Conceptual Formulation

Der Zugriff auf Linux-Partitionen im Ext2 oder Ext3 Format ist von Windows XP standardmässig nicht unterstützt. So können dort abgelegte Daten unter Windows weder angeschaut noch geändert werden. Windows XP bietet jedoch die Möglichkeit zusätzliche Dateisystemtreiber zu laden. Ein beschränkt funktionsfähiger Dateisystemtreiber für Ext2 wurde in einer Abschlussarbeit des NDS Software Engineering bereits entwickelt. Davon ausgehend soll ein einfach handhabbarer und robuster Dateisystemtreiber realisiert werden, der neben Ext2 auch das protokollierende Dateisystem Ext3 unterstützt.

Aufgaben

Es ist ein Dateisystemtreiber für Windows XP zu entwickeln, der folgende Eigenschaften aufweist:

- Unterstützung der Dateisystemformate Ext2 und Ext3
- Lese- und Schreibzugriff
- Sinnvolle Umsetzung der Linux Dateisystemrechte auf Windows Dateisystemrechte
- Robustes Betrienbsverhalten v.A. hinsichtlich Beschädigung von Ext2/3 Partitionen und Datenverlust
- Manuelles und automatisches Mounting/Unmountiung von Ext2/3 Partitionen
- Optional: gute Zwischenpufferung für hohen Durchsatz und kurze Reaktionszeiten

Nach Bedarf sind zusäzliche Hilfs- und Testprogramme zu erstellen, die Auskunft geben über den Zustand der eingehängten Ext2/3 Partitionen. Erfasste Parameter könnten sein: Anzahl geschriebener/gelesener Bytes, Anzahl Verzeichniszugriffe, Anzahl Fehler/Wiederholungen, mittlere Reaktionszeit und Durchsatzrate. Testprogramme sollten ein automatisiertes Prüfen von Ext2/3 Partitionen erlauben, um einfach festzustellen ob der Dateisystemtreiber einwandfrei läuft bzw. Probleme mit den entsprechenden Partitionen festgestellt wurden. Für die Entwicklung des Treibers stehen der IFS(Inetrface File System)-Kit von Microsoft, sowie alle benötigten Entwicklertools zur Verfügung.

Als Resultat soll eine demonstrierbare und in der Praxis einsetzbare Software vorliegen, deren Entwurf und Funktionsweise in einem Bericht dokumentiert sind. Die SW soll auf verschiedensten PC's funktionsfähig sein und mit einer Bedienungsanleitung ergänzt werden.

Es besteht die Absicht den entwickelten Treiber gratis interessierten Personen zugänglich zu machen, wobei aber die Entwicklung innerhalb der Lizenzvorgaben von Microsoft zu erfolgen hat.

Technologien

- Festplattenspeicherung auf logischer Ebene
- Datensysteme
- Windows XP Treiber, spez. Dateisystemtreiber

Unterschrift ....

Abstract

Ext2 and Ext3 are the most popular file systems on Linux operating systems. Because of the popularity of Linux, more and more people use this operating system besides Windows. The main problem is that data stored on a Linux partition cannot be accessed by Windows and vice versa. WinLFS is a solution for Windows, namely an additional Windows file system driver which is capable of reading from and writing to a Linux partition, which is formatted with the Ext2 or Ext3 file system.

WinLFS is a Windows file system driver based on a driver previously developed in a continuing studies diploma thesis. It is implemented following the IFS (Installable File System) Guide provided by Microsoft. Additionally a file system recognizer was developed. This recognizer finds any known (known are ext2 and ext3) partition on a disk on first access and loads the appropriate driver. Therefore, once installed, any ext2 or ext3 partition can be uses like any file system supported by Microsoft.

Not all functions could be implemented due the complexity and lack of time during this thesis.

Management Summary

Introduction

Windows is the most popular operating system (os) available today. Most companies and private users use Windows as their primary os. Besides Windows, there are other os's, like Linux, becoming more popular. Although other os's are eligible to use and suits better for certain purposes, most users cannot do without Windows, mostly because of interoperability problems.

This interoperability problems are the main reason why most users still use Windows as their primary and only operating system. WinLFS cannot and is not intended to change this fact but gives users the opportunity to interact directly with the Linux operating system on a single desktop. WinLFS makes it therefore possible to use Windows and Linux a on single machine without having the trouble of not being able to access data stored on Linux. Namely, with WinLFS, Windows users are given the possibility to access and modify any files created on Linux within Windows or create new files on a Linux partition.

The need of such a program like WinLFS arises because there is no built-in support in Windows to access those data. This means, there is no file system driver in Windows which supports file systems used on Linux. WinLFS is such a driver and supports Ext2 and Ext3 file systems.

Starting Position

WinLFS is based on a continuing studies diploma thesis. The main purpose of this continuing studies thesis was to write a read-only driver for Windows which supports ext2 file systems. WinLFS extends this functionality by adding support for writing.

Procedure

First problem was to find material on file system driver development. There are just a few people programming this kind of software and therefore, not much material or documentation is available. Second, file system driver development is very complicated and needs a lot of study prior to programming.

For efficiency reasons, the project was cut into parts which could be developed separately. This idea should work for experienced driver programmer but not for inexperienced ones like the authors of WinLFS. For this reason, WinLFS was created using a technique called extreme programming. This has the great advantage of two people trying to solve the same problem at one time. Extreme programming worked best in this situation and it was more efficient than working on a problem alone.

Because of its complexity, there was no point in trying to get the whole code working at once. For that, the code had do be programmed in little pieces which could be tested and merged if these tests were successful.

Results

As mentioned above, the problem was more complex as thought at the beginning. What was planned is a file system driver which recognizes and handles ext2 and ext3 partitions correctly. Additionally, a file system recognizer and a driver unload function which works on Windows XP was planned.

The file system recognizer and the driver unload function work, the driver itself does not. The available time was not sufficient to implement all required functions.

Outlook

This work was intended to provide a solution for the open source community to share Linux' ext2 and ext3 partitions with Windows. Even though this is a great idea, it is not recommended to finish this work during another semester thesis mainly due to the fact that already two semester thesis have worked on the code and it is unlikely for another team to understand everything which had been done so far in such a short time like one semester.

Contents

List of Figures

List of Tables

Part I

Requirements

This part includes use cases and all requirements (functional and non-functional).

Chapter 1

Vision

1.1 Introduction

This chapter summarizes high level goals and gives a brief product overview.

1.2 Purpose of this thesis

Access to ext2 and ext3 Linux partitions is not provided by Windows. But Windows XP has the possibility to load “installable file system” drivers (IFS).

The goal of this thesis is to develop a driver to access Linux file systems under a Windows XP System. The driver will be developed based on the ext2 read-only driver from a previous thesis. We envision to implement read and write access for ext2 and ext3. That includes the journalling functionality in ext3.

The highest priority has writing support and journalling, which is a part of ext3. An optional goal will be implementing a good caching to increase the speed. We aim to make the driver robust and easy to use.

1.3 Positioning

Existing ext file system drivers do not have writing support, does not support ext3 journalling or don’t run in a Windows XP environment. We aim to enhance an existing ext2 read only driver, which was developed during a previous thesis. The main disadvantages of this driver are:

- Only read functionality is implemented.
- No ext3 support.

- The file system recognizer don't recognize ext partitions at start up.

The driver can be used by everyone who wants to have read and write access to his Linux file systems on a Windows XP Platform. In contrast to other products, this driver will support all the needed features to have full read and write access to ext file systems and probably other Linux file systems later.

1.3.1 Problem Statement

Table 1.1: Problem Statement

The Problem of	Access partitions with ext2/ext3 file system.
Affects	Users, which have a system with Windows and Linux installed.
The impact of which is	That they need to reboot their system and start Linux in order to use ext2/ext3.
A successful solution would be	An open source and fairly easy to install file system driver for Windows, which allows the user to read from and write to such a partition. A system administrator should have the possibility to map user permissions from ext partition to Windows users and groups.

1.4 Stakeholder Description

1.4.1 Key High-Level Goals

Table 1.2: High-Level Goals

High-Level Goal	**Priority**	**Problems and Concerns**	**Current Solutions**
Easy to use	Medium	Ext specific options which a normal user do not understand.	Registry settings have to be done.
Robustness	High	There must be no loss of data.	Several tests are done.
Fast	Medium	Caching for faster access time and transfer rates.	n/a

1.4.2 User-Level Goals

- Windows User: Create, modify, delete, copy, move files and directories.
- Administrator: Mount/unmount partitions. Manage file permissions and permission mapping.

1.4.3 Alternatives and Competition

Table 1.3: Similar Products

Product	Description
EXT2IFS [5]	EXT2IFS is a read only driver. It provides access similar to other read only media like CDs. It supports directory listings and read only operations. It does not work on Windows XP.
Ext2Fsd [3]	Ext2 File System Driver for Windows NT with planned read/write access. But this driver is only available in version 0.01 Beta and it does not support Windows XP.

1.5 Product Overview

1.5.1 Product Perspective

The driver is meant to be used for a Windows XP environment with an existing ext2/ext3 partition. It is assumed that the installation and configuration will be done by a person which has fairly good knowledge about the Windows and Linux operation systems.

1.5.2 Summary of Benefits

Table 1.4: Benefits

Supporting Feature	Stakeholder Benefit
Mount/Unmount partitions	Change drive letters without rebooting
Read ext2/ext3 partition	Get the files from a Linux partition
Write ext2/ext3 Partition	Create, delete and modify files on a Linux partition

1.5.3 Assumptions and Dependencies

- Windows XP as a runtime environment
- existing ext2/ext3 partition

Chapter 2

Requirements in Detail

2.1 Introduction

This chapter describes all functional and non-functional requirements.

2.2 Functional Requirements

2.2.1 Functions

Table 2.1 shows all functions performed by WinLFS. Functions are sorted by phases. The different phases can be found in the project plan. R1.x correspond to phase Recognizer, R2.x to phase ext2 write, R3.x to phase ext3 read and R4.x to phase ext3 write.

In several requirements, links are mentioned. These links are soft- and hard-links on a Linux partition, not Windows links.

Table 2.1: Detailed description of functional requirements

Reference	Function description	Priority
R1.1	Recognize ext2 partition at boot time.	1
R1.2	Recognize ext3 partition at boot time.	1
R1.3	Start file system drivers.	1
R2.1	Write file on ext2 partition.	1
R2.2	Write directory on ext2 partition.	1

R2.3	Change permissions on a file, directory or link which is on a ext partition.	2
R2.4	Create links on ext2 partition.	3
R2.5	Show ext2 partition preferences.	3
R2.6	Caching for faster writing.	3
R3.1	Read file on ext3 partition and read journal.	1
R3.2	Read directory structure on ext3 partition and read journal.	1
R4.1	Write file on ext3 partition and write journal.	1
R4.2	Write directory on ext3 partition and write journal.	1
R4.3	Create links on ext3 partition and write journal.	3
R4.4	Show ext3 partition preferences.	3
R4.5	Read journal an recover errors.	3
R4.6	Caching for faster writing.	3

2.2.2 Priorities

In the above table, tasks are prioritized. Table 2.2 describes this priorities.

Table 2.2: Functional Requirements Priority

Priority	Description
1	Highest priority. This task must be completed. It will be implemented first.
2	This task must be completed. It will be implemented after tasks with priority 1 have been completed.
3	Optional. If tasks with priority 1 and 2 are completed, optional tasks will be implemented, if there is enough time.

2.3 Non-Functional Requirements

2.3.1 Partitions

Every ext2 and ext3 Partition on the hard disk is recognized at boot time. Optionally, ext partitions on floppy disks are automatically recognized during the first floppy access. If a partition is recognized, a drive letter is associated to this partition, allowing users to work with a ext2/3 partition the same way as with a NTFS or FAT partition. Alternatively, if possible, users can mount or unmount partitions by using Windows XP disk manager. If this is not possible, a console application will be provided.

