosing the exact moment is a bit tricky. On the one hand, the support mechanism of the dynamic kernel environment must already be setup, on the other hand, the interpretation must be performed as early as possible, since other initializing subsystems might depend on the information passed on the command line.

Fortunately, in the DragonFly BSD kernel there already exists a subsystem initialization mechanism with sophisticated priority management. Below is a glimpse of the priority setting definitions:

```c
// sys/sys/kernel.h:90
/*
 * Enumerated types for known system startup interfaces.
 *
 * Startup occurs in ascending numeric order; the list entries are
 * sorted prior to attempting startup to guarantee order.
 * ...
 */
enum sysinit_sub_id {
    ...
    SI_BOOT1_POST       = 0x1800000,    /* post boot1 inits */
    ...
    SI_SUB_ROOT_CONF    = 0xb000000,    /* Find root devices */
    ...
};
```

As seen in the example listing above, the priorities are defined in wide intervals in order to leave room for introducing new subsystems in between the existing ones. Kernel subsystems, defined in multiple files around the source code tree, define the points at which they expect to be initialized.

Using a declaration, instead of direct control flow passing from one subsystem to the next, guarantees loose coupling between the components, but still maintains the proper relative order of their initialization. Subsystems loaded dynamically as kernel modules are also taken into account when determining the subsystem initialization order.

Given the above-described initialization mechanism, the following listing presents how the code interpreting the command line passed by GRUB[27] declares its required moment of initialization:

```
SYSINIT(multiboot_setup_kenv, SI_SUB_ROOT_CONF, SI_ORDER_MIDDLE,
        multiboot_setup_kenv, NULL)
```

The `SYSINIT` macro is used to define the subsystem name (in the example above: `multiboot_setup_kenv`), initialization order (`SI_SUB_ROOT_CONF` and `SI_ORDER_MIDDLE`) and initialization function (again `multiboot_setup_kenv`). Please note that this initialization step must be done between `SI_BOOT1_POST` and `SI_SUB_MOUNT_ROOT`.

At `SI_BOOT1_POST` the static kernel environment (inherited from the boot-loader in case of `dloader`) is copied into the dynamic kernel environment, accessible via a kernel API. Before this operation, the dynamic kernel environment is not initialized yet and not accessible programmatically.

At `SI_SUB_MOUNT_ROOT` the kernel mounts the root device and expects any hints as to what device to use to already be present in the dynamic environment. `SI_SUB_ROOT_CONF` seems to be a reasonable step at which to perform the GRUB command line interpretation.

The initialization function uses `strsep()` implemented in the kernel C library (`sys/libkern.h`) to parse the command line and set a `kenv` for each `key=val` entry on the command line. Malformed entries (i.e. not containing an equal sign `=`) are ignored.

This way, given GRUB passes a valid value for `vfs.root.mountfrom` (e.g. `vfs.root.mountfrom=ufs:/dev/ad0s1a`), the code responsible for mounting the root file system in `sys/kern/vfs_conf.c` is aware of the device to be used for that purpose. This makes the boot process using GRUB fully automatic and seamless, from the GRUB prompt straight to the login shell.

---

[27]See DragonFly BSD: `sys/platform/pc32/i386/autoconf.c:500`

5 Booting DragonFly BSD with GRUB on x86-64

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

First, when the specification was defined in 1995, the x86-64 was still to be unknown for the next 5 years.[28] Second, 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 is feasible: after all, the kernel manipulates the memory management settings as part of usual process context switching.

An alternative approach, however, would be to allow the processor to enter *real long mode* in which the addresses would be 64 bit wide and physical. That way the bootloader would be freed from setting up memory management on behalf of the OS. The kernel would be responsible for setting up memory management from the ground up in the most sensible way. This approach would also require the kernel to use the relocation macro technique (see the `R` macro description in section *4.2.6. Adjusting the kernel entry point*) in order to make the code linked for high logical addresses function properly before setting up the page table hierarchy and relocating.

Since GRUB doesn't enter long mode and can't make use of 64 bit addressing, no matter whether the architecture allows for that with or without paging, the only way to boot a 64 bit operating system by GRUB is by embedding a piece of 32 bit code into the 64 bit kernel image.

