Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Evidenčné číslo: 48ddf94d-1aa1-41c8-aee4-57756ba46b95

Ext2 Filesystem Support for HelenOS

2011

Martin Sucha

Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky

Ext2 Filesystem Support for HelenOS

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	Katedra informatiky FMFI
Školiteľ:	RNDr. Jaroslav Janáček, PhD.

Bratislava 2011

Martin Sucha

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Martin Sucha
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Ext2 Filesystem Support for HelenOS

Cieľ: The goal of the thesis is an implementation of the ext2 filesystem support for the operating system HelenOS. The microkernel operating system HelenOS currently supports only two specific filesystems and the FAT filesystem. Filesystems are implemented in HelenOS using a user-space server vfs that provides applications with an interface to access files. This server passes the requests for file operations on servers implementing individual filesystems. The ext2 filesystem is supported by most of the current UNIX-like operating systems, and it provides additional functions in comparison to FAT.

Vedúci: RNDr. Jaroslav Janáček
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 19.10.2010

Dátum schválenia: 19.10.2010

[illegible]
doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

M. Sucha
študent

[illegible]
Vedúci

Acknowledgements

I would like to thank my supervisor, RNDr. Jaroslav Janáček, PhD. for his guidance and useful advice, which he kindly provided whenever I stopped by his office.

I would also like to thank Mgr. Jakub Jermář for reviewing preliminary versions of the thesis.

Abstract

HelenOS is a multi-server microkernel-based operating system, which supports a wide variety of processor architectures. The system has a filesystem layer supporting multiple filesystems, with implementation distributed among several server processes. Services for client user-space applications are provided by a central virtual filesystem server, while individual endpoint filesystem servers provide implementations of concrete filesystems. The virtual filesystem server passes requests to correct filesystem and implements some high-level operations. HelenOS comes with support for several in-memory filesystems and a single disk-based filesystem — FAT16. While FAT filesystem works well, it is relatively simple and does not provide some advanced functions. As part of the thesis, we added read-only support for the second extended filesystem, which provides additional functions such as sparse files. The filesystem has a very good potential to be incrementally extended, as later versions of the filesystem (ext3, ext4), are partially backward compatible. The filesystem is described in chapter 2. The implementation of the ext2 filesystem in HelenOS is divided among multiple layers. A description of the implementation is provided in chapter 3.

Keywords: HelenOS, ext2, filesystem

Contents

List of Figures

List of Tables

Introduction

Operating systems may be divided into several groups based on how much code is run in privileged mode. An operating system using monolithic kernel runs all system code in the kernel, including device drivers and system services such as filesystem or networking support. On the other hand, microkernel-based operating system runs only a limited portion of code in kernel mode, where only basic system services such as thread scheduler or interprocess communication facilities are present. Device drivers and other system services run completely in userspace. A mixed approach using a hybrid kernel exists, in this case, more code is run the kernel than in a microkernel OS, but some system services are provided by userspace.

Most ubiquitous operating systems used to date such as Windows or Linux are based on a monolithic or hybrid kernel, but multiple microkernel operating systems exist, most of which are limited to a single architecture. HelenOS is a multiplatform, microkernel based, open-source operating system developed by international team of developers that supports half a dozen of processor architectures. The project was started and is mainly developed at the Faculty of Mathematics and Physics at Charles University in Prague, with contributions from all over the world[1]. While the system has basic functions working (and new features are actively being developed), at the time of starting this thesis, the only disk-based filesystem implemented in HelenOS was FAT16. In the mean time, several new people joined the project and started to implement different filesystems, such as cdfs, ntfs, minixfs or ext4. Our choice was the second extended filesystem, because it is widely used in Linux based operating systems and the filesystem implementation may be extended incrementally (as the filesystem is very extensible), possibly supporting newer revisions of the filesystem such as ext3 and ext4, which are backward compatible to large extent. While work on ext4 filesystem was started by a student of Faculty of Mathematics and Physics at Charles University in Prague at the same time this thesis started, we believe some duplication is common in open source projects and it enables evaluation of different approaches.

We aimed to implement read-only support for the second extended filesystem in HelenOS, while leveraging the benefits that microkernel architecture provides. The implementation should integrate seamlessly with HelenOS VFS layer, so that the filesystem can be used in standard way by HelenOS user-space applications.

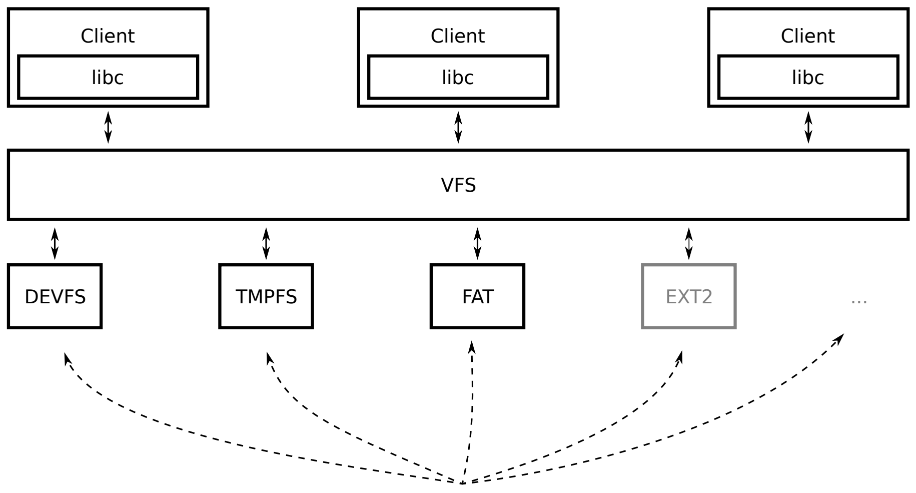
[1]the number of international contributions is expected to rise, as HelenOS was accepted as a mentoring organization for Google Summer of Code 2011

Chapter 1

Filesystem interface in HelenOS

HelenOS is a microkernel multiserver operating system and as such, large amount of system services is provided by user-space servers. The filesystem support is no exception. There are multiple components working together to enable client applications to work with files. These are distributed among several processes that communicate using HelenOS inter-process communication (IPC) subsystem. The IPC allows processes in HelenOS to exchange short messages (few processor registers) or to negotiate sharing or copying a portion of memory, if it is necessary to communicate larger chunks of data.

Figure 1.1: Processes participating in HelenOS filesystem layer



The processes may be divided into three groups. A client is the application that wants to use filesystem services to execute common tasks as reading and writing of files or reading directories. The application usually uses standard C library, which abstracts the low-level interface from the program itself. Central to the filesystem support in HelenOS is virtual filesystem server (VFS), it handles all client application requests, forwarding them to appropriate filesystem server if necessary. A third group, the endpoint filestem servers, provide implementations of filesystems. The endpoint filesystem servers mostly serve requests made by VFS, but may forward the request to another filesystem when it is necessary[1].

[1]when lookup requests cross filesystem boundaries at mount points

When the distinct parties in the filesystem layer need to refer to filesystem node, they use a node identifier, which is called a VFS triplet. The first part is a filesystem type identifier, or a filesystem handle, which VFS uses to uniquely identify a filesystem implementation (i.e. endpoint server). As there may be multiple instances of a sigle filesystem type handled by a endpoint server, the middle part of the triples identifies device handle the filesystem uses to get data from (a filesystem cannot be mounted multiple times using the same device). Last part of the VFS triplet is entirely provided by the endpoint filesystem server. This identifier is unique within a filesystem instance[2] and must stay the same while the filesystem is mounted[1].

1.1 C library support

The standard library implements convenient application programming interface that wraps low-level operations that the virtual filesystem(VFS) server provides through IPC. The C library API resembles that of POSIX, but it is not exactly the same. HelenOS uses slightly different error codes. Moreover, while HelenOS' filesystem API is currently quite usable, it does not provide some advanced features commonly present in other operating systems.

The standard library also implements high-level handling of path operations. This includes managing task's current working directory, exposed as `getcwd` and `chdir` functions, which is a common feature of operating systems. Additonally, the library is solely responsible for translating any relative paths used in a program to absolute paths before forwarding any request to the virtual filesystem server. This is because VFS operates on absolute paths, which allows its design, and that of endpoint filesystem servers, to be simpler[1].

The handling of absolute paths in the standard library has several implications. First of all, as a result of this design decision, filesystems in HelenOS do not report special directories `.` (dot) and `..` (dot-dot) in directory listings. This simplifies[3] client application code that traverses directory structure — it simply does not need to ignore such nodes. Secondly, when symbolic links will be implemented in HelenOS, the path absolutization process in standard library will probably affect which directory will be used if a `..` (dot-dot) component is present in a path after symbolic link. For example, if we take a symbolic link named `/a/link` which points to directory `/b/c` and we try to access path `/a/link/..`, we will get different results in HelenOS as opposed to POSIX compatible operating systems. The used path will be `/a` in HelenOS, whereas it shall resolve to `/b` according to POSIX standard.

Last but not least, the C library (in cooperation with HelenOS program loader) provides support for standard input and output streams. An application that wants to execute a new program may pass a list of VFS triplets to loader, which then passes it to libc in the new program. The standard library then opens the files the VFS triplets represent. This has a disadvantage that it is not possible to redirect standard output of a program in a way that the output is appended to the end of a file yet (As the intial position in the file after it is opened by libc is always zero). Currently, it is not possible to transfer full file descriptors between processes[4] either.

[2]which is identified by the first two values of the triplet

[3]although the difference may be barely noticeable

[4]which would solve the problem above

1.2 Virtual filesystem server

Virtual filesystem server (VFS) is a key component in HelenOS' filesystem support layer. It is a single service that a client application needs to be aware of to work with the filesystem. It not only routes all client requests to endpoint filesystem server, but also implements some operations that are common to all filesystems.

