Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Dominik Táborský

HelenOS Installer

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Martin Děcký
Study programme:	Computer Science
Specialization:	General Computer Science

Prague 2013

I wish to thank to my supervisor, my family and anyone who supported me.

Special thanks belongs to my father, who has left us just too soon. In memory of Daniel Táborský.

Název práce: HelenOS instalátor

Autor: Dominik Táborský

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Martin Děcký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Schopnost sebe sama nainstalovat na trvalé úložiště je jedna z věcí definujících použitelnost systému. V této práci se podíváme na naše možnosti jak toho dosáhnout v případě operačního systému HelenOS. Budeme se zabývat tím, jaké máme volby, jaké jsou jejich výhody a nevýhody a konečně jaké jsou jejich implementační detaily. Součástí práce je též prototypová implementace kritických částí, která je také popsána. Rozhodnutí při návrhu implementace jsou taktéž diskutována.

Klíčová slova: HelenOS, MBR, GPT, GRUB

Title: HelenOS installer

Author: Dominik Táborský

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Department of Distributed and Dependable Systems

Abstract: The capability to install itself on a permanent storage is one of the things that define usability of an operating system. In this thesis we look at our possibilities of achieving that within the HelenOS operating system. We discuss what options there are, what are their advantages and disadvantages and finally their implementation details. Prototype implementation has been written of those critical parts, which is also described. Implementation design decisions and their pros and cons are discussed as well.

Keywords: HelenOS, MBR, GPT, GRUB

Contents

Introduction

HelenOS [1] has been here for about ten years. The development is slow but interest for contributions among developers increases. Before writing this thesis HelenOS still had not had any tools for installation on hard disks - no partition editor, no boot loader, no installer and mainly not many useful filesystems. Since then large progress has been made. Several students have applied for tasks including writing support for ext4 [2], ext2 [3], MinixFS and VFAT.

For self-hosted installation few things are needed:

1. Partition the disk.

2. Create filesystem(s).

3. Install a bootloader.

4. Install the rest of the system.

Filesystems have already been covered. Installing the system itself is, fortunately, simple - it only requires file copying. The rest is a problem, though. Partitioning and bootloader installation can only be performed from other operating system, e.g., a GNU/Linux distribution. Both can be worked around if HelenOS is not the only system to be installed on the host. But in the other case it is impossible to install HelenOS on its own. Therefore the main goal of this thesis is to provide tools sufficient for successful self-hosted HelenOS installation.

For partitioning Master Boot Record (MBR) and GUID Partition Table (GPT) labels were chosen. Master Boot Record is a legacy partition scheme designed in 1980s. It is probably the most widely used scheme to day. GUID Partition Table is the replacement for the old MBR. It has much cleaner design and offers more features, e.g., redundancy and checksums.

GRUB is used for bootloading HelenOS. Bootloader itself does not depend on any operating system. What does, though, is the installation of the bootloader. It has to be installed at the right place with the right values encoded. That tool exists only for POSIX-compliant systems, which HelenOS is not. Writing a new tool is possible, but requires deep understanding of the image-compilation process. Thus porting existing solution was chosen since in theory it only requires replacing library calls.

In Section 1 we first skim over some basic information about HelenOS itself. We will talk about a bit of history and describe what makes the system special. Anyone already familiar with it may skip this Section.

Following with Section 2 we describe and explain the details of MBR and GPT. We will also suggest some possible designs for their implementation. Implementation of GRUB is described as well as comparison between GRUB Legacy and GRUB 2.

Next in Section 3 the details of prototype implementations of MBR and GPT labels are described. We will discuss what decisions were made and why and what effects they have. A description of a partition editor implementation follows. In the rest of the Section we also look at porting GRUB to HelenOS - how much work it required and what is the result.

Finally the process of installation itself is described in Section 4. Each step is explained and command examples are given.

1. HelenOS Overview

When Linus Torvalds released the first version of Linux kernel to the public, it generated quite some attraction. Some people really liked it and later became developers themselves. Other people, like Andrew Tanenbaum, considered the Linux kernel obsolete. This sparked a debate [4] that hasn't been resolved to this day. The question of course is which of microkernel and monolithic kernel design is the better way to go. It has to be noted, though, there are also other designs than these two, namely hybrid kernels, exokernels, etc.

Microkernels are usually designed to be as small as possible in terms of either number of features or code size. Only essential features remain in the kernel itself while the rest, like virtual filesystem, device mapper and networking, are implemented as servers. Kernel of a monolithic design, on the other hand, is a single executable image containing everything. It aims to be simple to develop and work with. Both designs have their own pros and cons.

While working on his university assignments, Jakub Jermář chose microkernel design. He developed the SPARTAN microkernel which is now used in HelenOS. The kernel now supports multitasking, virtual memory, symmetric multiprocessing and implements the HelenOS IPC[1] mechanism upon which all the user space services are built.

HelenOS is an experimental operating system built from scratch on top of SPARTAN. It is a set of services, libraries and applications. One of its goals is portability so it runs on several different architectures. It also strives to be as modular as possible and eventually become a fully usable system.

IPC Inter-Process Communication (IPC) [5] is the core functionality of a microkernel-based system. In HelenOS the communication can be both asynchronous and synchronous. The IPC implementation can be explained using caller-callee analogy. The caller picks up a phone, calls the callee and leaves it a message in its message box. Then the caller may either wait for a reply (simulating synchronous communication) or do something else, even call someone else. Then the caller may pick up the answer whenever it likes (as per asynchronous communication). In the meantime the callee picks up the call from the message box, processes it and either responds to it itself or passes it on to another service.

IPC can be used for:

- Short calls (one argument for method, 5 arguments for payload)
- Sending large data
- Sharing memory
- Interrupt notifications

Servers Servers are programs providing services. In Unix world these are also known as daemons. But in case of microkernel-based operating system they also provide functionality that is usually found in monolithic kernels. As already

[1] Inter-Process Communication

mentioned, this can be virtual filesystem, but also real filesystems, networking and more.