5.1 Embedding a 32 bit entry point into a 64 bit kernel

Given the aforementioned limitations of GRUB and the CPU the cleanest possible way of loading a 64 bit kernel is by embedding some 32 bit code into the 64 bit kernel. This limits the choice of kernel load location to the 32 bit addressable space.

In case of DragonFly BSD this 32 bit code would realize roughly the same functionality as `sys/boot/pc32/libi386/x86_64_tramp.S` which is part of the native `dloader`:

---

[28] According to Wikipedia: AMD64 was *announced in 1999 with a full specification in August 2000.*
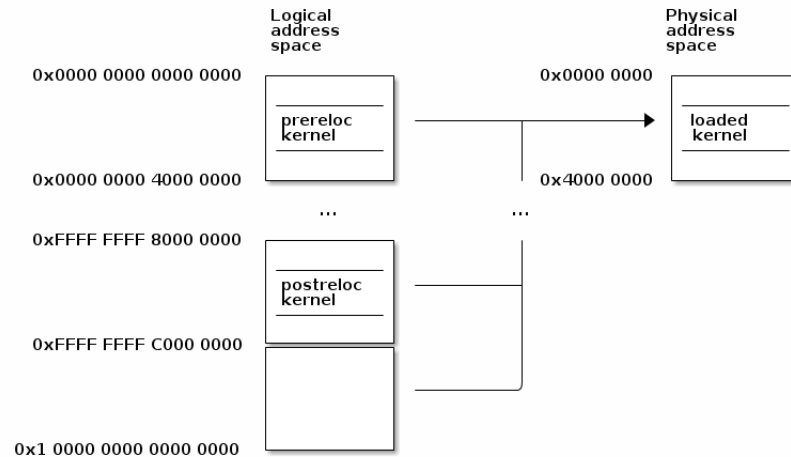
Figure 1: Each 1GiB block of logical memory is mapped to the first 1GiB block of physical memory (64 bit address space). Before relocation the code executes from *prereloc kernel*, after relocation from *postreloc kernel*.

- It would set a preliminary memory mapping. Each block of logical memory (without loss of generality 1GiB in size) would point to the first block of physical memory (see figure 1). That way the kernel could be loaded to low physical memory but still be linked to use high logical addresses. Both the prerelocation and postrelocation code would run without requiring tricks such as the `R` prerelocation macro.

- It would enable paging, enter long mode and jump to the 64 bit entry point of the kernel.

One problem with this approach is that there would be two distinct entry points to the kernel: one for `dloader`, which is capable of booting the kernel straight into long mode, and another for GRUB with protected mode only. That would require using the *a.out kludge*, i.e. apart from embedding the Multiboot header into the kernel image, embedding some extra fields which would override the meta information already stored in the ELF headers. As the names suggests, the technique is hardly elegant, but might be unavoidable in case of two different entry points. In other words, the *a.out kludge* is another way, apart from reading an ELF header, for GRUB to read metadata from a loaded file.

Implementation of this approach was tried during the project, but due to the involved complexity and limited amount of time, was not finalized. This path is open for pursuing in the future.

6 Tools of Kernel Developer's Trade

So, you want to write a kernel module. You know C, you've written a few normal programs to run as processes, and now you want to get to where the real action is, to where a single wild pointer can wipe out your file system and a core dump means a reboot.

– Salzman, Burian, and Pomerantz (2001), The Linux Kernel Module Programming Guide

Well... not really. The days when kernel debugging was done using a serial printer plugged into a spare headless PC and a core dump meant a reboot are now gone, thanks to virtualization.

The tools I used for virtualization are VirtualBox and QEMU with Linux KVM (Kernel-based Virtual Machine). Although the work was started using a VirtualBox virtual machine, I had to switch to QEMU/KVM in the middle of the project because VirtualBox does not support debugging with GNU gdb. Moreover, the 32 bit and 64 bit systems had to use separate virtual machines, so all the work with virtual machine management and system installation and setup had to be done twice.

As already mentioned, GNU gdb was used for GRUB and kernel debugging. Since the kernel image is linked to use high logical addresses and before relocation (just after being loaded by the bootloader) the kernel operates from a physical address, most of the debugging had to be done by setting breakpoints at manually calculated physical addresses instead of using symbol names. The process was arduous, error prone and consumed a lot of time.