VFS provides a naming service that keeps track of all registered filesystem types. For each filesystem type, it keeps a phone to the process that registered itself to handle it. Typically, one endpoint filesystem server handles only a single type of a filesystem.

The server is also responsible for managing file handles, which clients use to refer to open files. For each such open file, the VFS keeps an associated state, such as size and current seek position. This enables VFS to execute certain operations, such as seek, without the need to comunnicate with an endpoint filesystem server[1].

Like other operating systems, HelenOS supports the concept of mounting. There is one hierarchical filesystem namespace, where filesystems may be mounted on existing directory nodes. The virtual filesystem server itself handles a special mount point `/`. It is the only mountpoint handled by VFS. Mounts over filesystem nodes are handled by endpoint filesystem servers, so if the mount point is not `/`, the VFS delegates the action to the respective filesystem for further processing. The virtual filesystem server accomplishes this by looking-up[5] the node corresponding to the mount point and forwards the request to the filesystem this node belongs to. Lookups and processing of mounting in endpoint filesystem servers is described in section 1.3.

Yet other feature delivered by VFS is support for locking of nodes. For each open node, the server keeps an internal read-write lock to prevent concurrent access to a given node[6]. This feature moves common functionality of locking nodes from endpoint filesystem servers to the VFS, which is useful mainly in simpler filesystem implementations. In fact, there are cases when this behaviour is not wanted, such as when the filesystem needs to block a client until another client performs an action with the node. The author discovered one example when he initially wanted to familiarize himself with the HelenOS filesystem layer by writing a filesystem that provides pipes[7]. Another difference of a pipe vs. regular file is that for pipes, the size and seek position has no useful meaning and therefore the server should not increment the file size on writes. The limitations has since been fixed by core HelenOS developers by allowing a filesystem to specify such behaviour in its filesystem registration structure (`vfs_info_t`).

The VFS also manages a pathname lookup buffer (PLB). This is a cyclic buffer used to store path information to be used by the endpoint filesystem servers. It is shared as read-only memory to the filesystem drivers, therefore the VFS has ultimate control of its contents. Whenever a lookup request is to be made, the VFS reserves space in the buffer and stores a canonical path[8] in the newly claimed space. This buffer is used to avoid copying the path among VFS and multiple endpoint servers as the lookup operation may cross multiple mount points[1].

[5]the lookup request is forwarded to the filesystem mounted at the root mount point

[6]concurrent access to different filesystem nodes is still allowed

[7]which he did not finish as that wasn't the goal of the thesis

[8]canonical path does not contain dot and dot-dot components or two consecutive slashes

1.3 Filesystem drivers

Filesystem drivers for a concrete filesystem are implemented as separate server processes. Each filesystem driver waits for, and serves, requests from VFS, which uses IPC protocol specially designed for this purpose (VFS out protocol)[1]. Client applications do not connect directly to the endpoint filesystem servers.

Usually upon startup, the endpoint filesystem server register itself in VFS by providing an instance of `vfs_info_t` structure. At that point, the filesystem should start listening for requests from VFS. A client may then request VFS to mount the filesystem using a filesystem identifier, optional device handle and mount options.

When a filesystem is being mounted on a node in another filesystem (or directly to the filesystem root), it receives the `VFS_OUT_MOUNTED` IPC message. After sending the `VFS_OUT_MOUNTED` message, the mounter initiates a data transfer of mount options string. The filesystem server should then attempt to initialize the filesystem using the given mount options. In case of disk-based filesystems, this involves reading the filesystem superblock, or its equivalent, from block device identified by device mapper handle provided as the first IPC argument. The server then either responds with an IPC answer containing an error code, or a three-argument `EOK`[9] IPC answer that contains, in order, index, size and link count of the root directory of the filesystem.

While the `VFS_OUT_MOUNTED` is sent when the filesystem is being mounted, a pair message, `VFS_OUT_UNMOUNTED`, is sent when the filesystem should unmount. This message, just like all others in VFS out protocol, contains a device handle to identify the filesystem instance to be unmounted. The filesystem driver should finalize all filesystem state and unmount the filesystem, with result of the operation indicated by a no-argument IPC reply. Although VFS keeps track of open nodes, and responds with `EBUSY` right away if there are any, the filesystem server may encounter a different error, that needs to be reported back to the client.

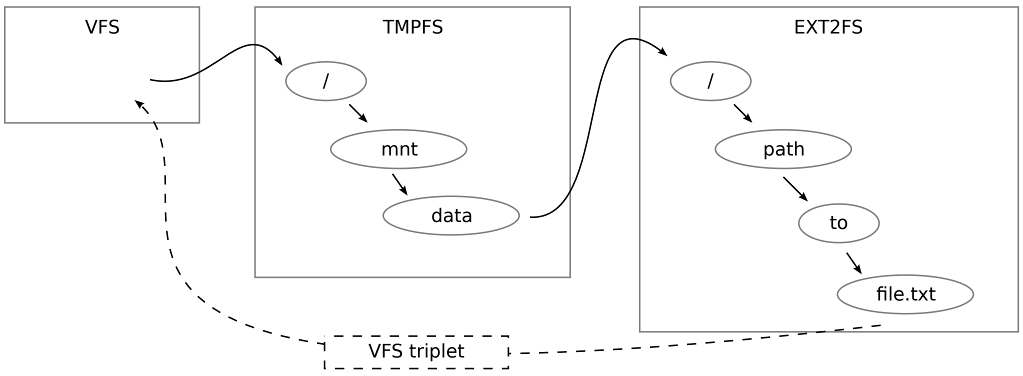
As already mentioned above, endpoint filesystem servers are also responsible for mounting other filesystems on existing filesystem nodes. Whenever a filesystem server receives the `VFS_OUT_MOUNT` message, it should mount another filesystem on a node indicated by the message arguments. The first and second arguments of the message define a mount point[10], using device handle and index. The second pair of arguments define the filesystem to be mounted. This is the mountee filesystem handle, as allocated during filesystem registration, and device handle of the device to be mounted. After the `VFS_OUT_MOUNT` message, a connection to the mountee filesystem is sent to the mounter filesystem using a `IPC_M_CONNECTION_CLONE` message, which is a special system message and causes a connection to be cloned for use in the receiver task. After the mounter acknowledges the connection, the VFS initiates a data transfer of string containing mount options. When all this communication is done, the mounter filesystem has enough information to prepare for the target node and send `VFS_OUT_MOUNTED` message to the mountee filesystem server. Upon receiving answer from the mountee filesystem, the filesystem takes any necessary steps to update its state and itself responds to VFS by forwarding the answer.

Just like `VFS_OUT_MOUNTED`, the `VFS_OUT_MOUNT` message has its counterpart used while unmounting. The messsage is called `VFS_OUT_UNMOUNT` and instructs the endpoint filesystem server to unmount a filesystem from the mount point specified in the message. The mount point is specified as a node usually is, i.e. device handle in argument 1 and

[9]EOK represents no error

[10]the node in the mounter filesystem

Figure 1.2: Example path name lookup of */mnt/data/path/to/file.txt*



node index in argument 2 of the message. All other information the filesystem needs to unmount the mount point, such as mountee phone, was communicated when the mount point was estabilished (and the filesystem needs to store the information also for lookup operations to work). The filesystem just needs to request the mountee to unmount using `VFS_OUT_UNMOUNTED` message and update its state to free the mount point.

Another operation that the endpoint filesystem server needs to implement is translation from file names to node indices. As the file name, when delivered to VFS, may contain multiple mountpoints along the path, it may be necessary to bubble the request through multiple endpoint filesystem servers. When the endpoint filesystem server receives the `VFS_OUT_LOOKUP` IPC message, it should lookup a node corresponding to a path in portion of the pathname lookup buffer[11] (PLB) indicated by message arguments[1]. This is the message with most arguments in the VFS out protocol:

1. index of the first unresolved character in the PLB
2. index of the last unresolved character in the PLB (inclusive)
3. device handle that identifies the filesystem instance
4. lookup flags
5. index of the old node when linking

The behaviour of this operation might be controlled by lookup flags. Caller may specify that the lookup only targets regular files (`L_FILE`), directories (`L_DIRECTORY`), root directory (`L_ROOT`), or mount points (`L_MP`). When the server finds a mount point, it usually crosses it by forwarding the lookup request (with adjusted path start position) to the mountee filesystem server. When the `L_MP` flag is set, it the mount point is not crossed if it is the last component of the PLB path. This is useful mainly during unmounting, when the VFS needs to identify the node the mount point is created over. Additionally, if the flags contain `L_CREATE` flag, the filesystem should create the target node. If, in addition to `L_CREATE`, the `L_EXCLUSIVE` flag is used, the operation will fail if the file already exists (creation of directories always fails if the target exists). `L_LINK` should cause the filesystem to link the node identified by the fifth argument to a node. Additonally, while performing

[11]described in the section about VFS

operations itself, the VFS may set additional flags unavailable to clients, such as L_UNLINK to remove a leaf node or L_OPEN to indicate opening operation.

A message called VFS_OUT_READ is used to read data from the endpoint filesystem server. The arguments contain a device handle, node index, and low and high part of position. The position is split between third and fourth argument as it is 64-bit number and indicates the position a file/directory to read from. Just after sending VFS_OUT_READ message, the client initiates a data transfer which is used to return the nodes contents. In case of files, this is the raw data of the file, starting at the file offset given by position. In case of directories, the meaning of the position value is not that precisely defined, although it starts at 0 for the first directory entry and the operation returns a name of directory entry. The VFS then advances the file handle's position according to the number returned in the first argument of the reply to the original VFS_OUT_READ.

Writing is done similarily, using the VFS_OUT_WRITE IPC message. This differs from reading in that the node must be a file and the data transfer has the opposite direction. The answer has, in addition to bytes to advance, a second argument that stores the new size of the file. Otherwise, semantics of the message are similar to the VFS_OUT_READ message.