The reasoning behind separating these services from kernel and each other is simple - security, reliability and modular design. Each of these servers runs in its own virtual memory space. This means these processes are isolated and cannot interfere with each other, just like normal user space applications. When one of the servers aborts - and it may be from any reason - it doesn't bring down the whole system, but only the server. That server can be then restarted and its functionality restored. And finally, if resources, e.g., memory, disk or even processing power, are tight, the system can be stripped down of unnecessary modules.

Devices The *loc* service registers detected devices and creates respective files in the `/loc` folder subtree. For our intentions we need to access our hard disk. When running under qemu [6], the hard disk will usually be connected as an ATA[2] device. These devices are accessible as normal files. First disk is represented as `/loc/devices/\hw\pci0\00:01.0\ata-c1\d0`

[2]Parallel AT Attachment, standard interface for hard disks, optical drives, etc.

2. Analysis

To successfully install HelenOS from its running image, several issues need to be addressed. Firstly, the target media (usually a hard disk) needs to be set up and prepared before the system files can be copied over. Lets see what specific tasks need to be done:

Partitioning Partitions are specific contiguous areas of a disk. Each partition contains either a filesystem or subpartitions. See Section 2.1.

Formatting For the hosting partition a filesystem is needed. There are tools that take care of this for each filesystem.

Bootloader Bootloader is a program that loads the kernel and other important files into memory and passes execution to the kernel. See Section 2.3.

Copying Lastly, all OS files need to be copied over to the destination filesystem.

2.1 Partitioning

To set up a host partition, we need to choose a partition scheme. On architectures IA-32 and AMD64[1] there are two layouts that are most commonly used: MBR and GPT. The MBR, Master Boot Record, is of an older design from 1980s. Although it was designed for much smaller disks, it can cope with up to 2TB disks. On the other hand, the GPT, GUID Partition Table, was designed to replace MBR and fix some of its flaws. GPT can handle larger disks and many partitions, features redundancy, CRC32[2] checksums and is quite extensible. More detailed descriptions of these two now follow.

2.1.1 Master Boot Record

MBR[7] was designed as a small and simple scheme. It fits in only one disk sector and that is always the first one. MBR is the name of that sector. Inside that sector is space for bootloader code, disk ID, four partitions and a signature. This is described in more detail later on. These four partitions are called the *primary partitions*. Each of these partitions consists of a status field, type field, starting address and length. Again, this will be described later in more detail.

Originally it only supported four partitions, but innovation and technological progress allowed and required more. Nowadays the number of partitions is theoretically unlimited. This was achieved by what is called *logical partitions*.

Logical partitions

Logical partitions were added some time later when disk grew larger and offered more possibilities. Since the reader can clearly see there was no space in the MBR itself for more partitions, a new mechanism was engineered. One (and only

[1] Classic Intel x86 32-bit and 64-bit PCs

[2] 32-bit Cyclic Redundancy Checksum

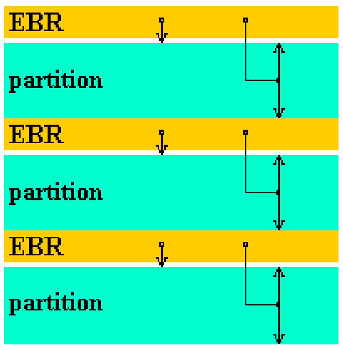

Figure 2.1: Values in the first partition entry in an EBR

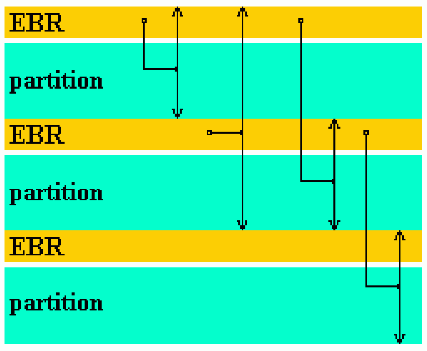

Figure 2.2: Values in the second partition entry in an EBR

one) primary partition may be set to *extended* type. The type field has a special meaning in this case. It means that inside this extended partition there will be a linked list of logical partitions. In a sense, logical partitions are *subpartitions* to the extended one.

The linked list of logical partitions is chained by what is called the *Extended Boot Record* (EBR). These structures have the same format as the Master Boot Record. The difference is the third and fourth primary partitions are zeroed and first and second partitions have special meaning.

The first partition in these EBRs is the actual partition the EBR is for. The starting address field is the difference between its real address on the disk and the starting address of the extended partition, i.e., an offset. The length field of the first partition is its length. See figure 2.1.

The second partition entry in the EBR is not an entry for an actual partition, but instead describes the next partition. As figure 2.2 shows, the starting address is the address of the next EBR, the length is the number of blocks between that EBR and its partition's end and the type is also extended partition. If there is no following partition the whole second partition entry is zeroed.

The boot record format

The Boot Record is divided into these fields:

Boot code Here goes the code that is started by the BIOS. It is responsible for booting the OS. Although is it fairly small, it is enough for passing

Offset	Length	Description
0	440	boot code
440	4	disk signature (optional)
444	2	zeroes (usually, but field is optional)
446	64	table of primary partitions
510	2	MBR signature

Table 2.1: The Master Boot Record sector description

execution to other code - that is how GRUB works. More on GRUB in section 2.3.1. Note: the boot code can actually be 446 bytes long due to following optional fields.

disk signature This is a 32-bit disk signature from Windows NT. It is also used in other systems, including Linux. This field is optional.

partition table This table holds four 16-byte partition records. This means there can be only four so-called primary partitions.

MBR signature The signature is two bytes: 0x55 followed by 0xAA. Classic Intel PCs are little endian so this translates to 0xAA55.

A partition entry consists of these fields:

Offset	Length	Description
0	1	Status
1	3	Starting sector in CHS
4	1	Type
5	3	Last sector in CHS
8	4	Starting sector in LBA
12	4	Length in sectors

Table 2.2: MBR partition entry description

Status This single byte field has usually only two values: passive or active. It was used by early bootloaders that scanned the MBR for active partition and if it found it, it executed the code found there.