The main piece of software I worked on during the project was DragonFly BSD. Hsu (2004) introduces the system well and points out some of the differences from its parent, FreeBSD. Just to name a few features which stand out:

- DragonFly BSD tries to scale linearly on modern multi-core CPUs by multiplication of certain subsystems and data structures (e.g. the scheduler, memory allocator, ...) onto all cores and using non-blocking algorithms where possible, which allows for mostly asynchronous operation of all cores,

- HAMMER (and upcoming HAMMER2) filesystem, whose feature list is too long to quote here, but the most prominent are the built-in historical access and master-to-multiple-slaves streaming,

- Swapcache, which is an extension to the well-known swap partition mechanism; the feature allows an SSD swap partition to cache any

28

filesystem operation(s) in order to speed up file systems stored on non-SSD devices.

All the modifications to the kernel image and verification of changes in the linker script had to be checked by inspecting the resulting image file. The utilities for these tasks are `readelf` and `objdump`, which come from the GNU binutils package.

Parts of the code written during this project are 64 bit assembly. The book by Seyfarth (2011) was a great help in putting these down.

The *6.828: Operating System Engineering* course material from MIT (Cates et al. 2012), as well as the source code (along with commentary) of Xv6, a simple, educational operating system (Cox, Kaashoek, and Morris), were also of considerable help.

7 Related Work

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

Coreboot[29] is a BIOS firmware replacement. It is based on the concept of *payloads* (standalone ELF executables) that are loaded in order to offer a specific set of functionality required by the software that is to be 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). This contrasts with GRUB, which is installable on a personal computer by a power-user.

UEFI[30] (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 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, but it still maintains backward compatibility with legacy systems which understand only MBR. The maximum number of partitions on a GPT partitioned volume is 128, with the maximum size of a partition (and the whole disk) being 8ZiB (2^{70} bytes).

In essence, UEFI is similar to the Multiboot Specification as it addresses 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 with 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.

---

[29]http://www.coreboot.org/
[30]http://www.uefi.org/home/

Das U-Boot[31] is another open source bootloader, though targeted mainly at embedded devices. The range of architectures it supports is much wider than in the case of GRUB. Thanks to its architecture, Das U-Boot is capable of supporting a multitude of hardware devices from the same mainline code base - this feature was developed in order to avoid proliferation of forks of the code just for the sake of supporting more and more devices. Das U-Boot, like GRUB, is able to run as a Coreboot payload. Yaghmour (2003) states that *though there are quite a few other bootloaders,* Das U-Boot, *the universal bootloader, is arguably the richest, most flexible, and most actively developed open source bootloader available.* However, it is not the default bootloader of any of the big Unix-like operating systems - neither FOSS nor commercial.

Last but not least, I feel obliged to mention the article by Vidal (2007), entitled *Making NetBSD Multiboot-Compatible*, which was one of the main inspirations for this project. While the idea described in that article is generally the same as pursued in my project, none of the implementation could be reused due to differences between DragonFly BSD and NetBSD at the source code level.

---

[31]http://www.denx.de/wiki/U-Boot/

8 Conclusions and Future Work

Two significant pieces of code were implemented out as part of this project:

- a GRUB module for reading `disklabel64` partition tables, which is merged into GRUB mainline,

- support for booting the x86 variant of DragonFly BSD by GRUB, which does not interfere with the existing boot path.

Implementation of the Multiboot Specification in DragonFly BSD, described herein, was done for the x86 platform. Based on the finding by Tigeot (2013) that *80.15% of downloaded packages are for the amd64/x86_64 architecture*, it was decided that the x86 platform will be dropped in the future, though the exact release in which this will happen has not been decided yet.

No special care was taken to ensure proper loading and functioning of loadable kernel modules (also known as `.ko` files - *kernel objects*). It is possible that the mechanism works transparently due to the changes already introduced to the kernel. This issue needs further attention and, possibly, development effort.

Implementing the GRUB – DragonFly BSD interoperability for x86-64 is an open field for future work. It can be done either according to the scenario described in one of the previous sections or by extending Multiboot and GRUB to allow for loading ELF64 kernels natively.