A file may be truncated by sending the VFS_OUT_TRUNCATE message. This also has the same arguments as the VFS_OUT_READ message. The same message may also enlarge the file if the position is beyond the end of file. The answer to this message does not have any arguments.

Additional information about a node may be obtained using a message called VFS_OUT_STAT. It has only two arguments — the device handle and node index. The VFS uses this message to retrieve a stat structure, containing metadata such as link count, type of node (file/directory) or size.

The VFS sends a VFS_OUT_OPEN_NODE and VFS_OUT_CLOSE messages when nodes are opened and closed, respectively, to allow the filesystem to maintain state for open nodes. Both messages take device handle and node index as arguments. The open message returns low and high parts of size, the link count and node type.

The filesystem receives a VFS_OUT_DESTROY when a node should be removed from the filesystem. As is common, the arguments are device handle and node index. At this point, there should not be any references to the node, so the filesystem is expected to free the node's on-disk structures.

Last but not least, the filesystem driver may be requested to synchronize changes of a node to disk using a VFS_OUT_STAT IPC message. This message too contains an identification of the node passed as the device handle and node index.

1.4 libfs

As the filesystem servers are responsible for operations that don't differ much among filesystems, a library that abstracts this common code exists. This enables code of the filesystem driver to focus on actual implementation of the filesystem specific behaviour instead of reinventing the wheel[1].

The library provides a skeleton implementation of several common operations. It provides simple interface to register a filesystem with VFS. The filesystem server needs to provide a VFS phone, vfs_info_t structure and connection handler function and the libfs will do the rest, returning a filesystem handle.

A higher level abstraction is provided by the library. When a filesystem provides implementation of libfs operations, stored in a libfs_ops_t structure, it is possible to call libfs to entirely respond to some of the messages of the VFS out protocol. This includes complex operations such as `VFS_OUT_LOOKUP`[1].

The library represents each filesystem node by a `fs_node_t` structure. The implementation needs to provide libfs operations such as retrieving a root node, retrieving nodes by index, releasing a reference to a node, or searching for a node in a directory by name[1].

It is worth noting that if the operations to get node references are called multiple times for a single node, the returned pointers to the node must be the same so long as the reference count of the node is still non-zero. This is necessary to allow libfs to keep state for referenced nodes, such as information about mount points.

Chapter 2

Ext2 filesystem

The second extended filesystem was designed for Linux, an operating system that aimed to implement unix-like interface. In this spirit, it is possible to store special objects such as devices, FIFOs, sockets among files. Directories in the second extended filesystem may be viewed as special types of files too[4].

One goal that creators of ext2 had in mind when designing the filesystem was that it should be extensible[2]. This means that multiple revisions of the filesystem are to some extent compatible. The filesystem itself contains information about revision used to format the filesystem, expressed as a major and minor revision number.

Starting with revision 1 of the filesystem, additional compatibility features were added to the filesystem. The superblock contains three 32-bit flag fields to signal the features used in the filesystem. Each field corresponds to a different set of features[4]:

- Features that are backward compatible even for writing. The implementations are free to modify the filesystem even when such a feature is not supported and it is guaranteed that they do not break the metadata
- Features that are read-only backward compatible. The implementations are free mount the filesystem read-only, but writing to the filesystem would damage the metadata. Examples of features in this category are the sparse superblock feature.
- Features that are incompatible. If the implementation does not support the features, it should refuse mounting the filesystem. Examples of incompatible features include file compression.

First revision[1] of the second extended filesystem had fixed inode size and index of the first ordinary inode. This changed in revision 1, where those values are stored in a superblock. The presence of this information in superblock allows to extend the filesystem structures even further.

Another feature introduced in revision 1 of the ext2 filesystem is a possibility to reduce number of backup superblocks stored in the filesystem. With revision 0, a copy of the superblock was stored in every block group and led to unnecessarily high redundancy for the superblock. The issue was more apparent on larger filesystems, where the number of block groups was higher. The revision 1 thus optionally allows the superblock backup copies to be stored only in block groups 0, 1 and powers of 3, 5 and 7. This is the sparse superblock feature[3].

[1]revision with major number 0

The superblock contains a field indicating the operating system that created the filesystem. There are minor variations of the filesystem across different operating systems that use ext2. For instance, in Linux and GNU Hurd variants, certain fields in inode structure use more space than was initially used in earlier versions of the filesystem. Such fields include for example uid number or gid number. GNU Hurd uses extended mode field as well as additional field used to indicate inode number of a program that is to interpret the contents of the inode[3].

2.1 Physical layout

The physical disk space occupied by the ext2 filesystem is divided into blocks. All blocks in a filesystem are of equal size, which is determined when formatting the filesystem and must be 1024 bytes multiplied by a specified power of two. The block size[2] is stored in a superblock structure described below. Most common block sizes are 1KiB, 2KiB, 4Kib and 8KiB, which is caused by Linux supporting block sizes up to memory page size[3].

The filesystem uses fixed byte order for on-disk storage of all its fields. The byte order is always little endian (or least significant byte first) — this probably originated from Linux being initally developed on x86 architecture, which uses the same byte-order.

Basic filesystem metadata is stored in a superblock structure, located at offset 1024 bytes from the beginning of the underlying block device[3] and spans 1024 subsequent bytes. The superblock is replicated several times as it holds important data. The structure always fits inside single filesystem block as minimum block size for ext2 filesystem is 1024 bytes.

The superblock contains various filesystem information in a large amount of fields. There is a 16-bit magic field at offset of 56 bytes from the beginning of the superblock[4], that identifies the filesystem. This field always contains a value of 0xEF53 in a valid ext2 filesystem. The superblock also holds information about various filesystem structures, mainly counts and sizes of blocks, block groups, inodes and fragments[5]. Information about state of the filesystem is also stored in the superblock. This includes information whether the filesystem was cleanly unmounted, how many times the filesystem was mounted, when was it last mounted or when was it last written.

Blocks are further grouped into block groups, so that various filesystem metadata is spread accross whole disk. Metainformation about block groups is stored in a block group descriptor table, which resides in blocks following the superblock [6]. It is an array of block group descriptor structures. A block group descriptor is 32 bytes long structure, with fields as described in table 2.1, with the rest reserved for future use.

A block group itself consists of optional backup of superblock and block group descriptor table[7], followed by always present block bitmap, inode bitmap, inode table and data blocks. The size of each component depends on count of inodes per block group and blocks per block group, that may be determined by reading the superblock[4].

Block and inode bitmaps represent which blocks or inodes (referred to collectively as item in the following sentences), respectively, of the block group are used. Used item is represented by a bit of value 1 and unused as 0. The least significant bit of byte

[2]or, more exacly, only the number of bits to shift the number 1024 to the left

[3]thus residing in the first or second block

[4]thus 1080 bytes from the start of the block device

[5]fragments are portions of blocks and are currently the same size as blocks in Linux

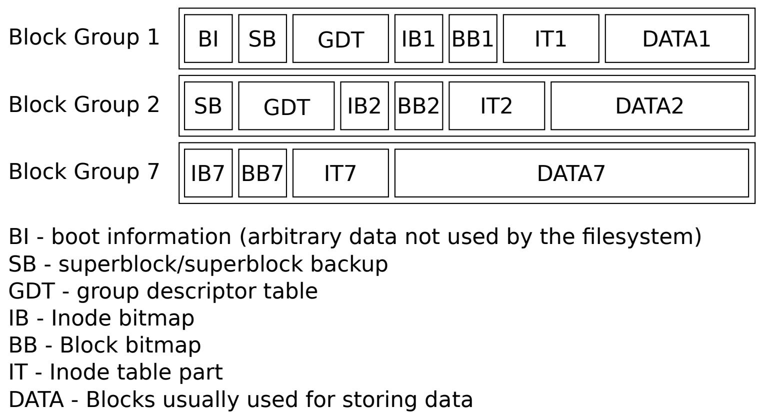
[6]exact amount of blocks dedicated to block group descriptor table depends on block size and number of block group

[7]if the filesystem is marked as using sparse superblock feature, the backup is present only in few selected block groups, otherwise it is present in every block group

Table 2.1: Structure of the block group descriptor

32 bit	Block number where block bitmap of this block group starts
32 bit	Block number where inode bitmap of this block group starts
32 bit	Block number where inode table of this block group starts
16 bit	Count of free blocks in this block group
16 bit	Count of free inodes in this block group
16 bit	Count of inodes allocated to directories

Figure 2.1: Example physical layout of the second extended filesystem


Block Group 1
BI
SB
GDT
IB1
BB1
IT1
DATA1
Block Group 2
SB
GDT
IB2
BB2
IT2
DATA2
Block Group 7
IB7
BB7
IT7
DATA7
BI - boot information (arbitrary data not used by the filesystem)
SB - superblock/superblock backup
GDT - group descriptor table
IB - Inode bitmap
BB - Block bitmap
IT - Inode table part
DATA - Blocks usually used for storing data

1 corresponds to item 1, while the most significat bit of byte 1 corresponds to item 8. Other bytes in the bitmap work the same, with byte 2 covering items 9-17 and so on.

The block bitmap covers all blocks of the block group. Special blocks reserved for superblock and block group descriptor table backup (if present), inode and block bitmaps and inode table itself, are marked as used when the filesystem is formatted. This ensures that additional data structures may be added to block groups in future, as new data blocks are allocated after checking the block bitmap for free blocks.

Inode table is split accross all block groups evenly. There is a fixed amount of inodes per block group specified in the superblock and this determines the size of the inode table portion stored in each block group. With inode having a fixed size, it is not hard to locate a given inode.