Starting address in CHS The address of the first sector of the partition. Address is in CHS coordinates[3].

Type Single byte field describing use and/or partition's filesystem. Some typical values are 0x00 for unused partition, 0x05 for extended partition and 0xEE for GPT partition inside a GPT Protective MBR (see section 2.1.2).

Last sector address in CHS Address of the last sector in CHS addressing.

Starting sector in LBA Address of the first sector in LBA[4].

[3]Cylinder-Head-Sector; obsolete addressing which relies on physical properties of hard disks.

[4]Logical Block Addressing is the replacement for CHS. It does not suffer from its limitations and is simpler to work with.

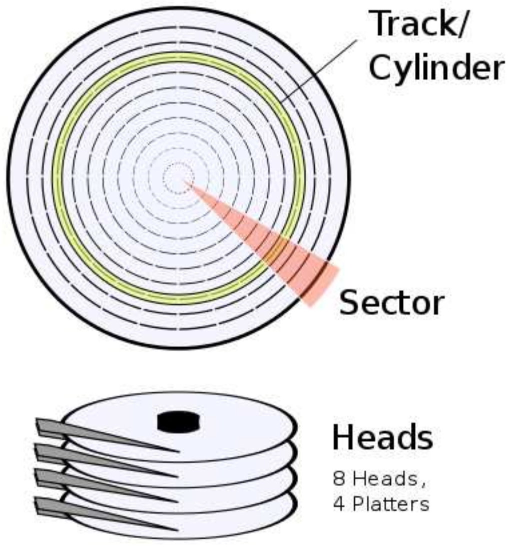

Figure 2.3: The Cylinder-Head-Sector scheme. Cylinder refers to the whole vertical stack of tracks on all platters.

Length in sectors The length of the partition in number of sectors it spans over.

Addressing

The reader now may be wondering why there are two kinds of addressing both describing the same thing. The Cylinder-Head-Sector (CHS) is the originally used kind of addressing. The name and its principle comes from the way hard disks work. Hard disks are composed from multiple stacked magnetic disks. Each of these disks has two sides that can carry the data. So for each of these sides there is a head. The head moves between the outer and the inner rim of the disk, where data are stored in circles, called cylinders. Each cylinder consists of a number of sectors. So CHS describes which head should read which sector on which cylinder. See figure 2.3 for illustration. This addressing was initially sufficient, but soon disk sizes outgrew the capabilities of such system. That was caused by the fact that number of cylinders and sectors grew while the number of heads remained the same. The initial design did not count on such data density so the bit width for the number of cylinders and sectors was not enough. That was at first overcome by ignoring the physical properties of disks and the address could contain something like 1024 heads which is nonsense in the real world. But it still was not enough for the ever growing disk capacities.

The Logical Block Addressing (LBA) was the answer to that. LBA is a single number meaning number of sectors from the beginning of the disk. Although this meant the disk controllers had to be more complex to translate the address to its physical representation, it lost the complexity of CHS and its limitations. Instead only single limitation remained - the 32 bits of the four-byte integer means disks larger than 2TiB[5] are not addressable completely. Workaround for this limitation is to use either a single partition beginning somewhere inside the 2TiB and its length set to 0xFFFFFFFF, or a different label instead of MBR.

Possible implementations

There are multiple ways to represent data and work with them. Let us discuss these options and see how they fit.

Split primary and logical This approach directly represents the actual partitions. The primary partitions would reside in an array, while all logical partitions would form a linked list. This would mean different access to primary and logical partitions, different handling and might cause reimplementing functions for both types of partitions. For this reason this scheme was discarded.

Common list, no rules This representation was initially implemented. The code did not rely on any rules. Primary partitions could be scattered anywhere through the list and if there were less than 4 of them, the rest were unused. This turned out to be problematic, since operating systems rely on ordering of partitions. Changing the order of (either primary or logical only) partitions might make the system unbootable. Partitions were allowed to overlap until the list was to be written to disk. Then there were checks to prevent overlapping.

Common list with rules This is the actually implemented approach. Primary partitions are at the beginning of the list and there are always four of them (some of them may be unused). Other rules are that no two partitions overlap (except for the extended partition[6]) and that they are inside the actual disk. Although we need to differentiate between primary and logical partitions in some functions, the fact is they are all inside a list and so some methods remain simple, e.g., destroying the whole list.

Conclusion Designing smart data structures is not always easy. Historical development of MBR brought such complexity. It made implementation hard with special cases that need to be dealt with. Among these is, for example, the location of EBR for the first logical partition, which always has to be the same. Although implementing the third option interlaced the code with conditional branching, deciding whether the partition is primary or logical, it was simply necessary because of the different natures of both kinds of partitions.

2.1.2 GUID Partition Table

The GUID Partition Table (GPT [8]) came as a replacement for the aging MBR. GPT uses only LBA[7]. It maintains backward compatibility with MBR by saving

[5]This is only valid for 512-byte sector disks, since $512B * 2^{32} = 2TiB$. Newer disks support 4096-byte sectors.

[6] Actually the extended partition has to overlap with all the logical and also completely encapsulate them, which is also checked for.

[7]Logical Block Addressing

a "protective MBR" in the first sector. From MBR perspective the disk contains one partition of unknown type (actually 0xEE as stated before) and no free space.

The GPT is a modern label featuring redundancy, checksums, extensibility and supports very large disks. From implementation perspective it is simpler to work with. The design came hand in hand with the UEFI[8] specification [9] which replaces BIOS altogether.

GPT is divided in two parts: the header, containing all the metadata, and the partition entry array, containing all partitions. The GPT header is stored in the first logical block, LBA1, and in the last addressable block, LBA -1, providing redundancy. It contains all GPT-specific data. The GPT partition array guarantees space for at least 128 partitions each of 128 bytes of metadata and is stored in LBA2 - LBA33 (at least, with 512-byte sectors) with redundant copy at the end of the disk. See section 2.1.2 for clarification.