2.3.2 File Permissions

File permission are handled differently on NTFS and ext2/3 partitions. WinLFS tries to merge this permissions as smart as possible. Merging file system permissions is described in a USENIX [1] paper [2] and will be covered in greater detail in later chapters.

2.3.3 Reliability

Most important, WinLFS must not damage or corrupt any ext2 or ext3 partition. Once loaded, the driver is always available without user interaction.

2.3.4 Performance

Performance is fundamental for any file system driver. The faster the better. Therefore, WinLFS intends to be not slower than 1.5 times the speed of NTFS.

2.3.5 Installation

WinLFS can be installed with the set up tool which starts by clicking on the executable file. It runs on Windows XP exclusively. Installation can be performed by any Windows user who has basic experience in installing programs.

2.3.6 Supportability

After installation, there is nothing to be done to maintain WinLFS.

2.3.7 Implementation Constraints

All WinLFS code is implemented following the GNU Coding Standards [6].

2.3.8 Interfaces

WinLFS uses Windows IFS Development Kit.

[1] http://www.usenix.org

2.3.9 License

WinLFS must not violate Window's IFS Development Kit license. Especially `ntifs.h` must not made public available and must therefore not be shipped with the source code. There is a free version of `ntifs.h` available at `http://www.acc.umu.se/~bosse/ntifs.h`.

Apart from this, WinLFS is free Software distributed under the terms of the GNU General Public License. You can get a copy of this License at `http://www.gnu.org/licenses/gpl.txt`.

Chapter 3

Use Cases

3.1 Use Case Diagram

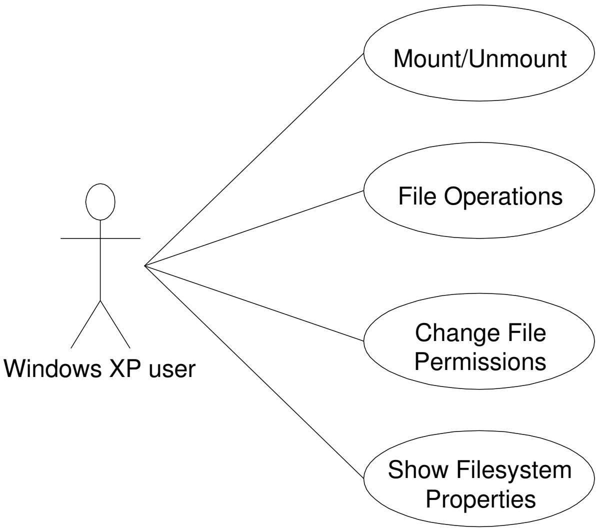


Figure 3.1: Use Case Diagram WinLFS

3.2 Description

Diagram 3.1 shows all actions which an actor can take. All use cases are written in casual style, which describes use cases in an informal paragraph format. This style is more elaborate than the brief style, though less elaborate than fully dressed. It describes the main success scenario including possible errors an conditions. For WinLFS, this style seemed most accurate.

3.3 Actors

Windows XP users who use WinLFS file system driver to access their ext2 or ext3 partition for reading or writing.

3.4 Use Case Descriptions

3.4.1 UC01 - File Operations

Table 3.1: Use Case UC01

Use Case Name	File operations
Use Case Number	UC01
Actors	Windows XP user
Purpose	Perform an operation on a file, directory or link.
Overview	The actor reads or writes a file, a directory or a link on either an ext2 or an ext3 partition.
Formality Type	casual
Priority	1
Cross References	*Functions:* R2.1, R2.2, R2.4, R2.6, R3.1, R3.2, R4.1, R4.2, R4.3, R4.5, R4.6
Main success scenario	
	1. Actor opens a file browser and clicks on a ext2 or ext3 partition. 2. System displays directory structure. 3. Actor tries to copy, move, delete or edit a file, directory or link. 4. System performs requested operation.

Errors	• File, directory or link not found on disk • Partition full • No permission
Preconditions	Partition is mounted
Postconditions	-

3.4.2 UC02 - Mount / Unmount

Table 3.2: Use Case UC02

Use Case Name	Mount / Unmount
Use Case Number	UC02
Actors	Windows XP user
Purpose	Mount or unmount a partition.
Overview	If a partition is not mounted automatically at boot time, the actor can manually mount or unmount a ext2 or ext3 partition by using Windows XP file system Manager. Alternatively the actor can mount a partition by using the provided shell program.
Formality Type	casual
Priority	2
Cross References	*Functions:* R1.1, R1.2, R1.3
Main success scenario	1. System recognizes ext partition on start-up. 2. System starts file system driver. 3. File system driver mounts ext partition. 4. Actor works with mounted partition.

Alternate scenario	1. System could not mount ext partition. 2. Actor opens Computer Management Window and clicks on logical device. 3. System shows all partitions (including ext2 and ext3). 4. Actor selects a partition and associate desired drive letter x. 5. System mounts partition and shows it in the file browser as drive x.
Errors	• Partition not found • Type of partition not valid
Preconditions	Partition is unmounted or mounted.
Postconditions	Partition is mounted or unmounted.

3.4.3 UC03 - Change Permissions

Table 3.3: Use Case UC03

Use Case Name	Change permissions
Use Case Number	UC03
Actors	Windows XP user
Purpose	Change permissions on a file, a directory or a link.
Overview	The actor can manually change the permission on a file, a directory or a link by using the preference window.
Formality Type	casual
Priority	2
Cross References	*Functions:* R2.3

Main success scenario	1. Actor right clicks on a file, directory or a link. 2. System opens the preference window. 3. Actor changes permission. 4. System changes permission.
Errors	No permission
Preconditions	Partition is mounted and file can be found
Postconditions	Permission changed

3.4.4 UC04 - View Partition Information

Table 3.4: Use Case UC04

Use Case Name	View partition information
Use Case Number	UC04
Actors	Windows XP user
Purpose	View information about a partition.
Overview	The actor can view the characteristics of a partition like free space, permissions, etc. These characteristics are shown in the preferences window.
Formality Type	casual
Priority	2
Cross References	*Functions:* R2.5, R4.4
Main success scenario	1. Actor right clicks on a file, directory or link. 2. System opens a window and shows characteristic of the file, directory or link.
Errors	Error on partition - unable to read
Preconditions	Partition is mounted
Postconditions	-

Part II

Technical Report

This part describes the technical facts.

Chapter 4

File System Recognizer

4.1 Overview

This Chapter describes what a file system recognizer is and how it works. The original version is called ExtFsr and was programmed by Matt Wu mattwu@163.com and was only able to recognize ext2 and load the belonging driver. More information about this is available on his website [7].

The functionality was enhanced to differentiate between ext2 and ext3 and then loading the appropriate driver.

4.2 What a file system recognizer is

A file system recognizer is a standard kernel mode driver like every file system driver. It looks at physical media devices and if it recognizes the media format it loads the appropriate full file system driver. The advantage of such a driver is that it needs less memory than a full file system driver. Some full file system driver might never be required, therefore a small driver like a file system recognizer can save several hundred kilobytes of system memory.

4.3 Start and mount procedure

Depending on the settings in the registry, the driver will be loaded on system boot process. Otherwise the driver can be loaded by the command `net start extfsr` manually, given the fact the driver has an entry in the registry.

A mount request will be triggered when windows accesses a not yet mounted partition. When the recognizer receives such a request, it checks the magic number in the super block to decide if it is able to handle the partition. If yes, it checks if journaling is used by the file system.

If ext3 is recognized and a try to load the driver fails, the recognizer tries to load the ext2 driver. That's possible because a ext3 partition is fully accessible through a ext2 driver.

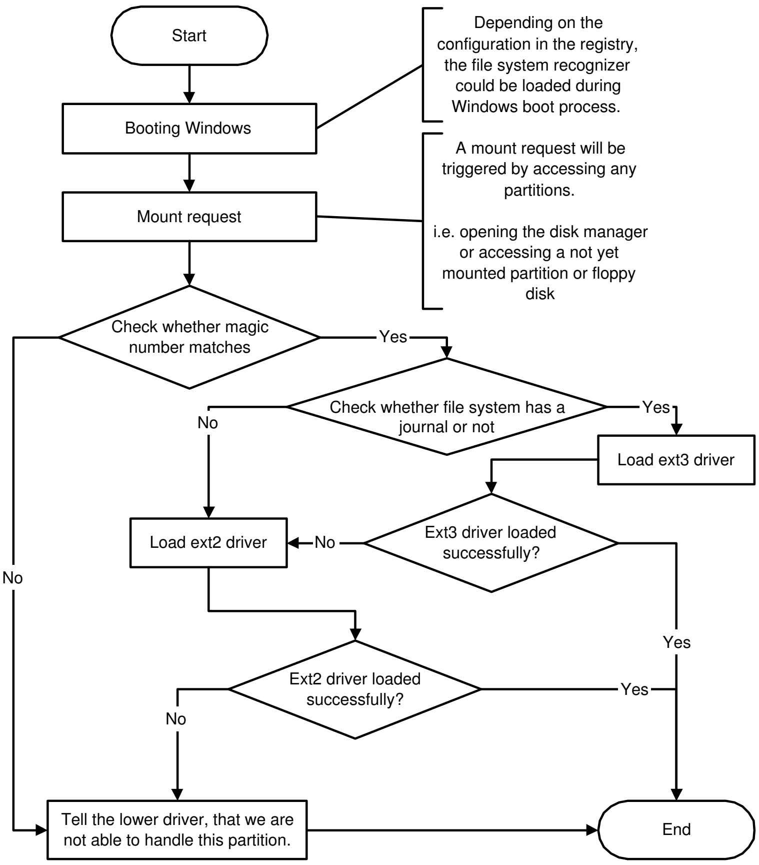

Start
Booting Windows
Depending on the configuration in the registry, the file system recognizer could be loaded during Windows boot process.
Mount request
A mount request will be triggered by accessing any partitions.
i.e. opening the disk manager or accessing a not yet mounted partition or floppy disk
Check whether magic number matches
Yes
No
Check whether file system has a journal or not
No
Yes
Load ext3 driver
Load ext2 driver
No
Ext3 driver loaded successfully?
Yes
Ext2 driver loaded successfully?
No
Yes
Tell the lower driver, that we are not able to handle this partition.
End

Figure 4.1: Start and mount procedure

4.4 How to recognize a ext file system

First, while initializing the driver, it creates a "disk file system device" and registers it as a file system.