1. Determine in which block group the inode is stored. As inode indexes are stored with base 1, the corresponding block group number may be computed as

$$\text{block group} := \left\lfloor \frac{\text{inode number} - 1}{\text{inodes per group}} \right\rfloor$$

2. Lookup the corresponding block group's descriptor and determine start of this portion of inode table (local start)

3. Determine the index within this portion of inode table as

$$\text{local index} := ((\text{inode number} - 1)\ \mathit{mod}\ \text{inodes per group})$$

4. Determine the block number and offset within block

$$\begin{aligned} \text{block number} &:= \text{local start} + \left\lfloor \frac{\text{local index} * \text{inode size}}{\text{block size}} \right\rfloor \\ \text{offset} &:= (\text{local index} * \text{inode size})\ \mathit{mod}\ \text{block size} \end{aligned}$$

Last, but not least, the block group contains the most significant part, the one that is the reason to actually have a filesystem - data blocks. Remaining blocks after inode table are usually used for data, but the filesystem structures theoretically allow for data block to be any block from a block group. The only assumption that a filesystem implementation needs is that every block of the block group that is marked as free may be used to store new data. As special blocks are marked as used, and are never deallocated, the filesystem is very extensible.

2.2 Logical layout

The filesystem is a tree hierarchy of filesystem objects such as files and directories. Every filesystem object is represented by a data structure called inode[8]. An inode contains metadata about the object, but does not itself contain information about its location (path). File names in ext2 filesystem are stored only in directory entries. Metadata of an inode include information such as its type and access rights, owner and group number, different access times stored as a UNIX timestamp[9] and location of data blocks[2].

The second extended filesystem uses a special preallocated inode number for root directory, as well as other special inodes. For example, there is an inode to which bad blocks found during filesystem check are assigned. This prevents their use by the filesystem for storing data. Another special directory present in the ext2 filesystem is the undelete directory, referenced by root directory entry as `lost+found`. It is also used by filesystem check programs to store references to inodes that are not accessible from the filesystem (e.g. when a power failure occurs when a file is still in use, but deleted from the directory tree). These special inodes have fixed inode numbers[2]. List of some of these special inodes may be found in table 2.2.

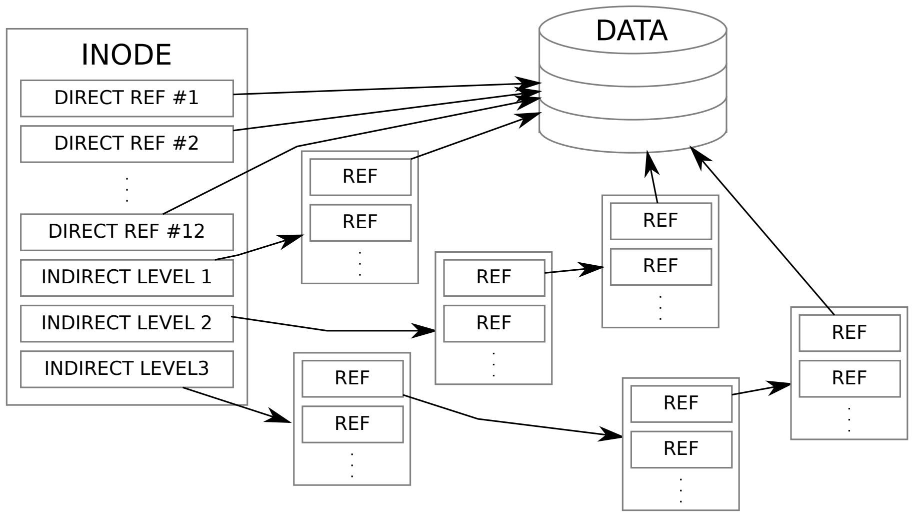
Table 2.2: Special inode numbers

1	Inode for bad blocks
2	Root directory
5	Boot loader inode
6	Undelete directory (/lost+found)

For each inode containing data[10], the filesystem maintains a list of data blocks allocated for its contents. The layout chosen in ext2 was designed to minimize the number of block reads when reading the file, however, for very large files a large amount of metadata is written[11]. The inode contains references to the first 12 data blocks, if present, embedded directly in the inode structure[3]. Additional data blocks are referenced from blocks containing lists of block references. There are three levels of these references:

[8]the reason of why these are called inodes is historical, probably originated from the term ”index node”
[9]number of seconds since January 1st 1970 00:00
[10]there are some inode types that don’t require data blocks
[11]this is solved in ext4 by its extents feature

Figure 2.2: Direct and indirect data block references



- Singly-indirect block references. The inode structure itself references a block that contains a list of references to data blocks.
- Doubly-indirect block references. The inode references a block that lists singly-indirect block references
- Thirdly-indirect block references. Last data block reference in inode is a reference to a block containing doubly-indirect block references.s

All those references together form an array of data block references for the object's contents.

2.3 Files

The primary purpose of regular files is to hold user data, therefore data stored in this object's data blocks corresponds to data used by client applications. A file inode data may be longer as any other inode type. This is because file inodes do have 64-bit length with high 32-bits stored in a place of directory extended attribute block pointer. Moreover, a file can contain "holes", or regions with unallocated data blocks. Such a file is said to be sparse. While implementations do not need to do it explicitly, a sparse file may be detected by consulting properties of the associated inode — its number of currently allocated data blocks is lower than the number of blocks required to hold fully populated file of a specified size.

Unallocated block in a sparse file is marked at corresponding index in the block array as a reference to block with number 0 (zero)[12]. The unallocated block reference may be present in direct blocks as well as any indirection level, therefore it is possible to have large holes in files quite efficiently. When an implementation encounters unallocated block, it should treat it as if it was wholly filled with zeros[5].

[12]the actual block 0 on the block device contains bootloader data and/or superblock, depending on the size of block, so it cannot be directly used as a source of data

2.4 Directories

Directory objects contain a list of references to other filesystem objects and are the only type of objects which define filesystem object (file) names. Each object reference in a directory is labelled with a name that is unique within that directory. Directories may be nested to form a hierarchical namespace that uniquely identifies filesystem objects by name. However, this does not work the other way around as single object may either be referenced from more distinct directories[13] or may not be referenced at all. An example of the latter are orphan files (i.e. files already deleted from a directory but still open by an application).

Unlike file, a directory cannot usually be hard-linked, so that the structure of the directory namespace forms a tree[2]. The directed graph where vertices are directories and edges connect a parent directory with all its children directories, however, is not a tree as there are implicit directory hard links present in every directory. These are the `.` (dot) and `..` (dot-dot) directory entries that point to the current and parent directory, respectively.

The filesystem uses directory entries to map a name to an object reference. The filesystem supports long file names so it is not practical to use constant-length directory entries. Instead, the name may be of variable length and every directory entry has fields that contain its size in bytes as well as size of the name field of the entry[3]. This way, the filesystem can hold long file names while the space in directory contents can be used more efficiently when using short file names.

The structure is slightly different for revision 1 of the filesystem and later versions, where the 16-bit length of the name is replaced by two 8 bit fields - 8 bit length of the name and 8-bit referenced inode type[14].

Table 2.3: Structure of linked list directory entry

32 bit	Referenced inode number
16 bit	Size of the entry (including padding to the next entry)
8 bit	Length of name in bytes (16 bits in rev. 0)
8 bit	Referenced inode type (only in rev. 1)
variable	Name of the object

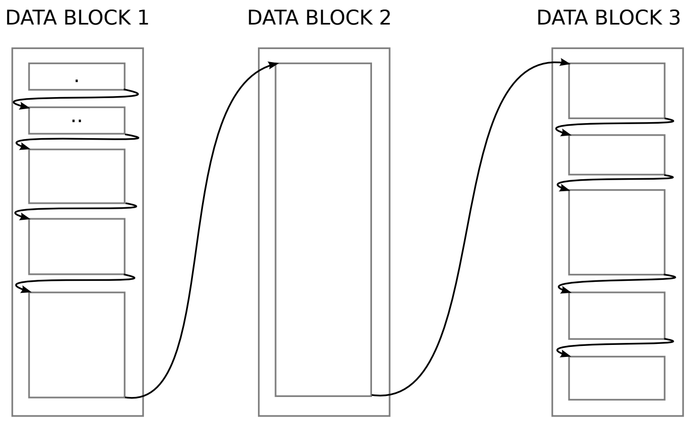
Directory entries are stored in chunks within the data blocks. Every data block holds at least one linked list directory entry structure and may contain multiple directory entries if the space in the block permits. The first directory entry structure starts at the beginning of the data block. Additional directory entries, if present, are located after the first entry and must be aligned at the 4 byte boundary. As the filesystem implementation may easily skip over individual directory entries by reading the size of the current directory entry and advancing the data pointer by a value it just discovered, the structure forms a singly linked list.

While reading a directory, the implementation must know when a given entry is last in the block, so it does not read bogus data. This is achieved by defining that the last directory entry in a block ends at the block's boundary (the size of the entry is enlarged

[13]e.g. in case of hard-linked files

[14]Most implementations limit filename to 255 bytes anyway, so the field was reused to store file type for better performace

Figure 2.3: Example linked list directory



if necessary). The definition above also ensures that a single directory entry cannot span multiple data blocks. This enables simpler implementations as the implementation only needs to work with a single data block at once.

In a linked list, items are usually deleted by adjusting the pointer to the next entry of the item that precedes the one being deleted to point to node after the one being deleted. The linked list of directory entries is no exception. Since the entries are aligned specially at the boundaries of data blocks, there is a corner case if the first entry needs to be removed. While it is possible to relocate a second entry in the block to the beginning, it does not work if there is only one directory entry in a block. Instead of relocating a node, the filesystem allows the referenced inode number in a directory entry in a block to be set to zero, indicating an unused entry.