The GPT label formats

All fields are written little endian. Firstly let us look at the header:

Offset	Length	Description
0	8	Signature
8	4	Revision number
12	4	Header size
16	4	Header CRC32
20	4	Reserved; must be zero
24	8	Current LBA
32	8	Backup LBA
40	8	First usable LBA
48	8	Last usable LBA
56	16	Disk GUID
72	8	Partition table LBA
80	4	Number of partition entries
84	4	Size of each partition entry
88	4	Partition table CRC32
92	*	Reserved; must be zeroes for the rest of the block

Table 2.3: GPT header description

Signature The string "EFI PART", 45h 46h 49h 20h 50h 41h 52h 54h ('h' for hexadecimal).

Revision number For GPT versions 1.0 through 2.3.1 the revision number is 00h 00h 01h 00h.

Header size Header size in bytes. Must be greater than 92 and less or equal to the logical block size.

Header CRC32 A CRC32 checksum for the whole header (the whole block). During computation this fields is to be set to zero.

[8]Unified Extensible Firmware Interface

Current LBA LBA of the current header. The value is 1 for the main header or the LBA of the last accessible block for the backup one.

Backup LBA Flipped with Current LBA.

First usable LBA GPT offers theoretically infinite number of partitions. Therefore the first usable LBA may not be always 34. Also, the specification states there should be at least 16,384 bytes reserved just for the partition table. This means First usable LBA is at least 34 (for 128-byte partition entries on 512-byte sectors).

Last usable LBA Similar to First usable LBA. Usually the last LBA - 35.

Disk GUID Globally Unique Identifier. A unique number for disk identification.

Partition table LBA Address where the table starts. For main header it is usually 2, for backup header it depends on the size of the table (usually LBA -34).

Number of Partition Entries Number of active partitions in the table.

Size of each partition entry Size of the partition entry in bytes. Must be a multiple of 128.

Partition table CRC32 CRC32 checksum of the whole partition array.

The partition entry contains:

Offset	Length	Description
0	16	Partition ID
16	8	Starting sector
24	8	Ending sector
32	8	Attributes
40	72	Partition name

Table 2.4: GPT partition entry description

Partition ID A globally unique ID (GUID) of the partition.

Starting sector Address in LBA.

Ending sector Address in LBA.

Attributes 64-bit attribute field.

Partition name A null-terminated string.

Partition array

There are minimum 16,384 bytes reserved for partition array. With the usual 128 bytes for each partition that means 128 partitions. That is the minimum required by the specification. For 512-byte sectors the First Usable LBA is 34 and for 4k-sectors it's 6, similarly for the Last Usable LBA.

Possible implementations

In this case the design was fairly simple due to GPT's through-thought design and clear specification document.

A resizeable array An array offers ease of use and performance due to its simplicity. Even if the initial number of partitions were not enough a new array would be allocated replacing the old array with its contents copied over. If the new array has double the capacity of the old one and there were unreasonably many partitions, the amortized time to add a new one would still be a constant. The bonus is the array can be immediately written to the disk without any data mangling or conversion. This includes endianity conversion because all values are stored as little-endian by default (conversion back and forth happens inside library functions).

This option has been selected and implemented.

A linked list A list can offer adding a partition anywhere in constant time assuming a pointer is provided. Getting a pointer may cost linear time, though. The advantage is then no data copying. The disadvantage is slower random data access. Since not much data is expected the advantage is not large at all, the same may be valid for the disadvantage. But if we are unlucky traversing the list may keep causing pagefaults, slowing other processes. Another reason is the writing process would have to be adapted to traverse the list and write partitions one by one.

Conclusion The decision for data structure was driven simply by practicality of the solution. While developers are usually concerned with time and space constraints, this is not the case. The size of the data representing all the partitions will not cross the 16kiB boundary in most cases, so the array will not ever resize. And even if it was ever needed, the code already supports it.

2.2 Formatting

The process of setting up a filesystem is taken care of by using tools mkfat and mkmfs for FAT and MinixFS, respectively. Similar tools for other filesystems may be implemented in the future. All we need to know about these is how to use them.

2.2.1 Usage

After creating partitions using a partition editor, a service is needed to detect them and set up their respective device files. For example, after setting up a MBR label, we simply run the *mbr_part* service. We need to supply an argument to that command, which will be a device where we want to detect the partitions. An example may be the first disk as mentioned at the end of Section 1. This sets up a file `/loc/devices/\hw\pci0\00:01.0\ata-c1\d0p0` representing the first partition on the first disk.

Now we are able to create, for example, a FAT filesystem structure on this partition by issuing the command `mkfat devices/\hw\pci0\00:01.0\ata-c1\d0p0`. Note that this command may also receive other parameters.

2.3 Bootloader

The purpose of a bootloader is to load a kernel and possibly some modules or a ramdisk. HelenOS was using GRUB 0.97 (Legacy) to boot up, but during writing of this thesis GRUB was updated to GRUB 2 but it still supports the old one.

GRUB Legacy was recently marked as obsolete and replaced with GRUB 2. GRUB Legacy is still in use today even though it is not developed anymore. Many still prefer it to its successor since it is so powerful yet simple to use. Among other reasons for that are the complexity of GRUB 2 configuration and the fact that GRUB Legacy does its job and does it well. The difficulties of maintaining, adding new features and bugfixing GRUB Legacy have created the need for a new design.

2.3.1 GRUB Legacy

GRUB Legacy[10] was originally developed as a Multiboot specification reference implementation. GRUB stands for ,,GRand Unified Bootloader" (a play on Grand Unified Theory). The aim was to create a bootloader for the x86 architecture for booting Multiboot-compliant kernels. Features include menu interface, network boot, chain-loading, etc. GRUB was handy in different scenarios - when properly set up it was very powerful tool.

But it is quite complex piece of software, too. It is mostly written in C but some parts are in assembly code. The probably biggest problem is in the structure of the software. The same code is used twice - in stage2 during boot-up and in the executable that runs under already running OS. This means this executable uses the same low-level operations as the stage2. Although code reuse is usually a good approach, in this case it makes the code hard to understand by mixing two separate things together. As time went on the complexity of it made the code inextensible and hard to maintain.