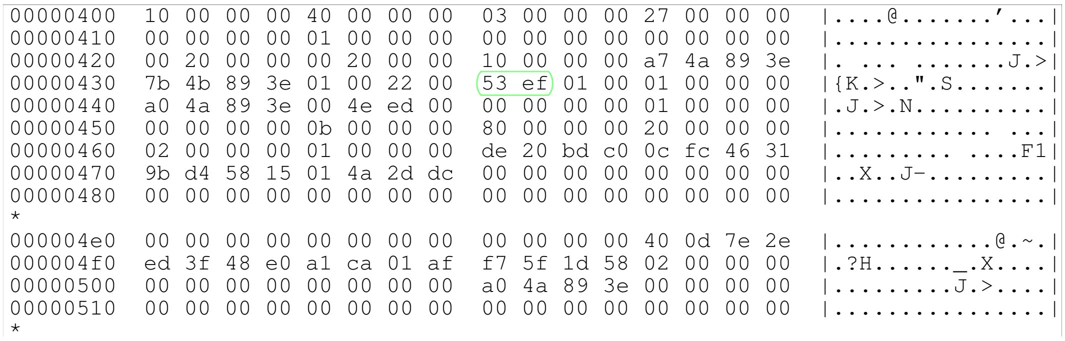
```
Status = IoCreateDevice(
  DriverObject,
  sizeof(ExtFsrDeviceExtension),
  &driverName,
  FILE_DEVICE_DISK_FILE_SYSTEM,
  0,
  FALSE,
  &ExtFsrDeviceObject
);

IoRegisterFileSystem(ExtFsrDeviceObject);
```

From now on a mount request will reach our file system recognizer when a lower driver calls IoCallDriver with an IRP_MJ_FILE_SYSTEM_CONTROL. To handle such requests, we will have to register the function FileSystemControl as follow:

```
DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL]=FileSystemControl;
```

The Irp Minor Value will be IRP_MN_MOUNT_VOLUME when windows tries to mount a new partition. The FileSystemControl function will read the superblock and compare the magic number. The magic number of ext2 and ext3 is 0xef53 (little-endian) or 0x53ef (big-endian). It can be accessed by the struct of the super block (`ext3_super_block->s_magic`). The following dump shows the place of the magic number in the super block.



```
00000400  10 00 00 00 40 00 00 00  03 00 00 00 27 00 00 00  |....@.......'...|
00000410  00 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000420  00 20 00 00 00 20 00 00  10 00 00 00 a7 4a 89 3e  |. ... .......J.>|
00000430  7b 4b 89 3e 01 00 22 00  53 ef 01 00 01 00 00 00  |{K.>..".S.......|
00000440  a0 4a 89 3e 00 4e ed 00  00 00 00 00 01 00 00 00  |.J.>.N..........|
00000450  00 00 00 00 0b 00 00 00  80 00 00 00 20 00 00 00  |............ ...|
00000460  02 00 00 00 01 00 00 00  de 20 bd c0 0c fc 46 31  |......... ....F1|
00000470  9b d4 58 15 01 4a 2d dc  00 00 00 00 00 00 00 00  |..X..J-.........|
00000480  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000004e0  00 00 00 00 00 00 00 00  00 00 00 00 40 0d 7e 2e  |............@.~.|
000004f0  ed 3f 48 e0 a1 ca 01 af  f7 5f 1d 58 02 00 00 00  |.?H......_.X....|
00000500  00 00 00 00 00 00 00 00  a0 4a 89 3e 00 00 00 00  |.........J.>....|
00000510  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

Figure 4.2: Dump of a Super Block with Magic Number

If the super block, returned by the LoadSuperBlock function, contains the correct ext magic number, the status will be set to STATUS_FS_DRIVER_REQUIRED. Whether the partition is an ext2 or an ext3 partition is decided by checking the journal flag. This flag is set with the third bit of `ext3_super_block->s_feature_compat`

```
// definition of the macro and mask
#define EXT3_FEATURE_COMPAT_HAS_JOURNAL  0x0004
#define EXT3_HAS_COMPAT_FEATURE(sb,mask) \
                              ( sb->s_feature_compat & mask )

// usage of this macro
if( EXT3_HAS_COMPAT_FEATURE(ExtSuperBlock, \
                              EXT3_FEATURE_COMPAT_HAS_JOURNAL) )
{
  // some code
}
```

The recognized file system version and whether the full file system drivers are loaded or failed is written to the device extension struct of this DeviceObject. The device extension struct is a self declared struct to save device object relevant properties.

If STATUS_FS_DRIVER_REQUIRED is returned to the lower driver, it will call FileSystemControl again but with Irp Minor Value IRP_MN_LOAD_FILE_SYSTEM. Now the recognizer tries to load the full file system driver for ext2 or ext3 depending on the recognized version memorised in the device extension.

```
if (Extension->journaling)
{
  RtlInitUnicodeString(&RegistryPath, EXT3FSD_REGISTRY_PATH);
}
else if (!Extension->journaling)
{
  RtlInitUnicodeString(&RegistryPath, EXT2FSD_REGISTRY_PATH);
}

Status = ZwLoadDriver(&RegistryPath);
```

When both driver, ext2 and ext3, have either loaded or failed, the recognizer will first unregister itself as a file system. So it will no longer receive mount requests. Then it deletes the device object and gives the command to unload itself from the system.

```
if(    (Extension->ext2.loaded || Extension->ext2.failed)
    && (Extension->ext3.loaded || Extension->ext3.failed) )
{
  IoUnregisterFileSystem(ExtFsrDeviceObject);
  IoDeleteDevice(ExtFsrDeviceObject);
  RtlInitUnicodeString(&RegistryPath, EXTFSR_REGISTRY_PATH);
  ZwUnloadDriver(&RegistryPath);
}
```

4.5 FileSystemControl

Diagram 4.3 shows the way to handle a mount request to the recognizer. If the first call to this function with IRP_MN_MOUNT_REQUEST will return STATUS_FS_DRIVER_REQUIRED, we will receive a second request with IRP_MN_LOAD_FILE_SYSTEM. Only on second pass a file system will be loaded if everything went right.

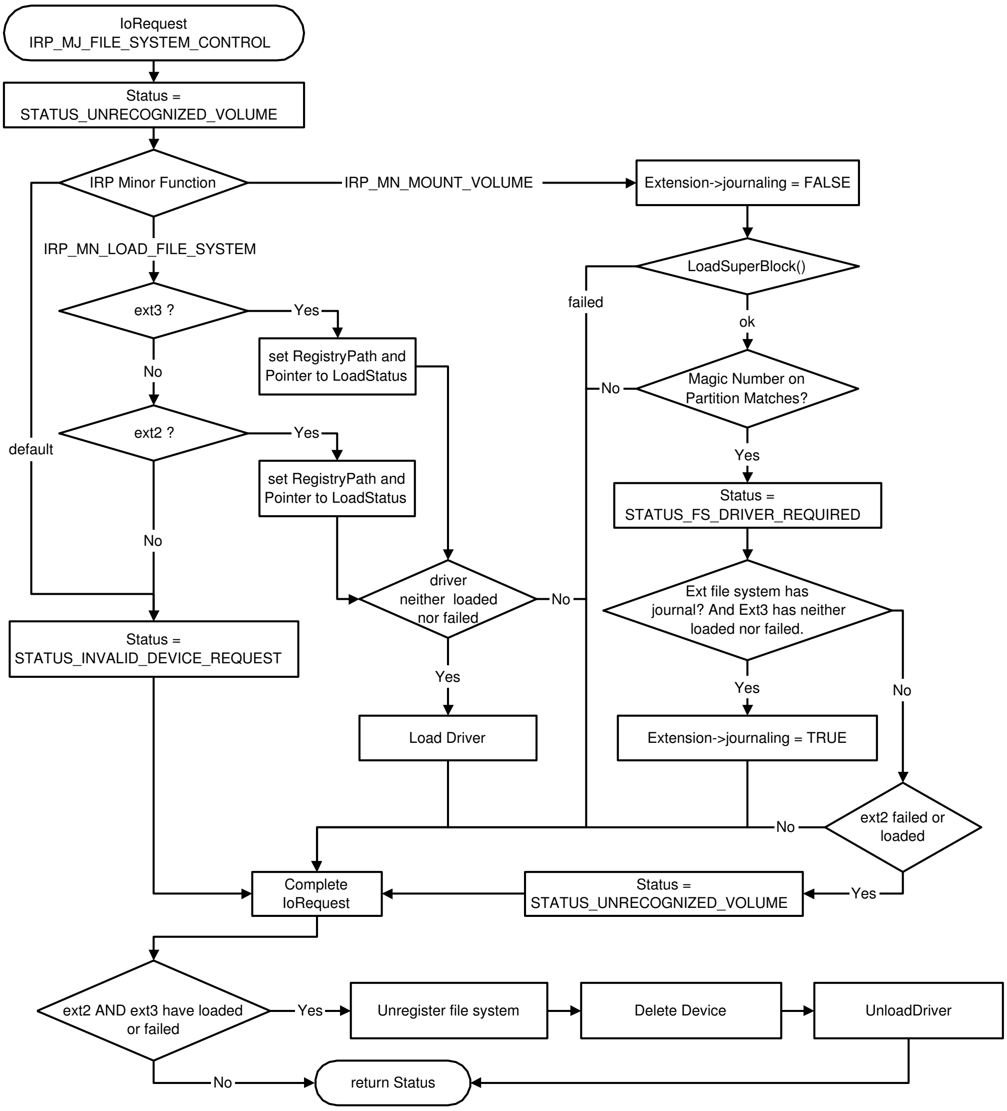


Figure 4.3: File System Control Diagram

Chapter 5

Unload functionality

5.1 Overview

There's no official way to unload a file system driver with Windows XP. But during development it would be very comfortable if one had the possibility to unload the driver. So you could avoid the need to reboot Windows each time you have to test the new driver version.

Due to the fact it would be a really useful feature, there is a workaround available already. First we spent a few hours to search an own solution, but then we received a hint in a news group. Bo Brantén told us, that he solved this problem already in his open source RomFS driver.

5.2 Realisation

5.2.1 Private IoControl

IOCTL_PREPARE_TO_UNLOAD is a macro using the system-supplied macro CTL_CODE to set up a new I/O control code.

```
#define IOCTL_PREPARE_TO_UNLOAD \
    CTL_CODE(FILE_DEVICE_UNKNOWN, 2048, METHOD_NEITHER, FILE_WRITE_ACCESS)
```

The macro takes the following parameters: (partially taken from DDK Help)

DeviceType matches the value set in the DeviceType member of the driver's DEVICE_OBJECT structure.

FunctionCode is in the range 0x800 to 0xfff for private I/O control codes defined by customers of Microsoft. Values in the range 0x000 to 0x7ff are reserved by Microsoft for public I/O control codes.

TransferType indicates how data is passed to the driver. With METHOD_NEITHER, the driver can be sent such a request only while it is running in the context of the thread that originates the I/O control request. Only a highest-level kernel-mode driver is guaranteed to meet this condition.

RequiredAccess indicates the type of access that must be requested when the caller opens the file object representing the device. With FILE_WRITE_DATA, the driver can carry out the requested operation only for a caller with write access rights.

5.2.2 Request to unload

If the request to unload from the unload tool reaches the driver, the Ext2FsdDeviceControl function will be called. The IoControlCode can be accessed through the IRP stack and must have the value of IOCTL_PREPARE_TO_UNLOAD. Further the function Ext2FsdPrepareToUnload will be called. That's the place where the driver will check if it is prepared to unload already or if there exists any mounted devices.

Now, when everything is okay, the driver calls IoUnregisterFileSystem and IoDeleteDevice. Furthermore it sets the unload function.

```
NTSTATUS Ext2FsdPrepareToUnload (IN PFSD\_IRP\_CONTEXT IrpContext)
{
  //   This is only a short overview, you can find the
  //   whole code in devctl.c

  __try {
    if (FlagOn(FsdGlobalData.Flags, FSD_UNLOAD_PENDING)) {
      KdPrint((DRIVER_NAME ": *** Already ready to unload ***\n"));
      Status = STATUS_ACCESS_DENIED;
      __leave;
    }

    if (!IsListEmpty(&FsdGlobalData.VcbList)) {
      KdPrint((DRIVER_NAME ": *** Mounted volumes exists ***\n"));
      Status = STATUS_ACCESS_DENIED;
      __leave;
    }
```

```

      IoUnregisterFileSystem(FsdGlobalData.DeviceObject);
      IoDeleteDevice(FsdGlobalData.DeviceObject);

      FsdGlobalData.DriverObject->DriverUnload = DriverUnload;
      SetFlag(FsdGlobalData.Flags, FSD_UNLOAD_PENDING);
      KdPrint((DRIVER_NAME ": Driver is ready to unload\n"));
      Status = STATUS_SUCCESS;
    }
    __finally
    {
      IrpContext->Irp->IoStatus.Status = Status;
      FsdCompleteRequest(IrpContext->Irp,
          (CCHAR)(NT_SUCCESS(Status) ? IO_DISK_INCREMENT : IO_NO_INCREMENT));
    }
    return Status;
  }
```

5.3 DDK Kit Version

Because the Unload procedure differs on Windows XP compared to previous versions, there was a compiler directive to decide, which procedure has to be used.