The current implementation for x86 is capable of loading the DragonFly BSD kernel only from a UFS filesystem, but the hallmark feature of DragonFly BSD is the HAMMER filesystem. Enabling GRUB to read HAMMER and boot from a HAMMER volume would be a major step forward. HAMMER2 design is described by Dillon (2011), while Lorch (2009) and Oppegaard (2009) wrote more on the first version of the filesystem.

Interoperability between GRUB and UEFI firmware is an interesting topic due to the *Secure Boot* protocol defined by UEFI 2.2. The protocol requires UEFI firmware to load and pass control only to software which has an acceptable, cryptographically-strong digital signature. The protocol was criticized as a possible tool for vendor lock in, as on machines with UEFI firmware it could make it impossible to install open source operating systems side by side with proprietary ones. Different approaches to signing a kernel and a system bootloader are possible, but in case of FOSS operating systems all include the use of a *shim* - a small bootloader signed with a valid certificate, which then loads another bootloader – usually non-signed GRUB. It's an open question whether to sign kernel images and modules of an operating system: signing improves security of the boot process, but makes it impossible to recompile the kernel by the end user. There are examples of both policies – signing and non-signing – in the Linux world: Fedora Linux kernels and

modules are cryptographically signed, while kernels shipped with Ubuntu are not. Devising a rational policy for using UEFI, secure boot, GRUB and DragonFly BSD is an interesting field of endeavour.

9 References

Cates, Josh, Russ Cox, Bryan Ford, Emil Sit, and Max Krohn. 2012. Course no. 6.828: Operating System Engineering. http://pdos.csail.mit.edu/6.828/2012/index.html.

Chamberlain, Steve, and Ian Lance Taylor. 2003. Using ld: the GNU Linker. Boston, MA, USA: Free Software Foundation, Inc.

Cox, Russ, MF Kaashoek, and Robert Morris. Xv6, a simple Unix-like teaching operating system. Technical report. http://pdos.csail.mit.edu/6.828/2012/xv6.html.

Dillon, Matthew. 2011. HAMMER2 Preliminary Design Document. http://apollo.backplane.com/DFlyMisc/hammer2.txt.

GNU Hurd contributors. 2010. The GNU/Hurd User's Guide. http://www.gnu.org/software/hurd/users-guide/using_gnuhurd.html#Bootstrap.

Hsu, Jeffrey M. 2004. The DragonFlyBSD Operating System.

Levine, John R. 1999. *Linkers & Loaders*. San Francisco: Morgan Kaufmann Publishers.

Lorch, Daniel. 2009. Porting the HAMMER File System to Linux.

Okuji, Yoshinori K., Bryan Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. 2006. The Multiboot Specification Version 0.6. *Free Software Foundation*.

Oppegaard, Martin Christoffer Aasen. 2009. Evaluation of Performance and Space Utilisation when using Snapshots in the ZFS and HAMMER File Systems.

Ritchie, Dennis M., and Ken Thompson. 1974. The UNIX time-sharing system. *Communications of the ACM* 17, no. 7: 365–75.

Salzman, Peter Jay, Michael Burian, and Ori Pomerantz. 2001. The Linux Kernel Module Programming Guide. http://tldp.org/LDP/lkmpg/2.6/html/x40.html.

Seyfarth, Ray. 2011. Introduction to 64 Bit Intel Assembly Language Programming: Getting the most out of your Computer.

Steinberg, Udo, and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. *Proceedings of the 5th European Conference on Computer Systems*. ACM.

The FreeBSD Documentation Project. 2014. FreeBSD Architecture Handbook. http://www.freebsd.org/doc/en/books/arch-handbook/.

Tigeot, Francois. 2013. Francois Tigeot to the DragonFly BSD Operating System mailing list. http://article.gmane.org/gmane.os.dragonfly-bsd.user/958.

TIS Committee, and others. 1995. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee*.

Vidal, Julio M. Merino. 2007. Making NetBSD Multiboot-Compatible. http://www.onlamp.com/pub/a/bsd/2007/03/01/inside-multiboot.html.

Wikipedia. 2014a. LILO (boot loader) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=LILO_(boot_loader)&oldid=604713411.

---. 2014b. Loadlin — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Loadlin&oldid=607850985.

Yaghmour, Karim. 2003. *Building Embedded Linux Systems*. O'Reilly Japan.