GRUB Legacy works in stages. First, BIOS executes the boot code in the MBR. That is where *stage1* resides. Those 440 bytes handle some basic errors but otherwise is simple. It contains the address of the next stage where it hands over the execution. The next stage can be either *stage1.5* or directly *stage2*. Stage1.5 contains code that can mount different filesystems. It mounts the filesystem where the stage2 resides and loads that for execution. Stage2 then loads the configuration file and any needed modules. It displays the menu interface and finally boots the OS itself.

Omitting stage1.5 means stage1 needs to know the actual sector where stage2 begins so that it can be executed. The presence of stage1.5 enables the administrator to update or somehow else change the stage2 file. Because the underlying filesystem is free to move the file to any place on the partition and also split it into pieces, the sector where it starts can change. But since stage1.5 works on the filesystem level, it does not care where on the disk stage2 actually is. Note that stage1.5 is put between MBR and the first partition in the first 30 kilobytes of space. If that space isn't available, the installation will fail.

Although GRUB succeeded and gained widespread use, the flawed design forced the developers to start over. So around 2002 GRUB 2 was announced.

2.3.2 GRUB 2

GRUB 2[11] works very similarly to its predecessor. It also has stages. The first stage, a file called *boot.img*, is compiled and written in the boot code area in MBR, just like stage1 in GRUB Legacy. The first stage can load a sector from any 48-bit LBA address (unlike stage1 from GRUB Legacy). The second stage, named *core.img*, can be either inside the free space between MBR and the first partition or inside a filesystem on a partition. The core.img stage loads all needed modules and configuration.

GRUB 2 is more capable:

- more architectures and configurations
- support for more filesystems
- more commands
- powerful system of generating configuration file
- support for dynamically loadable modules

But these advanced features also have shortcomings - they add complexity. This, with GRUB Legacy's stability and usability, has caused somewhat slower adoption of GRUB 2 which is why it is still being migrated to from its predecessor. Many people preferred the old simple configuration over modern scripts that detect operating systems automatically including some HelenOS developers.

In cases where there is a GPT-labeled disk but no UEFI and there is only BIOS, GRUB 2 also natively supports the GPT label. In such case GRUB installs in a special partition called *BIOS Boot Partition*. This type of partition is specifically designed for this purpose. Otherwise UEFI is capable of booting operating systems directly.

In the end decision was made GRUB Legacy port is sufficient and porting GRUB 2 is not at this time worth the trouble. Not only because it would not bring any advantages or important features, it would also be fairly stripped of its own features. The reason is GRUB 2 generates all its configuration file using common shell scripting, which is simply not available to HelenOS.

2.4 File copying

Perhaps one day HeleneOS user will be able to simply run recursive copy and duplicate the root folder on the destination device. However, this is not the case today. All of applications, servers and data that are present in live instance of HelenOS are inside RAM, where they have been loaded beforehand with some help from the bootloader. This means two things:

1. No kernel is available as a file. The /boot directory is empty, since all files were either in the image.iso file or inside the initrd.img (ramdisk) file, which itself is in that image.iso as well.

2. HelenOS actually expects the root to be in the ramdisk. So far it does not mount any permanent storage device for mounting root.

Solutions There are two very different ways to solve these problems:

1. We could solve both of these by creating an application for dumping RAM contents onto disk. This way we possibly could create all the needed files.

2. A different approach is simple: copy all needed files into the ramdisk during its build phase. Since the ramdisk itself has to be present as well, it means the ramdisk has to be built twice.

The first solution obviously saves space, because such application would be much smaller compared to the ramdisk itself. It also saves time, since it takes double the time to load a ramdisk double its size. On the other hand, this application would require time to make and probably would become obsolete anyway. The second approach requires only small changes to the build process, but doubles the size of the resulting image. But thanks to the small size of the image anyway and fast modern computers, doubling the size seems like a reasonable trade-off for the quick implementation of the second option.

3. Implementation

Implementation is divided into two branches. Both contain the MBR and GPT implementation along with a partition editor while one of them includes GRUB Legacy.

MBR and GPT were implemented as libraries *libmbr* and *libgpt*, respectively. They provide an interface that can be used in any other program. The partition editor is called *hdisk*. Both libraries can be found in the `uspace/lib` folder as `mbr` and `gpt` subfolders. GRUB Legacy and hdisk are located in `uspace/app` directory as `grub` and `hdisk` subfolders.

3.1 libmbr

Master Boot Record implementation was uneasy due to lack of an actual specification document. Most knowledge was acquired on Wikipedia, but testing and troubleshooting with *fdisk* and the *hexdump* utilities were just as much required. Other online documents were also helpful.

3.1.1 The label structure

The implementation is hidden in several levels of abstraction. The user of the library uses only the general structure `mbr_label_t` as a parameter for most of the library functions. This can be viewed as the `this` parameter in C++ and other object-oriented languages.

The first field in this structure is a pointer to the MBR header container. That structure simply holds all data in the MBR block at the first logical block. It is needed to mention the four primary partitions are encoded into this structure when calling `mbr_write_partitions()`. They are not updated during any other library call.

The second field holds a pointer to a structure representing the actual partition list. It also contains numbers of primary and logical partitions. See section 3.1.2 for details.

The last field in the label structure holds the device identification number. This is the number used by the *libblock* library and *loc* server.

3.1.2 Data representation

The list is implemented using the circular doubly-linked list implementation in the *libc* library. It is not a clear list though - the differences between primary and logical partitions brought the need to make some assumptions about the list. The main added feature is that the list cannot grow below 4 items. It always has at least 4 partitions. If they are not being used they are nulled and their flag is set to unused.

Other assumption is that the list is always consistent - meaning it can be always written at any time between library calls and results will be expectable. Consistency is only achieved by using proper functions for altering the partition list. Such functions are only these:

- `mbr_read_partitions()`
- `mbr_add_partition()`
- `mbr_remove_partition()`

These assumptions can be broken by navigating through the list and changing values by other means. Such action is discouraged.