```
#if (VER_PRODUCTBUILD < 2600)
   IoDeleteDevice(FsdGlobalData.DeviceObject);
#endif
```

Even though we had the DDK Kit Version 2600, the build tool has a lower build number. So the comiler directives had to be removed that the method for Windows XP is hardcoded.

The changes are documented in the code.

5.4 Benefit

With this feature fixing minor bugs could be much more efficient. Unfortunately it is useless for bugs which cause a bluescreen.

Chapter 6

(Un)mount and unload tools

6.1 Overview

There are two tools for mounting, unmounting drives and unloading a driver. The mount and unmount tools are provided by Matt Wu mattwu@163.com and the unload binary is provided by Bo Brantén.

Both executables can be found on our cdrom in the Directory `X:\binaries\` where `X:` is the letter of the cdrom drive.

6.2 Mount.exe

This executable can be used for assigning a drive letter to a partition or release a mounted partiton.

6.2.1 mount

```
mount.exe 0 7 k:
```

The first number identifies the disk beginning from zero, the second number is the partition number beginning from one. The third parameter is desired drive letter.

6.2.2 unmount

```
mount.exe /umount k:
```

The parameter `/umount` is used to unmount a mounted partition by specifying a drive letter.

6.3 Unload.exe

This executable can be used to unload a running driver. Due to Windows XP is not able to unload a driver, there is needed this seperate tool. The driver must support unloading. The code needed to do so is described in Chapter 5.