With large number of directories, performance of linked list directories decreases significantly[3]. Linux developers therefore added a mechanism known as indexed directory entries to overcome this problem. In this case, a hash tree index is added into directory files in a backward compatible way. The first two linked list directory entries (for `.` (dot) and `..` (dot-dot) entries) are replaced with an index root structure, or rather, those are embedded in the index root structure, as is starts with those entries. The length of the second entry, however, is adjusted so that the next linked list directory entry starts at the next block boundary. Therefore, there is a space in the first block for indexed directory data (figure 2.4). This area will be skipped by ext2 implementations that do not support this feature[15], therefore enabling backward compatibility of this feature. Following the two "fake" linked list directory entries are other members of the indexed directory root structure[3].

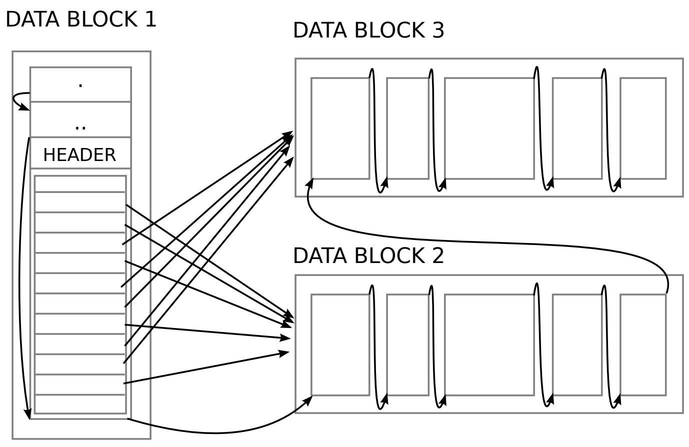
In the remaining space of the first block, that is still within the second "fake" linked list directory entry from the point of view of an implementation that does not support this feature and just after indexed directory root structure an array of indexed directory entries is located. At the beginning of the array, in place of the first indexed directory entry is a stucture with information about the array, that is its maximum and current size. The other elements in the array contain a pair of a hash and block number — the indexed directory entry structure. Those elements are sorted within the array by hash value, to allow faster searching by the value using a binary search algorithm. In case there

[15]due to the length of the second linked list directory entry

Table 2.4: Structure of rest of the indexed directory root

32 bit	Reserved field with value of 0 (zero)
8 bit	Hash type identifier
8 bit	Length of a single indexed directory entry
8 bit	Indirection level
8 bit	Padding

Figure 2.4: Example indexed directory



are more entries that would fit in a single such array[16], indirection levels can be added. In this case, the block number in indexed directory entry points to a block containing another indexed directory entry array[3].

2.5 Symbolic links and other special objects

The second extended filesystem supports the concept of symbolic links. Just as hard links, symbolic links allow to refer to another filesystem object from multiple places, but work in a little different way. A symlink[17] does not contain a reference to an inode directly, it rather stores a path name where the link points to. This allows linking to arbitrary points in filesystem namespace, even in different filesystems[3]. Symbolic links may also point to invalid filesystem objects, such as when the file being referred to does not exist. The downside of this approach is that the link itself requires disk space for the inode. Additionally, symlinks cause a overhead in node lookup, as the node resolution process must be restarted with new path[2].

As most filesystem paths in symbolic links are relatively short, the filesystem employs optimization for storing the links. If the length of the path is less than 60 bytes long, the pathname is stored directly in the inode in place of data block pointers. This avoids allocating a full block for an entity such small as is a symlink. When the path is 60 bytes or longer, ordinary data block allocation scheme is used[4].

[16]as it does not span multiple blocks
[17]symlink is a common abbreviation of symbolic link

The filesystem supports other types of special objects used in Linux such as block or character special devices. These special objects do not use data blocks and similarily to symlinks, use the fields for data blocks to store additional metadata required for those objects to work correctly[4]. These special objects are identified by major and minor numbers which Linux kernel uses to identify device type and unit respectively[2].

2.6 Extended attributes

Sometimes it is useful to store extended file attributes with the file or directory. These are name-value pairs stored in a separate filesystem data block linked from the inode by a field containing the block number. This data block is allocated outside of contents of any file or directory. If multiple files contain the same set of extended attributes, all may reference the same filesystem block. An extended attribute with the same name may not be present in the block multiple times[3].

Extended attribute block starts with a 32 byte header that contains a magic value of 0xEA020000, reference count of this extended attribute block (this is the number of inodes that point to this extended attribute block). Additionally, the header contains the number of blocks used to store extended attributes. Currently, Linux works with only a single extended attribute block for an inode, therefore this value always contains 1. The fourth 32-bit field in the header contains a hash of all attribute entry header hashes[3].

The attributes are stored after the block header, where a list of variable-length attribute entries starts. These entries contain lengths of name and value, block number[18], offset in a block to the attribute value and a hash of name and value. Values of attributes are usually stored starting from the end of the block, growing towards the start[3].

[18]currently always 0 inidicating the current block

Chapter 3

Our implementation

The implementation of support for the second extended filesystem is divided among multiple software components. We decided to split part of the ext2 filesystem support into a software library, so that multiple different programs directly manipulating the filesystem may be easily created. The library, called libext2, is responsible for working with basic filesystem structures and exposing internals of the filesystem in more convenient way. This library is used by several programs that expose the filesystem to the user.

The main product of the thesis, the ext2 filesystem server, is the component that registers itself with the VFS and enables ext2 to be mounted and used from applications in a standard way. Unlike other implementations of the second extended filesystem in other multiserver microkernel operating systems such as MINIX 3 or GNU Hurd, where filesystem implementations are a single unit, our ext2fs server shares code with other programs by using libext2 library. The filesystem driver supports reading sparse files and should work with both revisions of the second extended filesystem as well as variations defined by other operating systems[1].

During developement of the support for the second extended filesystem, we needed to perform some additional tasks. Therefore we wrote additional programs or extended existing ones. A program we developed, called ext2info, displays various filesystem internal structures and data. This program utilizes the libext2 library and reads the data directly from the block device. It was used during development and is useful for debugging. The program is described in section 3.3. Other programs we developed or modified are described in section 3.5.

3.1 Library for ext2 filesystem

The libext2 library forms the lower layer of support of the second extended filesystem in HelenOS. It allows programs to work with internals of the filesystem stored on a block device.

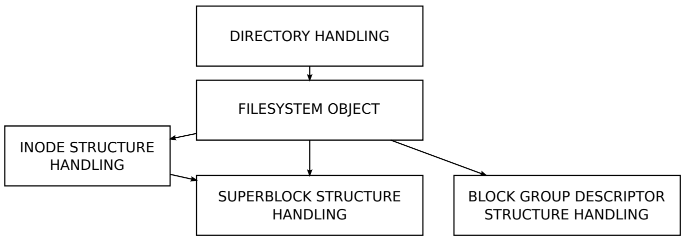
As the second extended filesystem may be viewed as multiple levels of abstraction, we decided to create a library that consists of several layers. This helps to separate concerns of individual pieces of the code.

At the lowest layer, functions to process the superblock structure[2], which is the first structure to be read in order to found basic properties of the filesystem, reside. Because the information contained in the superblock structure contains vital information about

[1] although we did not perform extensive testing

[2] the superblock is described in section 2.1

Figure 3.1: Dependencies of layers within libext2 library


DIRECTORY HANDLING
FILESYSTEM OBJECT
INODE STRUCTURE HANDLING
SUPERBLOCK STRUCTURE HANDLING
BLOCK GROUP DESCRIPTOR STRUCTURE HANDLING

layout of the filesystem, it needs to be handled in a special way — it is the only structure that the implementation reads directly from the device without involving libblock's block cache[3]. This does not impact performance as the superblock is usually read once, at the beginning of the program, and needs to be available whole time the application needs to manipulate the filesystem. It may, however, be wise to consider reloading the superblock using block cache for implementation that supports writing to the filesystem.

In addition to functions for reading the superblock from disk, there are facilities to manipulate in-memory state of the structure. A C language structure is used to read the individual fields of the superblock by name, instead of using precomputed byte offsets. Moreover, contents of this structure are not manipulated directly and are instead accessed using getter functions. We chose this approach as we aim to support multiple revisions of the filesystem and processor architectures the implementation will run on. The getter functions allow us to transparently return values for fields in correct endianness and even compute them on-the-fly if those were unavailable[4] in earlier revisions of the filesystem.

Last but not least, this layer contains function to verify some fields of the superblock, such as magic value or number of fragments vs. number of blocks.

All levels of the library higher than superblock handling functions use a block cache provided by HelenOS' block library. Libblock has support for handling blocks larger than size of blocks of the underlying block device (currently multiples of the block device size), that was added to libblock by HelenOS core team[5]. This allowed us to focus on implementing the functionality of the filesystem instead of writing this support, as the block size may vary among different instances of the second exteded filesystem.

Another group of functions handles block group descriptor table, which is needed to detemine layout of the higher level structures, the most important of which is inode table. These functions are not useful alone, but are used by filesystem functions which provide API abstractions of the filesystem to the users of the library.

These functions provide an abstraction of the filesystem object, as the library needs to handle multiple instances of the second extended filesystem on different devices at the same time. It is a requirement of the endpoint file system server drivers and the support for multiple open filesystems may be useful for example for faster synchronization of two filesystems on different devices[6] when write support will be available. The filesystem instance is represented by a structure called `ext2_filesystem_t`, which keeps track of the state of the library for a filesystem instance.

The filesystem object is usually the first object that the client application obtains

[3]which we cannot initialize at this point because it needs to know filesystem's block size
[4]or rather, were fixed by the specification
[5]in fact added due to cdfs demands shortly before base libext2 code was written
[6]e.g. for backup purposes

from the library by calling `ext2_filesystem_init`, this call also initializes the libblock block cache which is required to read blocks from the device. A corresponding function `ext2_filesystem_fini` closes the filesystem. Several common operations of the ext2 filesystem belong to the filesystem object.