These guarantees were implemented inside the library functions. They do not form any other abstraction above the provided doubly-linked circular list.

3.1.3 Usage

Typical use of the library is the following:

1. Call `mbr_alloc_label()` to allocate and initialize the label structure.
2. Call `mbr_read_mbr()` to read the MBR header from a device. This initializes the first and the last field.
3. Call `mbr_read_partitions()` to parse primary partitions from the header and read and parse logical partitions. This initializes the second field in the label structure.
4. However, adjusting the partition list is needed. This can be done by:
   - Adding a partition by calling `mbr_add_partition()`.
   - Removing a partition by calling `mbr_remove_partition()`.
5. Call `mbr_write_partitions()` to write all changes onto a device. This writes the MBR header as well as all the logical partitions. It checks whether EBR locations are possible and fixes them if not.
6. Call `mbr_free_label()` to free the allocated memory.

Adding a partition is performed as follows:

1. Call `mbr_alloc_partition()` to allocate a partition.
2. Call `mbr_set_flag()` if logical partition is wished.
3. Set partition start, length and type. Logical partition's EBR address can also be set, but is optional.
4. Call `mbr_add_partition()`. This performs checks on partition dimensions inside the disk and among other partitions.

Accessing the list and its contents is possible in three ways:

1. Use `mbr_part_foreach`. This is a macro that iterates through the list as a for cycle. Handy for simpler use cases.

2. Use `mbr_get_first_partition()` and `mbr_get_next_partition()`. The first function return a pointer to the first partition and the second returns pointer to the partition right after the provided or NULL if there are no more.

3. Use `mbr_get_list()` and iterate through the list manually. This approach is not recommended and is provided for extreme cases only.

3.2 libgpt

The GPT label has a overall cleaner and future-proof design than the MBR label which was helpful during implementation. The design and its advantages have already been mentioned in Section 2.1.2.

3.2.1 The label structure

To keep both libmbr and libgpt APIs[1] as similar as possible, libgpt also has a central label structure that works similarly to the *this* handle. The structure holds the same fields as its libmbr counterpart. The difference lies in the way the partition list is represented.

Representation As the design suggests, the partition list is represented by a dynamic array. This array can grow up or down (but never to less than 128 partitions) depending on the number of accommodated partitions. This is implemented simply by allocating a new array every time a different size is needed and all items copied over. Note that the array does not grow up by one but it doubles it size. Similarly, when growing down, it waits until it is by a constant less than half-full and then it resizes.

3.2.2 Usage

Typical use of the library is very similar to that of libmbr:

1. Call `gpt_alloc_label()` to allocate and initialize the label structure.

2. Call `gpt_read_header()` to read the GPT header from a device. This initializes the first and the last field.

3. Call `gpt_read_partitions()` to parse partitions. This initializes the second field in the label structure.

4. Adjust the partition list however is needed. This can be done by:
    - Adding a partition (more details below).
    - Removing a partition by calling `gpt_remove_partition()`.

5. Call `gpt_write_partitions()` to write all changes onto a device. This writes the GPT header as well.

[1]Application Programming Interface

6. Call `gpt_free_label()` to free the allocated memory.

Adding a partition can be performed in three different ways:

1. Using `gpt_get_partition()`. This is the recommended way since it is simple and effective. Usage:

    (a) Call `gpt_get_partition()` to get a poiner to a structure already inside the array.

    (b) Set partition start, length, type, name and optionally attributes.

2. Using `gpt_add_partition()`. This is not as effective as the previous method because it has to copy memory from the allocated partition structure to another partition structure already inside the array. It was implemented mainly for keeping APIs similar. Usage:

    (a) Call `gpt_alloc_partition()` to allocate a partition.

    (b) Set partition start, length, type, name and optionally attributes.

    (c) Call `gpt_add_partition()`. This performs checks on partition dimensions inside the disk and among other partitions.

    (d) Previous two steps can be repeated as many times as needed.

    (e) Call `gpt_free_partition()` to free the memory allocated for the partition structure.

3. Using `gpt_get_partition_at()`. This function returns a pointer to the partition structure at a specific index in the array. This was implemented to allow empty spots between partition entries. Usage:

    (a) Call `gpt_get_partition_at()` to get a pointer to a structure already inside the array at an index.

    (b) Set partition start, length, type, name and optionally attributes.

Accessing the list and its contents is possible in two ways:

1. Use `gpt_part_foreach`. This is a macro that iterates through the list as a for cycle. Handy for simpler use cases.

2. Use `gpt_get_partition_at()`.

3.3 hdisk

Hdisk was designed to be only a user interface to both libmbr and libgpt. It is not, however, dependent on any of them. It is extensible by implementing a specific object and its methods, that is common to all label-manipulating libraries.

3.3.1 User Interface

The user interface is simple, similar to that of *fdisk* known from GNU/Linux distributions. User inputs single letter to invoke each command which may request more input afterwards. Example of that is simply adding a partition. Some common commands are:

a Adds partition.

d Deletes partition.

h Prints list of commands.

p Prints current partition list.

r Re-reads partition records from the device.

q Quits hdisk.

w Writes the partition list to the device.

Hdisk is also extensible beyond the basic command set. Any label extension can implement command *e*, which may contain a new command set. Returning from this command resumes normal operation.

When started, hdisk automatically tries to guess current label by calling `mbr_is_mbr()` function. If MBR or GPT is detected, it is read. The first thing to usually do is to invoke the command *p* to print whatever has been detected.