```
unload.exe Ext2Fsd
```

The parameter specifies the name of the driver to be unloaded.

Chapter 7

Ext2 File System

7.1 Overview

This chapter gives an overview of the ext2 file system. It is not meant to be exhaustive, it should rather give a brief overview. For a more detailed documentation of this file system, please either read the documentation included in your Linux distribution or read understanding the Linux kernel [1].

7.2 Structure

Ext2 shares many properties with traditional Unix file systems. It has the concepts of blocks, inodes and directories. There is also a versioning mechanism to allow new features (such as journalling) to be added in a maximally compatible manner. There is space for other features like ACL's (Access Control Lists), undeletion or compression in the specification but these features are not of interest in this thesis and are therefore not documented.

Figure 7.1 shows the structure of an Ext2 partition. The first block (Boot Block) is never managed by the Ext2 file system, since it is reserved for the partition boot sector. The rest of the space is split up into blocks of fixed size (1024, 2048 or 4096 bytes). The size of a block is defined when the file system is created. Blocks are clustered into block groups in order to reduce fragmentation and minimise the amount of head seeking when reading a large amount of consecutive data. Since each bitmap (Data block Bitmap and Inode Bitmap) is limited to a single block, the maximum size of a block group is 8 times the size of a block.

Blocks are stored sequentially. Although both, the Super Block and the Group Descriptor are duplicated in each block group, only the Super Block an Group

Descriptor from Block Group 0 are used.

All structures are stored on the disc in little endian format, so a file system is portable between machines without having to know what machine it was created on.

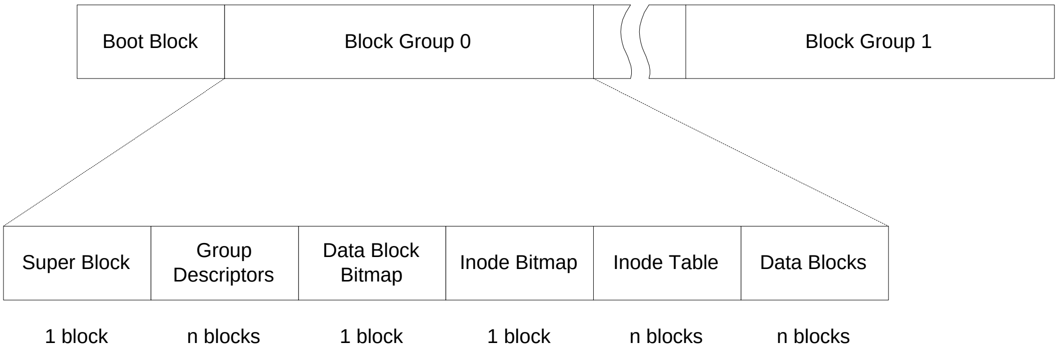


Figure 7.1: Structure of an Ext2 partition

7.2.1 Super Block

The Super Block contains all the information about the configuration of the filing system. The primary copy of the Super Block is stored at an offset of 1024 bytes from the start of the device, and it is essential for mounting the files system.

The information in the Super Block contains fields such as the total number of inodes and blocks in the file system and how many are free, how many inodes and blocks are in each block group, when the file system was mounted (and if it was cleanly unmounted), when it was modified, what version of the file system it is (see the Revisions section below) and which OS created it.

7.2.2 Inodes

The inode (index node) is a fundamental concept in the ext2 file system. Each object in the files system is represented by an inode. The inode structure contains pointers to the file system blocks which contain the data held in the object and all of the metadata about an object except its name. Every inode field is 128 bytes of size. How a file's data blocks is addressed can be seen in figure 7.2.

The metadata about an object includes the permissions, owner, group, flags, size, number of blocks used, access time, change time, modification time, deletion time, number of links, fragments, version (for NFS) and extended attributes (EAs) and/or Access Control Lists (ACLs).

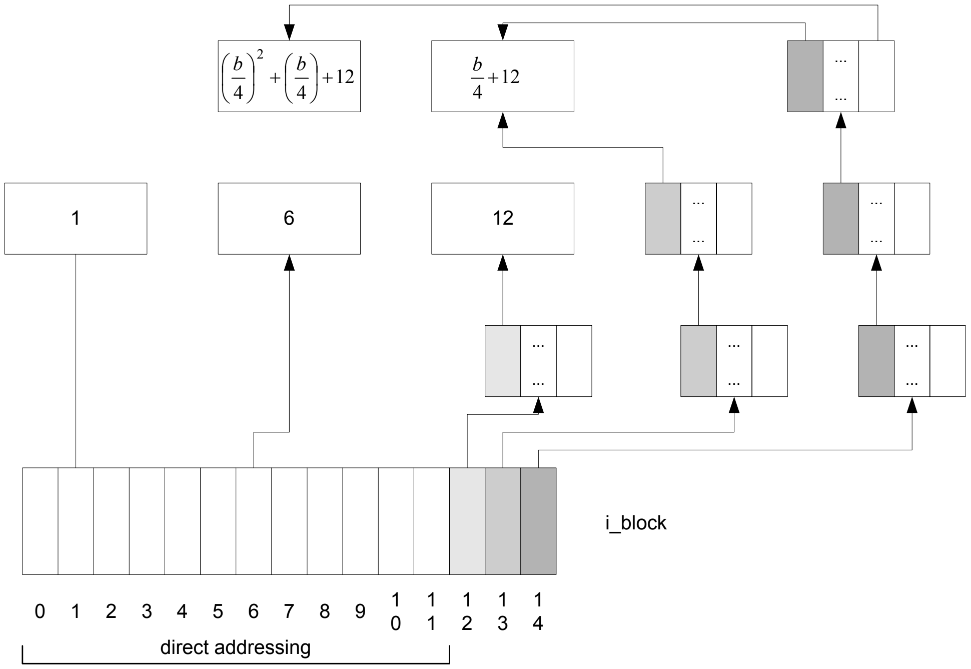


Figure 7.2: Data structure used to address the file's data block

There are pointers to the first 12 blocks which contain the file's data in the inode. This mechanism favourites small files. If a file is smaller or equal 12 blocks, it can be addresses directly. If a file is bigger, indirect addressing is used, this means, a file can be retrieved in two disk accesses: One to read the content of the `i-block` array and one to read the content of the data block itself. For even larger files, three or four consecutive disk accesses are needed to access a data block. (E.g. Component at index 12 contains the logical block number of a block that represents a second-order array of logical block numbers. They correspond to the file block number ranging from 12 to $\frac{b}{4}+11$ where b is the file system's block size.)

Chapter 8

File System Driver

8.1 Overview

This documentation does not provide information about the read functionality of the driver. Also the basic concepts of a file system driver and general descriptions are not included. This information have already been given in the documentation of the continuing studies diploma thesis [4].

8.2 Ext2 Write

8.2.1 General Data flow

Whenever a user mode application or a kernel mode application requests an operation from a file system driver (FSD), the I/O Manager allocates an IRP (I/O request packet) and calls the FSD with this IRP. The FSD accesses its I/O stack location in the IRP to determine what operation (indicated by the `IRP_MJ_XXX` function code) it should carry out on the target device. The target device is represented by the device object in its designated I/O stack location and is passed with the IRP to the driver.

8.2.2 Create a new file

An IRP_MJ_CREATE is sent to the FSD whenever a new file or directory is created, or when a an existing file, device, directory, or volume is being opened. Normally this IRP is sent when an application has called `CreateFile()`. An IRP_MJ_WRITE is sent to the FSD, for example, when an application has called `WriteFile()`.

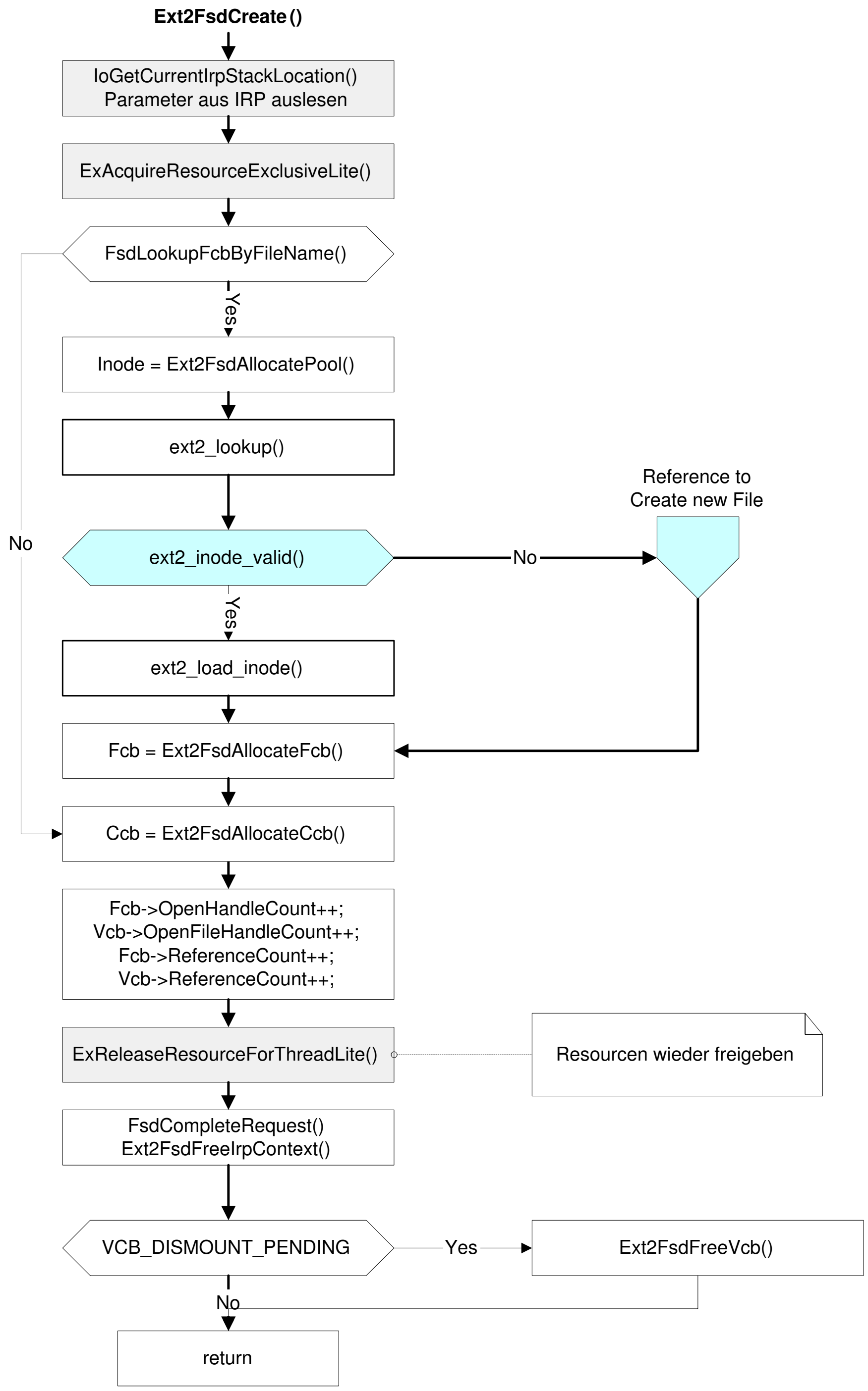


Figure 8.1: Process Flow when receiving IRP_MJ_CREATE

This means, whenever the FSD receives a IRP_MJ_CREATE, it tests if a new file needs to be created. If so, preliminary tests (file or directory name validity, flags, etc.) are made and a new Inode is created. Figure 8.1 illustrates this procedure of receiving an IRP_MJ_CREATE request and figure 8.2 illustrates the process of creating a new inode in detail. The two diagrams do not describe function in detail. They only give a general overview of the basic data flow.

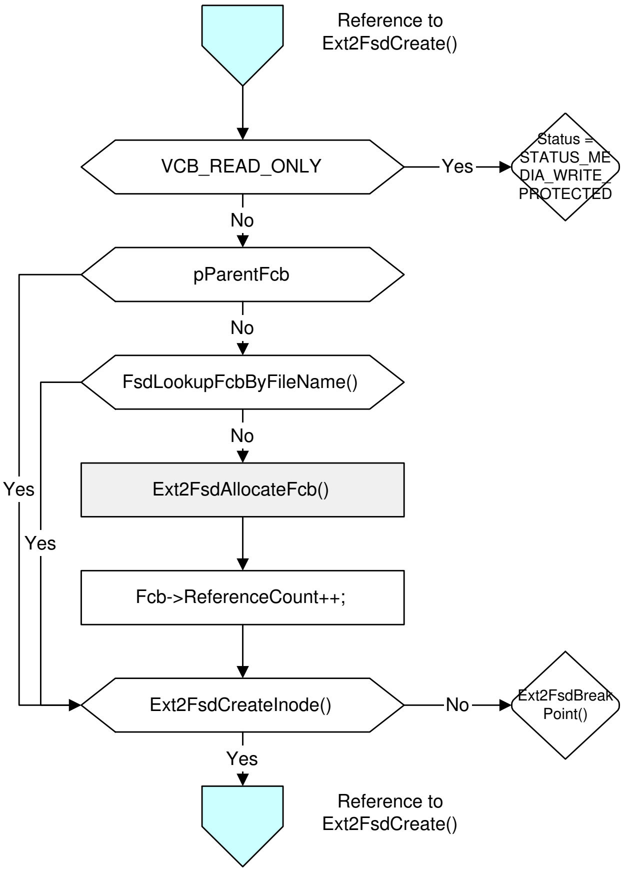


Figure 8.2: Process flow when creating a new inode

Every function in the above two diagrams are subject to describe in more detail. This has been done in the code documentation in part III. For even more details please consult the code.

Part III

Implementation

This part contains code documentation.

Chapter 9

Code Documentation

9.1 Overview

The following two chapters 10 and 11 comprises the documentation of the code written or modified during this thesis. The missing parts can be found in the documentation of the continuing studies thesis.

9.2 Doxygen

The code documentation is generated by Doxygen[1]. This is a documentation generator like Javadoc and supports C, Qt and Javadoc comment styles.

We produced LaTeX and HTML output. The LaTeX output we integrated directly in our main documentation. The HTML version is available on the cdrom.

[1] http://www.doxygen.org/

Chapter 10

ExtFsr - Recognizer

10.1 Data Structure Documentation

10.1.1 _ExtFsrDeviceExtension Struct Reference

`#include <ExtFsr.h>`

Detailed Description

Device extension structure.

Structure to store persistent values that are needed by a later IRP request.

Data Fields

- LOAD_STATUS ext3

  Stores whether ext3 driver is loaded or failed.

- LOAD_STATUS ext2

  Stores whether ext2 driver is loaded or failed.

- BOOLEAN journalling

  Stores whether recognized partition has journalling or not.

10.1.2 _LOAD_STATUS Struct Reference

`#include <ExtFsr.h>`

Detailed Description

Load status structure.

Structure to store loaded and failed statuses of the drivers.

Data Fields

- BOOLEAN loaded

  Stores whether driver is loaded already.

- BOOLEAN failed

  Stores whether driver failed on a previous try to load.

10.1.3 ext3_super_block Struct Reference

`#include <ExtFsr.h>`

Detailed Description

Ext3 structure taken from linux source.

Only attributes used in this driver are described. Short descriptions of the other attributes can be found in the source code.

Data Fields

- USHORT s_magic

  Magic signature.

- ULONG s_feature_compat

  Compatible feature set.

Field Documentation

ULONG ext3_super_block::s_feature_compat

Compatible feature set.