There are functions to check sanity of the filesystem, which verify whether the opened filesystem is not totally bogus and can be read using the library. We chose to split the function that verifies the filesystem from the function that opens the filesystem to accomodate possible use for a program that verifies the filesystem (which could execute more exhaustive checks).

References to block groups and inodes may also be obtained by calling functions of the filesystem object. These references are structures that contain a pointer to a block, obtained from libblock, which holds the referenced structure of interest. Moreover, the reference contains pointer to the referenced structure itself within block's data and in case of inode references, it contains inode number as well. Currently, the references are created as needed and if two references to the same object are requested, two different references are returned. This is not an issue in read-only implementation, but in future, we plan to store reference count in the reference structure itself and make sure the same structure is returned if requested multiple times, so that it may be used to store a lock to protect the referenced object.

Yet other functions in the library handle contents of directories. We decided to provide a directory iterator structure that allows the client application to traverse the underlying linked list structure of the directory data to simplify usage in client programs. As the directory structure stored in the filesystem is a singly linked list, it is only possible to iterate the iterator forward.

3.2 ext2 filesystem server

The filesystem server is the component responsible for integrating ext2 support into HelenOS' filesystem layer. The server is based on libfs library that implements common operations required by VFS. In addition, ext2fs server utlilizes libext2 library to work with the second extended filesystem.

Upon startup, the server registers itself with VFS and starts to listen for requests. Requests are handled in a main loop and appropriate functions are called. This boilerplate code is similar in most filesystem servers in HelenOS and is written by hand. In the future, HelenOS developers plan to add an option to autogenerate this kind of functionality.

Some of the functions handling IPC requests are directly forwarding the request to the libfs library. Those are higher level operations common in most file system drivers[7] that are implemented using lower level functions provided by the filesystem server. The ext2fs driver is not different.

First request the server is likely to encounter is a request to mount the filesystem. Mount operation in ext2fs is quite simple, the server uses libext2 functions to initialize the filesystem. It then just needs to keep information about mounted instances. Information about every single instance of mounted filesystem is kept in a `ext2fs_instance_t` structure. Every instance is identified by a device handle as it is unique in the system - there cannot be two instances of the filesystem mounted from the same device. Moreover, the device handle is used in a VFS triplets and therefore available to all other requests sent to the filesystem server. Along the device handle, the structure keeps a pointer to

[7]and described in chapter 1

libext2's `ext2_filesystem_t` structure for the instance. Additionally, we later added a counter of open nodes for the instance so we can easily check that there are no open nodes when unmounting.

We chose to use simple linked list to hold multiple instances as we do not expect a large number of filesystem instances to be mounted at the same time. The list is protected by a mutex to prevent its corruption. As it is necessary to retrieve the instance by device handle in multiple places, we introduced `ext2fs_instance_get` function, which searches the list by a device handle, while correctly locking the mutex. This enables future expansion if a need arises to change underlying mechanism of storing instances.

As the ext2fs server uses libfs to implement most of its operations, it needs to manage libfs' nodes. Those are created in `ext2fs_node_get_core` function, which returns a node by filesystem instance and inode number. As inode number is unique within the second extended filesystem, it was an obvious choice to select as unique node number to use in VFS triplet. Therefore, any implemented ext2fs operation has easily accessible information to instantiate a node structure.

As open nodes may be referenced from multiple places at the same time, we chose to use hash table from HelenOS' standard library to store references to open nodes. The `open_nodes` hash table is global for the entire ext2fs server and is keyed by device handle and inode number — with the former uniquely identifiying the instance and the latter being unique node identifier. The `ext2fs_node_get_core` function tries to look-up the node in this hash table before instantiating a new node from disk. This satisfies libfs requirement to keep state for nodes and also allows the server to keep runtime information in the node structure.

Searching for a node by name in a directory is implemented in `ext2fs_match` function, which is used by libfs during lookup operations. The function simply walks the directory contents using an iterator provided by libext2, as support for indexed directory entries is not implemented. The file names are currently compared as a binary strings as the driver cannot assume anything about the file name in second extended filesystem[8]. While this should not usually cause any problems, in case the on-disk filename is not a valid UTF-8 string, the user may not be able to access it, because they will be unable to type the name on the keyboard. Moreover, the user-supplied string is zero terminated, therefore any on-disk string containing a null byte shall not match even if the user was able to include null byte in the supplied string[9].

Other operations implemented by ext2fs and used by libfs are getters of various properties the nodes have. The server retrieves such information from inode data accessible directly from `ext2_inode_t` stucture associated with the node. Remaining libfs operations required for write support, such as `ext2fs_create_node`, `ext2fs_destroy_node`, `ext2fs_link` and `ext2fs_unlink` are merely stubs returning `ENOTSUP` error, as out implementations does not support writing.

Operations not covered by libfs include reading (and writing) contents of files and directories. Read operation is among the more complex functions implemented by the filesystem server. As the complexity of `ext2fs_read` function grew over time, we decided to split its core part to two separate functions — `ext2fs_read_file` and `ext2fs_read_directory`. In the end, `ext2fs_read` only carries out actions common to reading files and directories, such as preparing the corresponding filesystem node.

When reading a file, the `ext2fs_read_file` function handles only a single data block at a time to simplify the code. However, this does not impact functionality of

[8]file names may contain any byte except forward slash

[9]the lengths do not match in this case

the filesystem driver as `read` (and `write`) function may process less data than was requested. The resulting amount of data read is returned as part of IPC answer. It is then only a matter of calculating the correct file data block number from file position (which is then resolved to actual filesystem block number by calling libext2 utility function `ext2_filesystem_get_data_block_index`) and copying the data to the buffer which is sent to client. There is a minor complication, however, as there may not be a block allocated for the position we are reading, which means we are dealing with a sparse file. This special case is handled by preparing and returning long-enough buffer of zeros instead of reading any disk block. Initially, our implementation did not support reading sparse files, because we did not possess accurate documentation of the feature. Later, we found a correct description in [5] and used it to add the feature to the implementation.

Reading directories is implemented in `ext2fs_read_directory` using libext2's directory iterator. As the interface to VFS uses index-based access to directory entries, the server needs to emulate this by iterating the linked list from the start. Performance implications of this behaviour and possible solutions are described later in section 3.4. In addition to emulating index-based interface, the server filters-out `.` (dot) and `..` (dot-dot) directory entries from listing of directory contents. While those nodes could be theoretically included in the output of directory listing, we believe there is no advantage in doing that, as VFS already works with canonical paths which do not contain those components.

3.3 Utilities for working with ext2 filesystem

During developement of libext2 a need to test the code arose. While it was possible to directly start implementing ext2fs server, it was less feasible. Creating a separate command-line tool to test the functionality implemented in the library proved to be useful. The tool, named `ext2info` can be used to display various internals of the second extended filesystem. Initially, we wanted to call the tool `ext2debug`, but it didn't work out well — the name is too long to store in the FAT filesystem that is used to boot the system[10].

The tool was built incrementally, as new functionality was added to the libext2 library. Current feature set covers all levels of libext2 library and includes:

- listing most superblock fields and values
- dumping contents of block group descriptor table
- showing information about given inode structure (by i-node number)
- displaying associated block numbers for given inode
- dumping data of inode's data block given by its position in file
- listing contents of directory inodes

Typically, the output is longer than the height of the screen for larger filesystems, as with growing size of the filesystem, the size of some of its internal structures grows too. For instance, there are 79 block groups for a 10GB filesystem with 4 KiB block size. As `ext2info` outputs the data to standard output, it may be redirected to a file using `redir` program provided by HelenOS and later viewed using a program that supports paging and/or scrolling such as `cat` shell builtin or HelenOS text editor.

[10]the command name was clipped to "ext2debu" as the filesystem only supports 8.3 filenames

Additionally, `ext2info` shortens the list of associated block numbers for an inode. This list grows linearly with the file size in case of non-sparse files[11]. Hopefully, it may be easily compressed by folding successive block indices in groups and showing only beginning and ending index of those groups. As the file may be sparse, the command shows the list as a mapping: for each of the filesystem block number, file-based block numbers are output as well. A range of unallocated filesystem blocks folded as a range may thus be unambiguously presented to the user.

3.4 Performance

While implementing the filesystem server and libext2 library, our goal was primarily to implement filesystem features and support for various versions of the filesystem and processor architectures, and performance was of a lower priority. Therefore we decided to measure the performance of the `ext2fs` filesystem server.

Results of benchmark of sequential reading of files show that performance of the `ext2fs` server is acceptable. The amount of time taken to read the file grows linearly with the file size (figure 3.2)[12]. It may be possible to enhance performance for this use-case, for example by reducing the necessary block reads while searching for a given data block by its position in a file. This will require changing the code that handles indirect blocks to store its state in a structure and continue with that state on next invocation. This change depends on ability of the endpoint file system server to associate an iterator with a given client, therefore requiring the VFS to communicate unique file descriptor id to the `ext2fs` server. While this change may enhance performance by not performing block lookups on every read, it may not yield much performace gain as the blocks used to resolve indirect block pointers are already cached in a block cache. Another possible enhancements include instruct the block cache to read a next block in background while the current block is being processed by client or return multiple data blocks at a time.

This set of tests yield another interesting fact. The time taken to read a file of a constant size (using `ext2fs`) increases with the number of attempts to read the file. This is shown in figure 3.3. The reason for this behaviour is currently unknown, but seems to be specific to our implementation as `fat` does not show similar behaviour. This unwanted behaviour is a bug and will need to be resolved.

Reading of directories shows another performance issue[13]. Time of directory read grows quadratically with number of directory entries (figure 3.4). This is caused by ext2fs using libext2's directory iterator that can currently only be initialized at the beginning of the directory. There are several solutions to this problem. One possible solution is to cache the iterator between requests in `ext2fs`, this will reduce the performance hit for sequential reading of one client, however, as soon as another client starts reading the directory, the performance will drop again. To store the iterator for every client separately, the VFS will need to be extended to expose file descriptor numbers to endpoint filesystem servers. Another possible solution is to use position value as byte offset into directory data and initialize directory iterator with that information [14]. This last solution has an advantage that it does not require changing the interface between endpoint filesystem servers and VFS, while it will work with multiple clients simultaneously reading the directory contents.