3.3.2 Label extensibility

To make using label-manipulating libraries easier, a common object has been designed. This is a simple structure containing information about current label format and pointers to label-specific functions, which serve as entry points to library functions. This greatly simplifies hdisk's main loop as it always calls functions using these pointers. Let us look at this structure:

```
typedef enum {
        LYT_NONE,
        LYT_MBR,
        LYT_GPT,
} layouts_t;

union label_data {
        mbr_label_t     *mbr;
        gpt_label_t     *gpt;
};

typedef struct label label_t;

struct label {
        layouts_t layout;
        aoff64_t nblocks;
        service_id_t device;
        union label_data data;
        unsigned int alignment;
        int (* destroy_label)(label_t *);
        int (* add_part)      (label_t *, tinput_t *);
        int (* delete_part)   (label_t *, tinput_t *);
        int (* new_label)     (label_t *);
        int (* print_parts)   (label_t *);
        int (* read_parts)    (label_t *);
        int (* write_parts)   (label_t *);
        int (* extra_funcs)   (label_t *, tinput_t *);
};
```

enum layouts_t This is an identifier of current label format.

union label_data Common pointer to the specific label structure.

struct label Common label object as described earlier. Each of these function pointers should be have a defined function for each specific label. Their function is clear from their names.

3.4 GRUB Legacy porting

To understand the way GRUB Legacy works we can look at its build process. This was achieved by configuring and building it on a GNU/Linux system and saving a log from the `make` process in a file. This was a major help in understanding its implementation.

The `stage2` folder is the core. It contains the main program logic, commands, filesystem support, etc. The actual stage2 and stage1.5 are built using these sources. Stage1 is generated from a single file `stage1.S` written in assembly code. The user space executable `grub` is built using code from folders `stage2`, `grub` and `lib`. Code from the `grub` directory creates only a user space entry point for stage2. The `grub` executable is also statically linked with a library assembled from source code in `lib`. This static library mainly generates file names for hard disks in all the supported operating systems so that they can be detected. While in GNU/Linux system this is as simple as `/dev/sda`, in HelenOS ATA disks are accessed as `/loc/devices/\hw\pci0\00:01.0\ata-c1\d0` and similar. This code can be found in the `lib/device.c` file. If the ATA disk naming scheme is ever changed in HelenOS and this GRUB Legacy port is still being used, this code will need to be updated.

GRUB Legacy also required many small changes in many different places. These were mostly simple, though, requiring including different header files and slightly different function calls, e.g., replacing `strlen()` with `str_len()`. A major change was required to code simulating BIOS read/write functions. This had to be implemented using *libblock*, the *loc* service and IPC. The code can be found in the `grub/asmstub.c` file around the 1100th line.

3.5 Build procedure changes

As previously noted, we decided to include all components, kernel and initrd.img image file into initrd.img file itself. We do this only for one level of recursion. The resulting image.iso is therefore twice the size than previously.

To create the image this way all we needed was to add one more target into the boot/Makefile file and slightly change boot/arch/ia32/Makefile.inc. The added target is:

```
build_full_initrd:
        rm -rf $(DIST_PATH)/boot/*
        mkdir -p "$(DIST_PATH)/boot/grub"
        make "$(INITRD).img"
        for module in $(COMPONENTS) ; do \
                cp "$$module" "$(DIST_PATH)/boot/" ; \
        done
        for file in $(GRUB_FILES) ; do \
                cp "$$file" "$(DIST_PATH)/boot/grub/" ; \
        done
        make "$(INITRD).img"
```

File boot/arch/ia32/Makefile.inc needed modifying to actually make that target:

```
#PREBUILD = $(INITRD).img
PREBUILD = build_full_initrd
```

4. Installation

All requirements are met and so installation is now possible. Description of this process now follows. Each step includes command-line examples.

Boot up It is required to boot up from such HelenOS image containing all components, kernel, initrd.img image, GRUB Legacy files, grub binary and hdisk. All these are included and/or built using the provided bazaar branch.

```
bzr merge lp:grub4helenos
```

Partition the disk First thing to do is to partition the hard disk. To do this, we run the hdisk partition editor. The argument will be an ATA or SATA drive that is recognized by HelenOS. Typical invocation while running on Qemu is `hdisk devices/\hw\pci0\00:01.0\ata-c1\d0`. At the time of writing HelenOS only supports creating either FAT or MinixFS filesystems so one of these should be selected as partition type. Only one partition is needed. Hdisk usage has already been described in Section 3.3.

Format the partition The created partition now needs to be recognized by the system. For this either `mbr_part` or `guid_part` needs to be run. Selection between those two depends on which was selected in previous step, MBR label or GPT label, respectively. Now it is time to create a filesystem using, e.g., `mkfat` for FAT.

```
mbr_part
mkfat --type 16 devices/\hw\pci0\00:01.0\ata-c1\d0p0
```

Copy all necessary files First of all, the destination filesystem has to be mounted. This requires starting the *fat* service. Then we create a folder to use as a mountpoint and mount the filesystem using the `mount` command. All components, kernel binary and initrd.img file are stored in the `/boot` directory. GRUB Legacy stage1, stage2 and menu.lst files are located in the `/boot/grub` folder. This means all we have to do now is to copy recursively the whole `/boot` directory and unmount.

```
fat
mkdir /mp
mount fat /mp devices/\hw\pci0\00:01.0\ata-c1\d0p0
cp -r /boot /mp
```

Setup bootloader The last step involves running only the `grub` command. First we set the root, then let grub find the stage1 file and finally run the setup itself.

```
grub
>root (hd0,0)
>find /boot/grub/stage1
>setup (hd0)
```

Reboot System is now ready to boot off hard disk.

Conclusion

The aim of this thesis was to allow the user of HelenOS to install the system on their hard disk without help of any third-party software. The thesis describes the whole process of installation and covers those parts which were originally missing. It describes two different and most-widely used label formats and discusses their advantages, disadvantages and implementation details. We also look at two common boot loaders and discuss their design.

Additionally it creates a new use-case scenario, where HelenOS may be used as a rescue system for repairing broken labels or reinstalling the boot loader.

Accomplishment

It can be said the main aim has been accomplished. Using only hdisk, the provided partition editor, GRUB Legacy, the boot loader, and some common commands already present in HelenOS system, the user is now able to install HelenOS. From now on the user does not depend on other systems to provide necessary tools for installation. HelenOS is one more step further.