This attribute is needed to determine if there is a ext2 or ext3 partition.

USHORT ext3_super_block::s_magic

Magic signature.

This is the signature of the partition. The ext2 and ext3 partitions have 0xEF53 at this place.

10.2 File Documentation

10.2.1 ExtFsr.c File Reference

Detailed Description

Complete file system recognizer.

This file contains all functions needed by file system recognizer.

`#include <ifs/ntifs.h>`

`#include "ExtFsr.h"`

Functions

- NTSTATUS ReadDisk (IN PDEVICE_OBJECT DeviceObject, IN LONGLONG Offset, IN ULONG Length, OUT PVOID Buffer)

  Reads data from physical disk into memory.

- PEXT3_SUPER_BLOCK LoadSuperBlock (IN PDEVICE_OBJECT DeviceObject)

  Loads super block from disk.

- NTSTATUS FileSystemControl (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)

Manage mount requests and loads file system drivers.

- VOID ExtFsrUnload (IN PDRIVER_OBJECT DriverObject)

  Unload function.

- NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)

  Entry point of this driver.

Function Documentation

NTSTATUS DriverEntry (PDRIVER_OBJECT *DriverObject*, PUNICODE_STRING *RegistryPath*)

Entry point of this driver.

It initializes global variables and registers the file system.

Parameters:

DriverObject Points to the next-lower driver's device object representing the target device for the read operation.

RegistryPath Pointer to a counted Unicode string specifying the path to the driver's registry key.

Returns:

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in ntstatus.h.

VOID ExtFsrUnload (IN PDRIVER_OBJECT *DriverObject*)

Unload function.

This function cleans up reserved resources if this driver is no longer in use.

Parameters:

DriverObject Points to the next-lower driver's device object representing the target device for the read operation.

NTSTATUS FileSystemControl (IN PDEVICE_OBJECT *DeviceObject*, IN PIRP *Irp*)

Manage mount requests and loads file system drivers.

This function manages IRP_MN_MOUNT_VOLUME and IRP_MN_LOAD_FILE_SYSTEM. If a ext3 file system is regocnized, it will try to load the ext3 driver. If that was not successfully, the next request of the higher level driver will try to load the ext2 file system.

If ext2 and ext3 both either are loaded successfully or have failed to load, the file system recognizer unloads itself.

Parameters:

DeviceObject Points to the next-lower driver's device object representing the target device for the read operation.

Irp I/O request packet.

Returns:

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in ntstatus.h.

PEXT3_SUPER_BLOCK LoadSuperBlock (IN PDEVICE_OBJECT *DeviceObject*)

Loads super block from disk.

This function allocates memory from nonpaged pool for the super block. The superblock is 1024 bytes and the sector size is 512 bytes. So we need two times the SECTOR_SIZE. As an offset we also take two times SECTOR_SIZE because the super block is the second block on a partition.

If the call of ReadDisk fails, the allocated memory will be freed and a null pointer will be returned. If a pointer to the buffer is returned, the allocated memory must be freed from calling function.

Parameters:

DeviceObject Device object of the mounted volume

Returns:

PEXT3_SUPER_BLOCK

NTSTATUS ReadDisk (IN PDEVICE_OBJECT *DeviceObject*, IN LONGLONG *Offset*, IN ULONG *Length*, OUT PVOID *Buffer*)

Reads data from physical disk into memory.

This function builds an IRP for a FSD request, calls the lower driver with this IRP and wait for the completion of this operation. The read data will be written to the Buffer.

Parameters:

DeviceObject Points to the next-lower driver's device object representing the target device for the read operation.

Offset Points to the offset on the disk to read from.

Length Specifies the length, in bytes, of Buffer.

Buffer Points to a buffer to receive data.

Returns:

Status Status of the Operation

10.2.2 ExtFsr.h File Reference

Detailed Description

Headerfile for ExtFsr.c.

This file contains structures, constants and function prototypes used by ExtFsr.c.

`#include <ntdddisk.h>`

Data Structures

- struct _ExtFsrDeviceExtension

  Device extension structure.

- struct _LOAD_STATUS

  Load status structure.

- struct ext3_super_block

  Ext3 structure taken from linux source.

Typedefs

- typedef _LOAD_STATUS LOAD_STATUS

  Load status structure.

- typedef _LOAD_STATUS * PLOAD_STATUS

  Load status structure.

- typedef _ExtFsrDeviceExtension ExtFsrDeviceExtension

  Device extension structure.

- typedef _ExtFsrDeviceExtension * DEVICE_EXTENSION

  Device extension structure.

Functions

- NTSTATUS ReadDisk (IN PDEVICE_OBJECT DeviceObject, IN LONGLONG Offset, IN ULONG Length, OUT PVOID Buffer)

  Reads data from physical disk into memory.

- PEXT3_SUPER_BLOCK LoadSuperBlock (IN PDEVICE_OBJECT DeviceObject)

  Loads super block from disk.

- VOID ExtFsrUnload (IN PDRIVER_OBJECT DeviceObject)

  Unload function.

- NTSTATUS FileSystemControl (IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)

  Manage mount requests and loads file system drivers.

Typedef Documentation

typedef struct _ExtFsrDeviceExtension * DEVICE_EXTENSION

Device extension structure.

Structure to store persistent values that are needed by a later IRP request.

typedef struct _ExtFsrDeviceExtension ExtFsrDeviceExtension

Device extension structure.

Structure to store persistent values that are needed by a later IRP request.

typedef struct _LOAD_STATUS LOAD_STATUS

Load status structure.

Structure to store loaded and failed statuses of the drivers.

typedef struct _LOAD_STATUS * PLOAD_STATUS

Load status structure.

Structure to store loaded and failed statuses of the drivers.

Function Documentation

VOID ExtFsrUnload (IN PDRIVER_OBJECT *DriverObject*)

Unload function.

This function cleans up reserved resources if this driver is no longer in use.

Parameters:

DriverObject Points to the next-lower driver's device object representing the target device for the read operation.

NTSTATUS FileSystemControl (IN PDEVICE_OBJECT *DeviceObject*, IN PIRP *Irp*)

Manage mount requests and loads file system drivers.

This function manages IRP_MN_MOUNT_VOLUME and IRP_MN_LOAD_FILE_SYSTEM. If a ext3 file system is regocnized, it will try to load the ext3 driver. If that was not successfully, the next request of the higher level driver will try to load the ext2 file system.

If ext2 and ext3 both either are loaded successfully or have failed to load, the file system recognizer unloads itself.

Parameters:

DeviceObject Points to the next-lower driver's device object representing the target device for the read operation.

Irp I/O request packet.

Returns:

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in ntstatus.h.

PEXT3_SUPER_BLOCK LoadSuperBlock (IN PDEVICE_OBJECT *DeviceObject*)

Loads super block from disk.

This function allocates memory from nonpaged pool for the super block. The superblock is 1024 bytes and the sector size is 512 bytes. So we need two times the SECTOR_SIZE. As an offset we also take two times SECTOR_SIZE because the super block is the second block on a partition.

If the call of ReadDisk fails, the allocated memory will be freed and a null pointer will be returned. If a pointer to the buffer is returned, the allocated memory must be freed from calling function.

Parameters:

DeviceObject Device object of the mounted volume

Returns:

PEXT3_SUPER_BLOCK

NTSTATUS ReadDisk (IN PDEVICE_OBJECT *DeviceObject*, IN LONGLONG *Offset*, IN ULONG *Length*, OUT PVOID *Buffer*)

Reads data from physical disk into memory.

This function builds an IRP for a FSD request, calls the lower driver with this IRP and wait for the completion of this operation. The read data will be written to the Buffer.

Parameters:

DeviceObject Points to the next-lower driver's device object representing the target device for the read operation.

Offset Points to the offset on the disk to read from.

Length Specifies the length, in bytes, of Buffer.

Buffer Points to a buffer to receive data.

Returns:

Status Status of the Operation

Chapter 11

Ext2Fsd - Full file system driver

11.1 Data Structure Documentation

11.1.1 _FSD_IRP_CONTEXT Struct Reference