[11]for a definition of sparse file, see chapter 2

[12]the graph is distorted a little due to phenomenon described in the next paragraph

[13]already mentioned in section 3.2

[14]fat uses similar approach with storing dentry index in position value

Figure 3.2: Average sequential file read times; 5 test runs per point, logarithmic scale

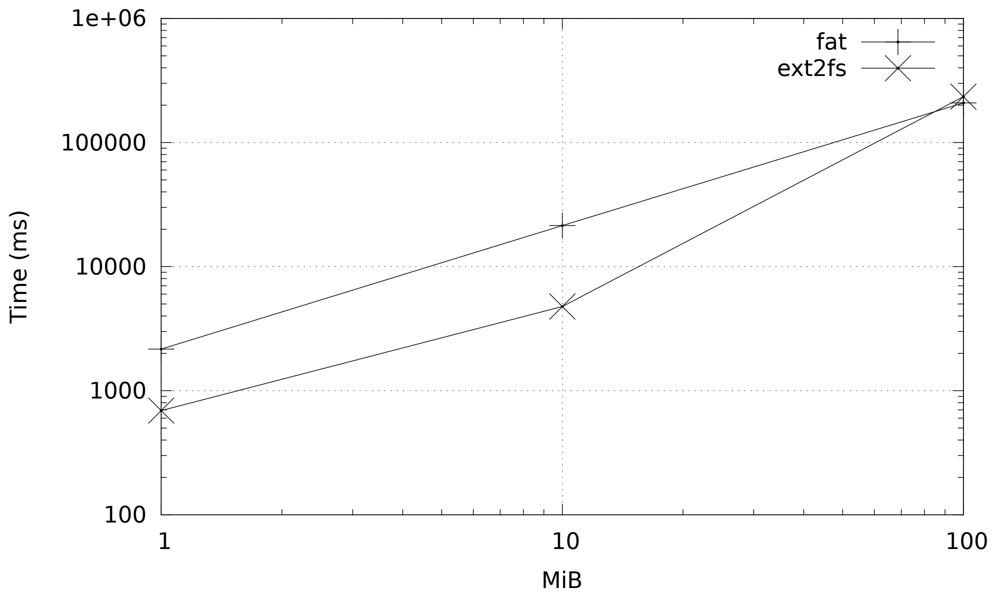


Figure 3.3: Repeated sequential file read times

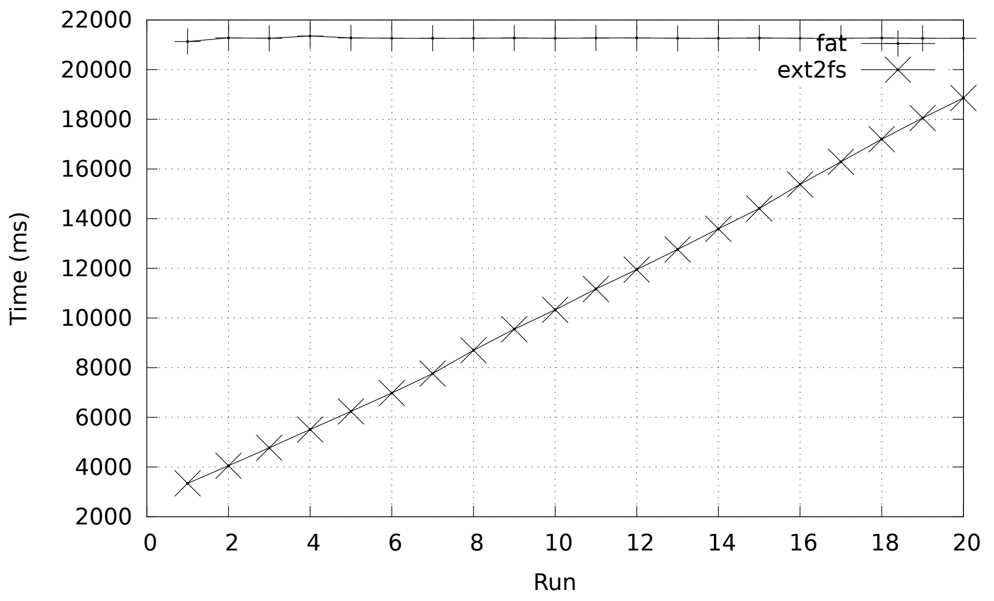
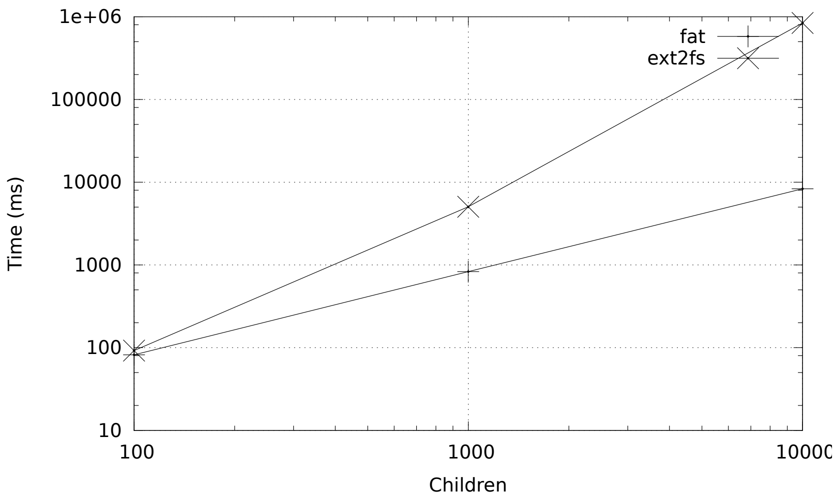


Figure 3.4: Average sequential directory read times; 5 test runs per point, logarithmic scale



3.5 Other enhancements

As we worked on the thesis, several other programs were written or enhanced. First of all, a program, `blkread`, to read and display block contents from block devices was written. Although similar functionality was already present in the system, implemened as a shell builtin command `bdd`, we proceeded to write a standalone program. This has the advantage that output of such program may be easily redirected to a file (output of shell builtins currently cannot be redirected). Moreover, the author learned how to work with libblock library.

While reading the superblock of the second extended filesystem, we needed to read a continuous array of bytes from disk directly, without the block cache being initialized. While the libblock library provides a function to read blocks directly, the physical block size of a device may vary. Therefore, we added `block_read_bytes_direct` function that allows reading portion of the block device using byte address instead.

During developement of the filesystem support, it was necessary to verify that reading works correctly even for large files. As this will take too much time to do manually for even relatively small files, we created a pair of programs to check reading of files. The first program (`gentestfile.py`) is written in Python and should be used from Linux. It creates a file containing a known pattern, as this allows to detect the errors better than computing a checksum. As the pattern the file should contain is known in advance, it is possible to detect precise position where the file differs from the expected pattern. We chose a monotonically increasing sequence of 64-bit numbers. A corresponding HelenOS program, `testread`, verifies that contents of the file match the pattern and displays the results. We initially used this program to also benchmark the speed of reading, but eventually, separate tools for benchmarking were written.

The `bnchmark` program runs various filesystem test runs and measures how long it takes to execute them. The command allows to specify the type of test to run, number of tries, note and a test path and outputs results to stdout in CSV format for further processing. Currently, only two tests are supported, but the program is extensible and allows tests to be added easily. The first test reads a file sequentially from start to end using 8 KiB buffer. The other test reads a directory sequentially, using standard C directory functions. For creating a directory with large amount of children, we also created a simple python script, available in `tools\filldir.py`.

While currently not much used, we also added some utility macros to the tester program. These look similar to assert functions used in *Unit type frameworks, but have some limitations. The macros are used in a test of superblock decoding, where their usage significantly shortens required code.

Last but not least, we enhanced `cat` shell builtin. The program was enhanced in two ways. The first enhancement added support for converting the input to hexadecimal output[15]. This is useful for working with binary files, such as sparse files, as cat normally uses strings to display the file. Support for paging the output on screen[16] was also added to `cat`, as we implemented the `-m` option.

[15]although the feature should probably reside in a separate program after support for pipes is added to HelenOS

[16]again a feature that should probably be a concern of a separate program

Conclusion

The thesis aimed to implement read-only support for the second extended filesystem in HelenOS. We implemented several programs to achieve this goal and currently, it is possible to work with the filesystem using standard HelenOS applications. The implementation supports reading files and directories. Moreover, it possible to create additional applications working with the second extended filesystem, as we developed a library for working with internals of the filesystem.

While the implementation has some shortcomings in the performance area, these can be mitigated relatively easily as we proposed possible fixes. Additionally, implementing features such as support for indexed directories may as well enhance performance of the filesystem implementation.

The libext2 supports reading some of the fields of the second extended filesystem that are currently not supported in filesystem layer of HelenOS mainline, such as getting information about currently mounted filesystem (e.g. UUID and volume name, amount of free space). Reading of inode fields such as user and group id or additional information in directory entries (target inode type) is also supported, although it is not exposed via filesystem layer. The filesystem support in HelenOS may therefore be extended to allow these additional capabilities of the second extended filesystem to be used in the future.

While ext2fs supports reading files and directories, the second extended filesystem has additional features that are not supported yet. A feature that will also require changes in the HelenOS VFS is support for symbolic links as these may point to any path in the filesystem namespace. Current ext2fs implementation ignores symbolic links and other special files and does not report them in the directory listings. Currently, libext2 does not support reading symbolic links shorter than 60 characters, as those are stored directly in the inode in place of block references. Although adding a function to read such inodes should be trivial, it will need to be done before symbolic links could be supported.