Further improvement

Follow-up work may include porting GRUB 2, designing completely new partition scheme (similarly to, for example, BSD slices) or even implementing graphical partition editor. As previously noted however, GRUB 2 is not at this time worth the work. It does not bring any important features. And similarly for a graphical application, HelenOS is not yet ready. Features are still being added and changed and the platform as a whole does not seem stable enough. Conversely, the work may be extended in areas which are already standardized which is the case of GPT. One interesting feature that could be implemented, though, is *LVM*, Logical Volume Manager. LVM is quite similar to partition labels, but allows resizing partitions and extending them over several disks.

Developing for HelenOS

HelenOS is a new, live and strange platform. It is being designed and developed from scratch. It does not need to follow any rules nor standards. From this perspective it is very exciting environment.

However, the unconventional nature may also be the source of confusion and misunderstanding for HelenOS beginners. This can be illustrated with classic user input output functions: while `printf()` and `scanf()` are both standard functions, the latter is still missing. Its functionality had to be replaced with *clui* library and integer parsing functions from libc.

Another trouble is there is no specific goal. Writing a completely new general purpose operating system from scratch can be very tiresome, even more considering the user space applications have to be implemented as well. At least there is a library providing POSIX standard functions, but the developer still has to do some work to port an application. Some minimum effort needs to be put into setting up the compilation process and integrating the source code into the HelenOS source tree and build system. Fortunately, progress is being made also in this regard. One of the HelenOS developers introduced a set of scripts to help with this process. This tool is pointedly named *HelenOS coastline* [12] and can be downloaded using git.

And now back to the general-purpose feature. If HelenOS were specifically aimed at certain subset of use-case scenarios, the developers might concentrate their effort in both design and implementation. Without any focus developers may very well do whatever they want however they want. This may subsequently create arguments on topic of design and features as it recently happened. On the other hand, however, this freedom allows for unconventional ideas and solutions. Too many rules and premature design decisions may inhibit creative thinking and restrict people's minds. Perhaps there is a middle ground where these two principles meet and one can enjoy advantages of both without too much trouble. But HelenOS simply took one path and it certainly will be interesting to see where it gets.

A. Contents of the attached CD

A part of this thesis is also a CD containing this thesis itself, complete source code, documentation, bootable HelenOS image with Qemu scripts and a hard disk image with HelenOS installed.

This is the CD folder subtree:

Folder	Contents
`Documentation/`	This thesis, its LaTeXsource files and Doxygen-generated technical documentation built from source files
`Images/`	Bootable ISO image, bootable hard disk image and Qemu scripts
`Sources/`	One HelenOS branch containing all written source code

B. Building and booting HelenOS

Building from sources The attached CD contains one HelenOS bazaar branch in the `Sources` directory containing all of libmbr, libgpt, hdisk and GRUB Legacy. After unpacking the tarball it is recommended to install a cross-compiler. Even though the usual GCC may work, it is not supported and things may break. To install the cross-compiler run a script in the `tools/` subfolder:

```
cd tools
./toolchain ia32
```

Note that we are using the IA-32 architecture. The reason is GRUB Legacy was designed solely for this platform. Fortunately, AMD64 processors still support it.

After cross-compiler installation finish we can return to the root of the branch and build an image according to the default configuration by issuing the following commands:

```
cd ..
make PROFILE=ia32
```

This generates a file named `image.iso`, which is a CD image containing the GRUB boot loader, HelenOS kernel, its components and a ramdisk.

Running the system In the `Images` directory on the CD we can find two simple scripts for booting HelenOS from CD image and from a hard disk image. These scripts contain only Qemu execution with some default parameters. Both use the same hard disk image. To test the functionality we follow these steps (assuming we have already copied all necessary data from the CD onto our hard disk):

1. For testing the build process, build a bootable image.iso from the sources. This step is optional, the same resulting image.iso can be found in the `Images` directory.

2. Create an empty hard disk image:

   ```
   dd if=/dev/zero of=hdd.img bs=1M count=60
   ```

   The *count* parameter declares the size of the disk image in MiBs. Minimum size is about 30MiB. This step is optional, an image is already provided. The provided image is already formatted and has HelenOS installed.

3. To test the installation process, boot the image.iso using the script provided:

   ```
   ./start-qemu-cd.sh
   ```

   The whole installation process has been described in Section 4. Note the script has file names hard-wired.

4. To test whether the installation was successful, boot HelenOS from the hard disk image using the second script provided:

   ```
   ./start-qemu-hdd.sh
   ```

If HelenOS boots correctly and displays a working desktop, the installation has been successful. Again, note the script has file names hard-wired.

Bibliography

[1] HelenOS operating system `www.helenos.org`

[2] PRINC, František *HelenOS ext4 filesystem driver*. 2012

[3] SUCHA, Martin *Ext2 Filesystem Support for HelenOS*. 2011

[4] LINUX is obsolete
`https://groups.google.com/d/msg/comp.os.minix/wlhw16QWltI/XdksCA1TR_QJ`

[5] IPC for Dummies `http://trac.helenos.org/trac.fcgi/wiki/IPC`

[6] Qemu, processor emulator `http://wiki.qemu.org/Main_Page`

[7] Master Boot Record `https://en.wikipedia.org/wiki/Master_boot_record`

[8] GUID Partition Table `https://en.wikipedia.org/wiki/GUID_Partition_Table`

[9] UEFI Specification, Section 5 `http://www.uefi.org/specs/`

[10] GRUB Legacy `https://www.gnu.org/software/grub/grub-legacy.html`

[11] GRUB 2 `https://www.gnu.org/software/grub/`

[12] HelenOS Coastline
`http://vhotspur.blogspot.cz/2013/03/introducing-helenos-coastline.html`

List of Abbreviations

ATA	Advanced Technology Attachment
API	Application Programming Interface
BIOS	Basic Input-Output System
CHS	Cylinder-Head-Sector
CRC32	32-bit Cyclic Redundancy Checksum
EBR	Extended Boot Record
GPT	GUID Partition Table
GRUB	GRand Unified Bootloader
GUID	Globally Unique IDentifier
IPC	Inter-Process Communication
LBA	Logical Block Addressing
MBR	Master Boot Record
UEFI	Unified Extensible Firmware Interface