```
#include <fsd.h>
```

Detailed Description

Irp context struct.

Used to pass information about a request between the drivers functions.

Data Fields

- PIRP Irp

  Pointer to the IRP this request describes.

- UCHAR MajorFunction

  Major request.

- UCHAR MinorFunction

  Minor request.

- PDEVICE_OBJECT DeviceObject

  Pointer to the target device object.

11.1.2 ext2_t Struct Reference

`#include <ext2.h>`

Detailed Description

Ext2 structure.

This structure represents an ext2 partition in memory.

Data Fields

- super_block_t superblock

  Super block of this ext2 partition.

- group_desc_t * group_desc

  Pointer to the group descriptor of this partition.

11.1.3 group_desc_t Struct Reference

`#include <ext2.h>`

Detailed Description

Group descripton structure.

This structure represents the group description block in memory.

Block number 2 of a block group.

Data Fields

- _u32 bg_block_bitmap

Block number of block bitmap.

- _u32 bg_inode_bitmap

  Block number of inode bitmap.

- _u32 bg_inode_table

  Block number of first inode table block.

- _u16 bg_free_blocks_count

  Number of free blocks in the group.

- _u16 bg_free_inodes_count

  Number of free inodes in the group.

11.1.4 super_block_t Struct Reference

`#include <ext2.h>`

Detailed Description

Super block structure.

This structure represents the super block in memory.

Block number 1 of a block group.

Data Fields

- _u32 s_inodes_count

  Total number of inodes.

- _u32 s_blocks_count

  Filesystem size in blocks.

- _u32 s_blocks_per_group

  Number of blocks per group.

- _u32 s_inodes_per_group

  *Number of inodes per group */.*

- _u16 s_magic

  Magic signature.

Field Documentation

_u16 super_block_t::s_magic

Magic signature.

This is the signature of the partition. The ext2 and ext3 partitions have 0xEF53 at this place.

11.2 File Documentation

11.2.1 alloc.c File Reference

Detailed Description

Allocates and frees memory for structures.

This file contains all allocate and free functions to manage the memory used by structures.

```
#include "ext2.h"
```

```
#include "io.h"
```

```
#include "ntifs.h"
```

```
#include "fsd.h"
```

Functions

- inode_t * Ext2FsdAllocateInode ()

  Allocates memory for inode_t structure from paged pool.

- void Ext2FsdFreeInode (inode_t *inode)

  Frees memory for inode_t structure.

Function Documentation

inode_t* Ext2FsdAllocateInode ()

Allocates memory for inode_t structure from paged pool.

void Ext2FsdFreeInode (inode_t * *inode*)

Frees memory for inode_t structure.

11.2.2 cache.c File Reference

Detailed Description

Implements caching functions.

This file was not modified during this thesis.

```
#include "ext2.h"
#include "io.h"
```

11.2.3 create.c File Reference

Detailed Description

Functions to handle IRP_MJ_CREATE.

The I/O Manager sends this request when a new file or directory is being created, or when an existing file, device, directory, or volume is being opened. Normally this IRP is sent on behalf of a user-mode application that has called a function such as CreateFile or a kernel-mode component that has called IoCreateFile, ZwCreateFile, or ZwOpenFile.

```
#include "ntifs.h"
```

```
#include "ext2.h"

#include "fsd.h"
```

Functions

- NTSTATUS Ext2FsdCreateFile (IN PFSD_IRP_CONTEXT IrpContext)

  Creates the structures in the memory representing a physical file.

- PFSD_FCB FsdLookupFcbByFileName (IN PFSD_VCB Vcb, IN PUNICODE_STRING FullFileName)

  Checks if FCB for a file is in the memory already.

Function Documentation

NTSTATUS Ext2FsdCreateFile (IN PFSD_IRP_CONTEXT *IrpContext*)

Creates the structures in the memory representing a physical file.

If a file already exists on the disk, this function creates all needed structure to represent the file in memory.

If the file does not yet exists, it looks for free inodes on disk and creates the directory entries and structures for the new allocated inodes.

Parameters:

IrpContext Pointer to the context of the IRP

Returns:

Status if all allocations were successful.

PFSD_FCB FsdLookupFcbByFileName (IN PFSD_VCB *Vcb*, IN PUNICODE_STRING *FullFileName*)

Checks if FCB for a file is in the memory already.

Returns FCB for FullFileName if already exists in FcbList of Vcb or NULL if FCB not yet exists.

Parameters:

Vcb VolumeControlBlock of mounted Partition

FullFileName Full Path plus Filename

Returns:

Fcb of requested File

11.2.4 ext2.h File Reference

Detailed Description

Headerfile for ext2.c, super.c, group.c, inode.c and cache.c.

Declares any structures and function prototypes used by the driver. Only fields used during this thesis are listed. Short description of other fields can be found in the source code.

`#include "stdlib.h"`

`#include "windows.h"`

`#include "cache.h"`

Data Structures

- struct ext2_t

  Ext2 structure.

- struct group_desc_t

  Group descripton structure.

- struct super_block_t

  Super block structure.

Functions

- inode_t ∗ Ext2FsdAllocateInode ()

  Allocates memory for inode_t structure from paged pool.

- void Ext2FsdFreeInode (inode_t *inode)

  Frees memory for inode_t structure.

Function Documentation

inode_t* Ext2FsdAllocateInode ()

Allocates memory for inode_t structure from paged pool.

void Ext2FsdFreeInode (inode_t * *inode*)

Frees memory for inode_t structure.

11.2.5 ext2.c File Reference

Detailed Description

Implements functions to handle Ext2 partitions.

This file was not modified during this thesis.

```
#include "ext2.h"

#include "fsd.h"

#include "io.h"

#include "stdio.h"

#include "time.h"
```

11.2.6 group.c File Reference

Detailed Description

Implements functions to handle group descriptors.

This file was not modified during this thesis.

```
#include "ext2.h"
```

```
#include "io.h"
#include <stdio.h>
```

11.2.7 init.c File Reference

Detailed Description

Entry and Unload functions.

The Entry and Unload functions are needed to initialize and unload the driver.

```
#include "ntifs.h"
#include "fsd.h"
```

Functions

- NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING RegistryPath)

  Entry point of the driver.

Function Documentation

NTSTATUS DriverEntry (IN PDRIVER_OBJECT *DriverObject*, IN PUNICODE_STRING *RegistryPath*)

Entry point of the driver.

This function initializes the driver after loading.

Parameters:

DriverObject DRIVER_OBJECT structure of the lower driver.

RegistryPath Unicode string to the driver's registry key.

Returns:

If the driver initialization was successful.

11.2.8 inode.c File Reference

Detailed Description

Functions for inode operations.

```
#include "ext2.h"
```

```
#include "io.h"
```

```
#include "stdio.h"
```

```
#include "stdlib.h"
```

```
#include "string.h"
```

```
#include "time.h"
```

Functions

- NTSTATUS Ext2FsdCreateInode (PFSD_IRP_CONTEXT IrpContext, PFSD_VCB Vcb, PFSD_FCB pParentFcb, ULONG ParentInodeNumber, ULONG Type, ULONG FileAttr, PUNICODE_STRING FileName)

  Creates new Inode on disk and makes Directory entry.

- BOOLEAN Ext2FsdNewInode (PFSD_IRP_CONTEXT IrpContext, PFSD_VCB Vcb, ULONG GroupHint, ULONG Type, PULONG InodeNumber)

  Allocate a new Inode on the partition.

- NTSTATUS Ext2FsdAddEntry (IN PFSD_IRP_CONTEXT IrpContext, IN PFSD_VCB Vcb, IN PFSD_FCB Dcb, IN ULONG FileType, IN ULONG InodeNumber, IN PUNICODE_STRING FileName)

  Add a new directory entry.

- BOOLEAN Ext2FsdDeleteInode (PFSD_IRP_CONTEXT IrpContext, PFSD_VCB Vcb, ULONG Inode, ULONG Type)

  Free an inode on the disk.

- NTSTATUS Ext2FsdRemoveEntry (IN PFSD_IRP_CONTEXT IrpContext, IN PFSD_VCB Vcb, IN PFSD_FCB Dcb, IN ULONG Inode)

  Remove a directory entry.

- BOOLEAN Ext2FsdSaveGroup (IN PFSD_IRP_CONTEXT IrpContext, IN PFSD_VCB Vcb)

  Write group descriptor to the disk.

- BOOLEAN Ext2FsdSaveInode (IN PFSD_IRP_CONTEXT IrpContext, IN PFSD_VCB Vcb, IN ULONG Inode, IN inode_t *inode)

  Write inode to the disk.

- BOOLEAN Ext2FsdSaveSuper (IN PFSD_IRP_CONTEXT IrpContext, IN PFSD_VCB Vcb)

  Write super block to the disk.

- BOOLEAN Ext2FsdGetInodeLba (IN PFSD_VCB vcb, IN ULONG inode, OUT PLONGLONG offset)

  Get offset on physical partition.

Function Documentation

NTSTATUS Ext2FsdAddEntry (IN PFSD_IRP_CONTEXT *IrpContext*, IN PFSD_VCB *Vcb*, IN PFSD_FCB *Dcb*, IN ULONG *FileType*, IN ULONG *InodeNumber*, IN PUNICODE_STRING *FileName*)

Add a new directory entry.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

Dcb Directory control block

FileType Type of entry (file or directory)

InodeNumber Number of inode

FileName String with name of the new entry

Returns:

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in ntstatus.h.

NTSTATUS Ext2FsdCreateInode (PFSD_IRP_CONTEXT *IrpContext*, PFSD_VCB *Vcb*, PFSD_FCB *pParentFcb*, ULONG *ParentInodeNumber*, ULONG *Type*, ULONG *FileAttr*, PUNICODE_STRING *FileName*)

Creates new Inode on disk and makes Directory entry.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

pParentFcb FCB of the parent directory

ParentInodeNumber Inode number of the parent directory

Type Type of entry (file or directory)

FileName

Returns:

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in ntstatus.h.

BOOLEAN Ext2FsdDeleteInode (PFSD_IRP_CONTEXT *IrpContext*, PFSD_VCB *Vcb*, ULONG *Inode*, ULONG *Type*)

Free an inode on the disk.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

Inode Number of inode

Type Type of entry (file or directory)

Returns:

If the operation finished successfully.

BOOLEAN Ext2FsdGetInodeLba (IN PFSD_VCB *vcb*, IN ULONG *inode*, OUT PLONGLONG *offset*)

Get offset on physical partition.

Parameters:

vcb Volume control block

inode Number of inode

offset Offset on physical partition

Returns:

If the operation finished successfully.

BOOLEAN Ext2FsdNewInode (PFSD_IRP_CONTEXT *IrpContext*, PFSD_VCB *Vcb*, ULONG *GroupHint*, ULONG *Type*, PULONG *InodeNumber*)

Allocate a new Inode on the partition.

The algoritm of this functions tries first to allocate an inode in the same block group. If there are no more inodes available it searches inodes in other block groups.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

GroupHint Number of block group where search starts

Type Type of entry (file or directory)

InodeNumber Number of inode

Returns:

If the operation finished successfully.

NTSTATUS Ext2FsdRemoveEntry (IN PFSD_IRP_CONTEXT *IrpContext*, IN PFSD_VCB *Vcb*, IN PFSD_FCB *Dcb*, IN ULONG *Inode*)

Remove a directory entry.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

Dcb Directory control block

Inode Number of inode

Returns:

If the routine succeeds, it must return STATUS_SUCCESS. Otherwise, it must return one of the error status values defined in ntstatus.h.

BOOLEAN Ext2FsdSaveGroup (IN PFSD_IRP_CONTEXT *IrpContext*, IN PFSD_VCB *Vcb*)

Write group descriptor to the disk.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

Returns:

If the operation finished successfully.

BOOLEAN Ext2FsdSaveInode (IN PFSD_IRP_CONTEXT *IrpContext*, IN PFSD_VCB *Vcb*, IN ULONG *Inode*, IN inode_t * *inode*)

Write inode to the disk.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

Inode Number of inode

inode_t Pointer to the inode struct which has to be saved

Returns:

If the operation finished successfully.

BOOLEAN Ext2FsdSaveSuper (IN PFSD_IRP_CONTEXT *IrpContext*, IN PFSD_VCB *Vcb*)

Write super block to the disk.

Parameters:

IrpContext Pointer to the context of the IRP

Vcb Volume control block

Returns:

If the operation finished successfully.

11.2.9 super.c File Reference

Detailed Description

Implements functions to handle the ext2 superblock.

This file was not modified during this thesis.