Another feature of the second extended filesystem which is not implemented yet is support for extended attributes. While this feature is not as useful in read-only implementation, it may be used to store various metadata for files. The support could be useful for security support in HelenOS, developed by Štěpán Henek.

As the implementation currently supports only reading of the filesystem, the next possible step is to implement write support. Currently, the libfs library does not provide any support for locking, which is required for correct write support as various race conditions could lead to data corruption. The current support for structure references such as `ext2_inode_ref_t` and `ext2_block_ref_t` should therefore be extended to return the same instance when it is requested multiple times, in the same way ext2fs currently does for open nodes. This will allow adding locks to these ref structures, which could be used to ensure data consistency. Additionally, some existing facilities of the library will need to be expanded. For example, the code that iterates through data block references in the inode does not currently support changing the value of the referring block. This will be required if blocks are to be appended to, or removed from the file. Similar change will be required for directory support, where the directory iterator should allow inserting or removing directory entries into/from the linked list (and indexed directory structure when it is supported).

Additionally, while not strictly required for the write support, block library may be enhanced to support marking blocks as unused and communicating this information to the block device. The block device may then use a command such as `TRIM` to inform the actual hardware. If supported by the block layer and libext2/ext2fs, the feature could enhance disk lifetime in case of solid-state drives. Additionally, as HelenOS is still mostly developed in a virtualized environment, the support could lower disk usage for virtual hard drives.

In the more distant future, the code for ext2 filesystem could be used as a basis to build support for newer versions of the filesystem, such as ext3 or ext4, which provide more features.

Resumé

Operačné systémy môžeme rozdeliť na niekoľko kategórií podľa toho, koľko kódu vykonávajú v kernel-móde. Najviac privilegovaného kódu beží v monolitických kerneloch, ktoré takto spúšťajú ovládače zariadení a rôzne systémové služby ako napríklad podporu súborových systémov. V mikrokernelových operačných systémoch je to práve naopak, kernel obsahuje len kód potrebný na základnú podporu user-space programov, ako správu vlákien alebo komunikáciu medzi procesmi. Hybridné kernely sa nachádzaju niekde medzi monolitickými a mikrokernelmi.

HelenOS je mikrokernelový multiserverový operačný systém podporúci viac ako poltucet procesorových architektúr. Teda podpora súborových systémov, tak ako iné služby, sú implementované v užívateľskom priestore. Klientské aplikácie v HelenOSe komunikujú prostredníctvom IPC so serverom VFS (virtual filesystem server), ktorý poskytuje podporu pre súborové systémy. Jednak poskytuje klientským aplikáciám podporu pre rôzne vysokoúrovňové objekty ako sú file deskriptory. Dvak slúži ako centrálny správca súborových systémov. Tento server odovzdáva požiadavky na operácie so súbormi príslušným serverom implementujúcim konkrétne súborové systémy. Pred začatím tejto práce obsahoval HelenOS len niekoľko virtuálnych súborových systémov a jeden súborový systém s diskovým formátom — FAT16. Pretože viaceré súborové systémy zdieľajú rovnakú, alebo veľmi podobnú funkcionalitu, HelenOS poskytuje knižnicu implementujúcu spoločné časti[1].

Second extended filesystem (ext2) je súborový systém pôvodne vyvinutý pre potreby operačného systému Linux, podporuje teda unixovú sémantiku a okrem súborov a adresárov umožňuje ukladať aj ďalšie typy objektov, ako sú symbolické odkazy, rúry (pipes), sokety, či znakové a blokové zariadenia. Tento súborový systém je veľmi rozšíriteľný, umožňuje ukladať príznaky použitých vlastností priamo v superbloku súborového systému. To znamená, že implementácie môžu byť písané postupne, vlastnosť za vlastnosťou, bez toho, aby sa ohrozila integrita súborového systému, ktorý používa nejakú nepodporovanú vlastnosť.

Naša implementácia tohto súborového systému sa delí na niekoľko častí. Základnú podporu pre manipuláciu so súborovým systémom poskytuje knižnica *libext2*, ktorá umožňuje ostatným programom načítavať diskové štruktúry do pamäte a pracovať s nimi. Hlavným produktom práce je však samostatný server *ext2fs*, ktorý sa zaregistruje u VFS a poskytuje služby tohto súborového systému tak, aby ich mohli používať aplikácie bežným spôsobom. Tento server využíva spomínanú knižnicu a sám implementuje funkcie vyššej úrovne. Okrem týchto programov sme vyvinuli aj nejaké ďalšie programy užitočné pri hľadaní chýb alebo meraní výkonnosti súborového systému.

Výsledný produkt funguje pomerne dobre. Niektoré drobné chyby, ktoré sme objavili, sa dajú opraviť pomerne jednoducho a v práci sme popísali, ako by sa dali odstrániť. Do budúcna sa dá implementácia rozšíriť napríklad o podporu zápisu, prípadne ďalšie vlastnosti súborového systému ext2, ktoré zatiaľ nie sú implementované.

Bibliography

[1] JERMÁŘ, Jakub. 2008. Implementace souborového systému v operačním systému HelenOS. In *Proceedings of the 32nd EurOpen.CZ Conference*. Available on the internet: `http://www.helenos.org/doc/papers/HelenOS-EurOpen.pdf`

[2] CARD, Rémy - TS'O, Theodore - TWEEDIE, Stephen. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the First Dutch International Symposium on Linux*. ISBN 90-367-0385-9. Available on the internet: `http://e2fsprogs.sourceforge.net/ext2intro.html`

[3] POIRER, Dave. 2002. The Second Extended File System: Internal Layout. [online] Updated February 4th, 2011. Available on the internet: `http://www.nongnu.org/ext2-doc/index.html`

[4] Linux Kernel Documentation: The Second Extended Filesystem. [online] Available on the internet: `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob_plain;f=Documentation/filesystems/ext2.txt;hb=HEAD`

[5] John's spec of the second extended filesystem. [online] Available on the internet: `http://uranus.chrysocome.net/explore2fs/es2fs.htm`

Appendix A

Compiling and running

HelenOS uses Bazaar version control system for source code management. Source code accompanying this thesis was developed and is stored in bazaar branch helenos-ext2, which is based on HelenOS mainline. The source code for helenos-ext2 may be obtained in several ways.

First of all, if you are reading the printed version of the thesis, you should have received a bootable CD, which also contains the source code of the helenos-ext2 branch and PDF version of this thesis. The CD may be inserted into a real or virtual IA-32 (x86) compatible computer and used to boot HelenOS with ext2 support.

Secondly, the tar-gzipped source code and iso image have been submitted as attachments along the thesis.

Thirdly, the same files should be available from the project's web site, where the source code may also be browsed. Please direct your web browser at

```
http://ne.st.dcs.fmph.uniba.sk/projects/helenos-ext2
```

Last but not least, you may download the source code directly from the bazaar repository:

```
bzr branch http://ho.st.dcs.fmph.uniba.sk/~mato/bzr/helenos-ext2
```

Structure of the source code

The source code provided is based on HelenOS mainline and as such follows the structure of the repository. Our work is placed inside respective directories and mostly took place in the `uspace` subtree, where user-space code resides. Additionally, scripts `filldir.py` and `gentestfile.py` useful for preparing test filesystem images are stored under `tools` directory.

Structure of the relevant portions of the `uspace` directory is as follows:

- `app`: userspace client applications
  - `blkdump`: blkdump program for dumping block devices
  - `bnchmark`: program for benchmarking filesystem performance
  - `ext2info`: program for displaying various ext2 internals
  - `testread`: program for checking whether a file was read correctly
- `lib`: userspace libraries

    - `ext2`: libext2 library

- `srv`: userspace servers
    - `ext2fs`: ext2fs filesystem server

Building from source

To build HelenOS from source, you need to install a supported compiler toolchain. HelenOS contains a script to do that for you, it is located at `tools/toolchain.sh`. Please follow the instructions printed by the script.

After the toolchain is set-up, you may either run just `make` (which will display configuration menu when run for the first time), or `make PROFILE=<profile>` (where `<profile>` is a supported profile, e.g. `ia32` for 32 bit PC and `amd64` for 64-bit PC) to instruct the build system to create a bootable CD image `image.iso` in the branch directory. You may then boot the system from the image.

Running

To boot the system from image, either use a system emulator like QEMU or burn it to a real CD and instruct your computer to boot it. Example command for QEMU:

```
qemu -cdrom image.iso -hda /path/to/your/disk/image
```

After booting the system, a shell is presented to the user. Depending on build configuration and the disk you want to use for reading, the ATA driver may or may not have been started automatically. You may find the block devices already available by running `ls /dev/bd` command. If the block device of interest is not present in the output of the above command, please run `ata_bd` to discover primary and/or `ata_bd 2` to discover secondary ATA disks. Block devices created by ata_bd are named like `bd/ata1disk0` (this example is ATA primary master).

If the disk you use does not have partition table and contains the filesystem directly (as may be the case e.g. when running with a filesystem image directly in VM), you may skip this step. Otherwise, run a partition server to discover partitions on the device. If the block device of interest is ATA primary master, run `mbr_part bd/ata1disk0` for MBR partition tables (`g_part` may be used for GUID partition tables). A block device should be created for each partition (in our example, `bd/ata1disk0p0` for the first partition).

When you have the block device set-up, you may proceed to mount the filesystem. In order to mount it, the ext2 filesystem server must be running, so type `ext2fs` in the shell. To mount the filesystem, use the `mount` command (/data should be present in the system, you may also `mkdir` another directory and use that instead):

```
mount ext2fs /data bd/ata1disk0p0
```

From now on, you should be able to browse the filesystem using `ls` and `cat` commands. To unmount the filesystem after use, type `unmount /data`