```
#include "ext2.h"
#include "io.h"
#include "stdio.h"
#include "time.h"
```

11.2.10 write.c File Reference

Detailed Description

Functions to handle IRP_MJ_WRITE.

This file contains the functions needed for physically write down the data.

```
#include "ntifs.h"
#include "fsd.h"
```

Functions

- NTSTATUS FsdWrite (IN PFSD_IRP_CONTEXT IrpContext)

Dispatch Method for writing.

- NTSTATUS FsdWriteNormal (IN PFSD_IRP_CONTEXT IrpContext)

  Writes down the data to the disk.

- NTSTATUS FsdWriteComplete (IN PFSD_IRP_CONTEXT IrpContext)

  Completes the writing request for a cached file.

Function Documentation

NTSTATUS FsdWrite (IN PFSD_IRP_CONTEXT *IrpContext*)

Dispatch Method for writing.

This function calls FsdWriteNormal() when IRP_MN_NORMAL is set or FsdWriteComplete() when IPR_MN_COMPLETE is set.

Parameters:

IrpContext Pointer to the context of the IRP

Returns:

Status of called Function FsdWriteNormal() or FsdWriteComplete()

NTSTATUS FsdWriteComplete (IN PFSD_IRP_CONTEXT *IrpContext*)

Completes the writing request for a cached file.

Warning:

This function is not yet implemented.

It should call CcMdlWriteComplete() to free the memory descriptor lists.

Parameters:

IrpContext Pointer to the context of the IRP

Returns:

Status if the completiton of writing was successful.

NTSTATUS FsdWriteNormal (IN PFSD_IRP_CONTEXT *IrpContext*)

Writes down the data to the disk.

Warning:
This function is not yet implemented.

Parameters:
IrpContext Pointer to the context of the IRP

Returns:
Status if the data were writing down successfully.

Part IV

Tests

This part includes the test documentation.

Chapter 12

ExtFsr - Recognizer

12.1 Initial position

ExtFsr, the file system recognizer, is based on an existing ext2 recognizer. The driver consists of only two files with some lines of code which were already tested.

12.2 Development

After the hardest part, understanding the system and driver programming, it was not a big deal to decide if the partition contains a ext2 or an ext3 file system. This can be decided by an attribute of an already available structure.

12.3 Tests

Due to the simplicity of this new functionality to load different drivers depending on the file system, there was no need of a large test. The following tests were done successfully:

- Load the appropriate driver.
- Fallback to the ext2 driver if the ext3 driver is not present or failed to load.
- Unload the recognizer if both driver either are loaded or failed to load.

Chapter 13

Ext2Fsd - File system driver

13.1 Overview

During the development of the create and write routines the Windows bluescreens never were overcome. So the driver did not reach a state in which it would have been useful to make serious tests.

Serious tests in this context would be:

- Performance tests (number of files written to disk in a specific time)
- Stability tests which includes
  - Behaviour in case of a system crash
  - Behaviour when multiple application try to write simultaneously.
- Accuracy tests (does the driver any mistakes when processing create or write requests.

The only tests which has been done were debugging the driver step-by-step and watching the values of the variables until a bluescreen appeared. So it could be seen where the driver runs through and could be understood bit by bit what was executed on which request. This was very time consuming and unfortunately not as easy as debugging a user mode application.

Part V

Project Management

This part includes the Risk List, Project Plan, time evaluation, time variance comparison, list of used resources and personal statements.

Chapter 14

Introduction

14.1 Intention

This part covers the project and quality management of WinLFS. It is used to keep track of the nominal and actual time used for a task. Tasks are defined in the project plan in chapter 16 and nominal time estimation for this task can be found in the time variance comparison in chapter 17.

Additionally, in this part, personal reports and other personal statements can be found.

14.2 Process Model

At the beginning of this project, we planned to strictly use a prototype driven process model. The problem with this model was, we had no idea what a prototype for this project would look like. Therefore, we renamed the prototypes and called them phases. Our project is now split into different phases which we could determine and plan easier.

Chapter 15

Risk List

15.1 Team

Table 15.1: Team

Occurrence	$P[\%]$	**Consequence**	**Action**
Illness, accident.	1	Project conclusion at risk.	Not possible.
Lack of time because of exams.	30	Not kept milestone.	Good individual time planning.
Abort the study.	0.01	Project conclusion at risk.	Not possible.

15.2 Work

Table 15.2: Work

Occurrence	$P[\%]$	**Consequence**	**Action**
Developing tools are not suitable.	0.1	Can't finish the project.	Evaluate the tools well.
To much; need more time.	15	Can't finish the project.	Exactly plan the entire project and keep the milestones.

Gilding; spend to much time for one thing.	20	Can't finish the project.	Exactly specify goals and parts of them.

15.3 Goal

Table 15.3: Goal

Occurrence	$P[\%]$	**Consequence**	**Action**
Goal to complex/not reachable.	10	Not the full functionality.	Become acquainted with this matter early.
Not understand the technology:	10	Not the full functionality.	Make prototypes.
Underestimate complexity of the Windows architecture	40	Not all requirements can be finished.	Serious preparation for a better understanding.
Problems finding information concerning Windows XP drivers	20	Some tasks are very time consuming.	Try all information channels like IRC, newsgroups, etc.

15.4 Misc

Table 15.4: Misc

Occurrence	$P[\%]$	**Consequence**	**Action**
Data loss.	20	Can't finish the project.	Use CVS / make backups.

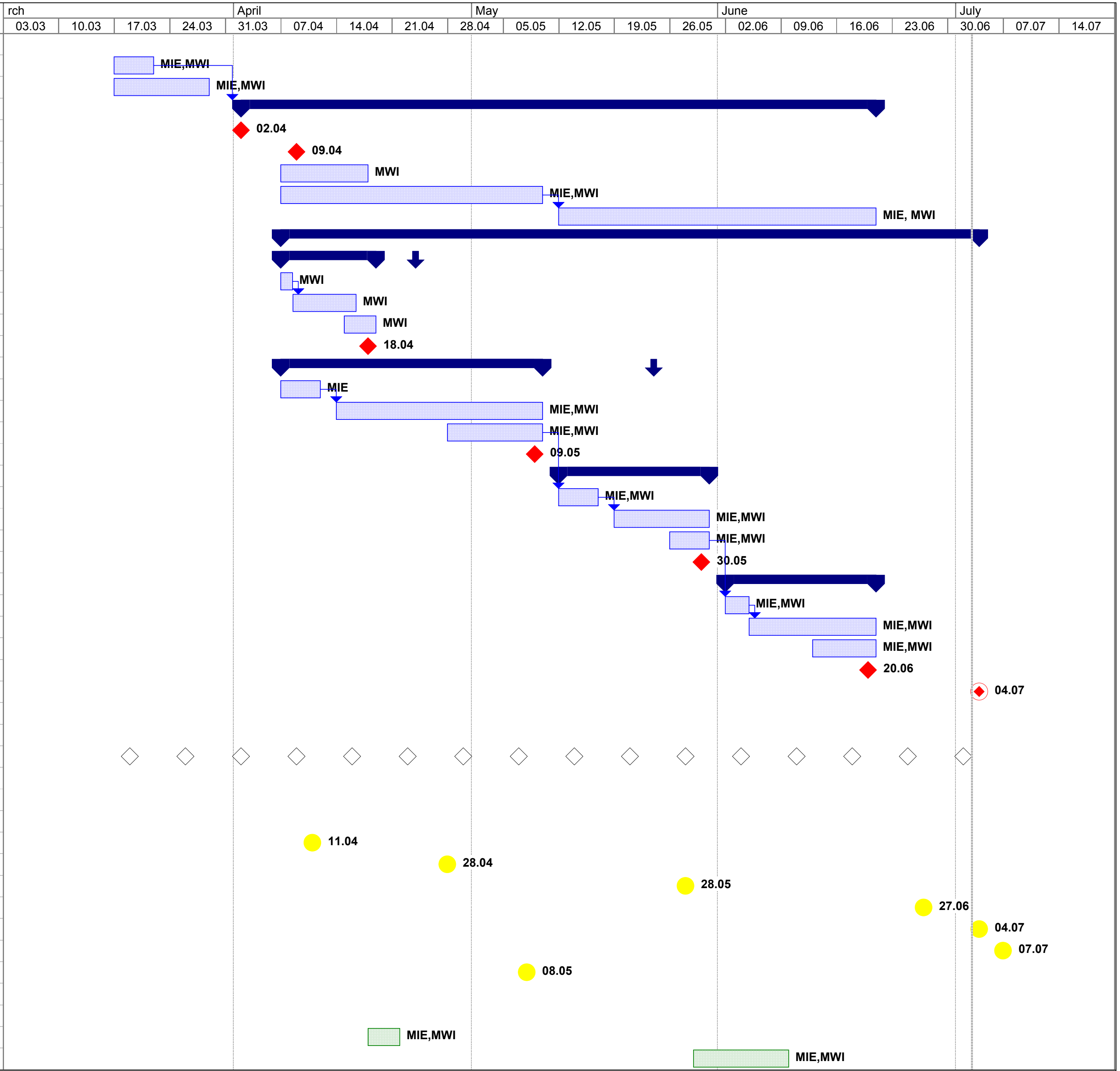
Chapter 16

Project Plan

16.1 Task description

The project plans are on the next pages. Tasks from the plan are described in table 16.1. On the next page is the first plan which was released in the second week after WinLFS had been started. The second plan is a revised version.

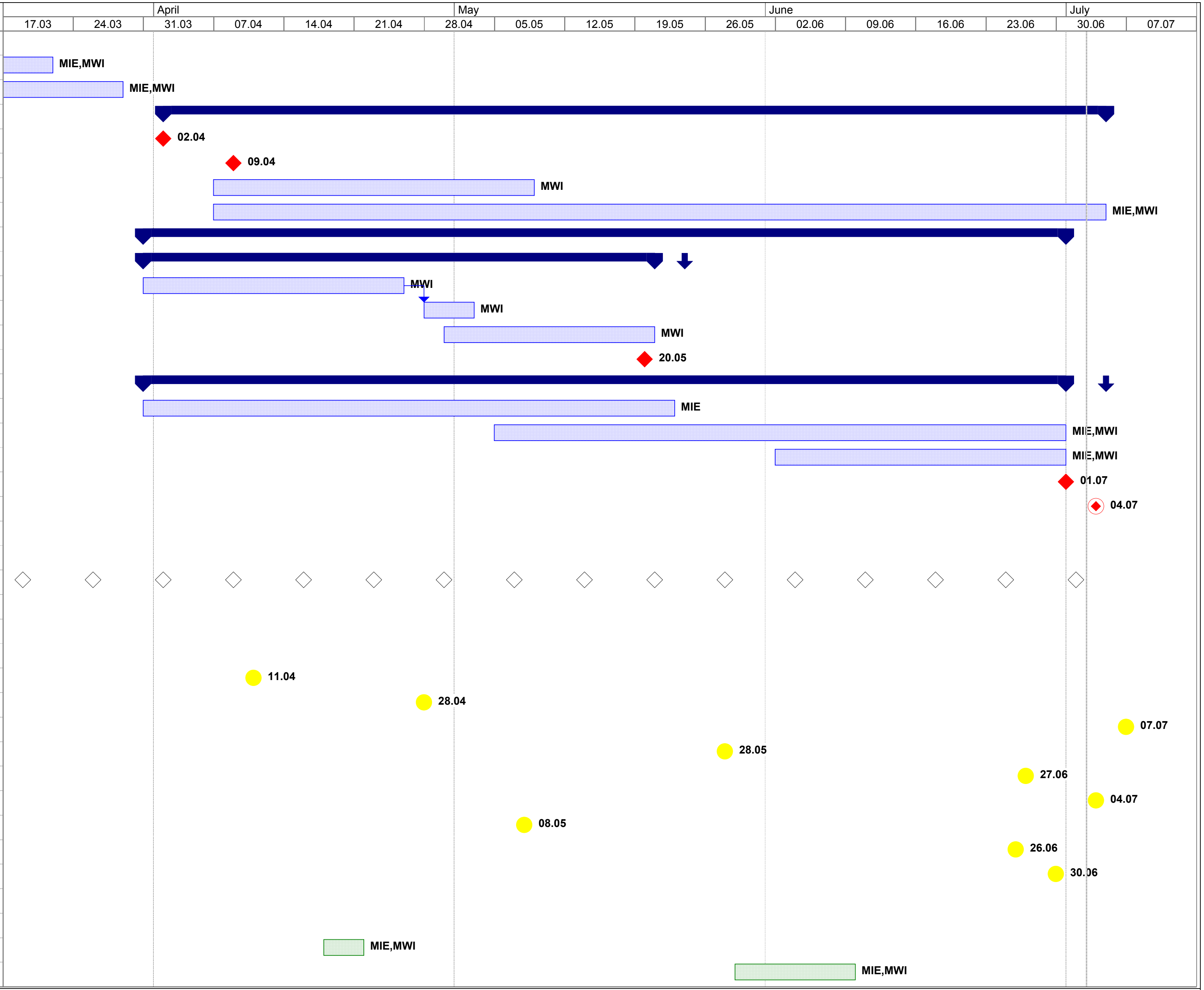
ID		Task Name	Duration	Start	Finish
1		**Tasks**	**70 days?**	**Mon 17.03.03**	**Fri 20.06.03**
2		IT Installation (HW, SW)	5 days?	Mon 17.03.03	Fri 21.03.03
3		Preparation work	10 days?	Mon 17.03.03	Fri 28.03.03
4		**Documentation**	**58 days?**	**Wed 02.04.03**	**Fri 20.06.03**
5		Project Plan	0 days	Wed 02.04.03	Wed 02.04.03
6		Requirements	0 days	Wed 09.04.03	Wed 09.04.03
7		Recognizer	9 days?	Mon 07.04.03	Thu 17.04.03
8		ext2	25 days?	Mon 07.04.03	Fri 09.05.03
9		ext3	30 days?	Mon 12.05.03	Fri 20.06.03
10		**Phases**	**64 days?**	**Mon 07.04.03**	**Fri 04.07.03**
11		**Recognizer**	**10 days?**	**Mon 07.04.03**	**Fri 18.04.03**
12		Analysis & Design	1.5 days?	Mon 07.04.03	Tue 08.04.03
13		Code	6 days?	Tue 08.04.03	Wed 16.04.03
14		Test	4 days?	Tue 15.04.03	Fri 18.04.03
15		finished	0 days	Fri 18.04.03	Fri 18.04.03
16		**ext2**	**25 days?**	**Mon 07.04.03**	**Fri 09.05.03**
17		Analysis & Design	5 days?	Mon 07.04.03	Fri 11.04.03
18		Code	20 days?	Mon 14.04.03	Fri 09.05.03
19		Test	10 days?	Mon 28.04.03	Fri 09.05.03
20		finished	0 days	Fri 09.05.03	Fri 09.05.03
21		**ext3 read**	**15 days?**	**Mon 12.05.03**	**Fri 30.05.03**
22		Analysis & Design	5 days?	Mon 12.05.03	Fri 16.05.03
23		Code	10 days?	Mon 19.05.03	Fri 30.05.03
24		Test	5 days?	Mon 26.05.03	Fri 30.05.03
25		finished ext3 read	0 days	Fri 30.05.03	Fri 30.05.03
26		**ext3 write**	**15 days?**	**Mon 02.06.03**	**Fri 20.06.03**
27		Analysis & Design	3 days?	Mon 02.06.03	Wed 04.06.03
28		Code	12 days?	Thu 05.06.03	Fri 20.06.03
29		Test	6 days?	Fri 13.06.03	Fri 20.06.03
30		finished ext3 write	0 days	Fri 20.06.03	Fri 20.06.03
31		final release	0 days	Fri 04.07.03	Fri 04.07.03
32					
33					
34		**Meetings**	**75 days**	**Wed 19.03.03**	**Wed 02.07.03**
51					
52					
53		**Exams**	**61 days**	**Fri 11.04.03**	**Mon 07.07.03**
54		Wissensbasierte Systeme	0 days	Fri 11.04.03	Fri 11.04.03
55		Java	0 days	Mon 28.04.03	Mon 28.04.03
56		Internetsicherheit	0 days	Wed 28.05.03	Wed 28.05.03
57		C#/.NET	0 days	Fri 27.06.03	Fri 27.06.03
58		Wissensbasierte Systeme	0 days	Fri 04.07.03	Fri 04.07.03
59		Java	0 days	Mon 07.07.03	Mon 07.07.03
60		Compilerbau	0 days	Thu 08.05.03	Thu 08.05.03
61					
62		**Holiday**	**37 days?**	**Fri 18.04.03**	**Mon 09.06.03**
63		Easter	2 days?	Fri 18.04.03	Mon 21.04.03
64		Pentecost	8 days?	Thu 29.05.03	Mon 09.06.03



Project: ProjectPlan
Date: Thu 03.07.03

Task	▭	Progress	▬	Summary	▬	External Tasks	▭
Split	┈┈┈	Milestone	◆	Project Summary	▬	External Milestone	◆
Deadline	⇩						

ID	ℹ	Task Name	Duration
1		**Tasks**	**80 days?**
2		IT Installation (HW, SW)	5 days?
3		Preparation work	10 days?
4		**Documentation**	**68 days?**
5	▦	Project Plan	0 days
6	▦	Requirements	0 days
7	▦	Recognizer	24 days?
8	▦	ext2	65 days
9		**Phases**	**66 days?**
10		**Recognizer**	**37 days?**
11	▦	Analysis & Design	20 days?
12		Code	5 days?
13	▦	Test	15 days?
14	▦	finished	0 days
15		**ext2**	**66 days?**
16	▦	Analysis & Design	39 days?
17	▦	Code	41 days?
18	▦	Test	21 days?
19	▦ ✎	finished	0 days
20	▦	final release	0 days
21			
22			
23	↻	**Meetings**	**75 days**
40			
41			
42		**Exams**	**61 days**
43	▦	Wissensbasierte Systeme	0 days
44	▦	Java 1	0 days
45	▦	Java 2	0 days
46	▦	Internetsicherheit	0 days
47	▦	C#/.NET	0 days
48	▦	Wissensbasierte Systeme	0 days
49	▦	Compilerbau 1	0 days
50	▦	Compilerbau 2	0 days
51	▦	E-Business	0 days
52			
53		**Holiday**	**37 days?**
54	▦	Easter	2 days?
55	▦	Pentecost	8 days?



Project: ProjectPlan
Date: Thu 03.07.03

Task	▭	Progress	▬	Summary	▬	External Tasks	▭	Deadline	⇩
Split	⋯	Milestone	◆	Project Summary	▬	External Milestone	◆		

Table 16.1: Task description

Task, Phase	**Description**
IT Installation (HW, SW)	• Windows XP installation. • Linux installation. • Setting up project server.
Preparation work	• Organizing related books and papers. • Reading for basic understanding. • Programming project homepage and time recording page. • Preparing LaTeX template.
Documentation	Includes making project plan and writing requirements which is general documentation. Additionally, each phase (recognizer, ext2, ext3) has his own technical documentation, including test reports.
Phases	Each phase (recognizer, ext2, ext3 read, ext3 write) is cut into 3 categories. In Analysis&Design, technical know how is acquired and decisions are made. Code and Test are self explaining.
Milestones	Each Phase has its own milestone. This is the date on which this phase should end and all related word should be done.

16.2 Fixed Dates

16.2.1 Meeting with Prof. E. Glatz

When	Wednesdays, 17.00
Who	Prof. E. Glatz, Marc Winiger, Michael Egli
What	Define weekly goals and check them.

16.2.2 Final Release

When	Friday, 04 July 2003
Who	Prof. E. Glatz, Marc Winiger, Michael Egli
What	Hand Over Final Release.

Chapter 17

Time Variance Comparison

17.1 Milestones

5 milestones were planned. Each marks the end of a phase or the end of the project respectively. First we needed to set up the environment properly. This includes installation of a Linux operation system and a Windows XP installation. Additionally, we made a ghost partition on a remote host for the likely case of destroying a partition. Gladly, this has never happened. This work and the finished version of the first project plan release was milestone 1. Milestone 2 was to define all requirements and write the respective documentation. This milestones could be planned easily and the work time was within the desired scope.

Milestone 3 marks the end of the file system recognizer and milestone 4 the end of the ext2 driver development. Milestone 5 marks the final release and is the end date of this project.

As you may encounter, the second project plan differs significantly from the first one. This is due a variety of challenges we had to get over. We started the work on the recognizer parallel to the file system driver, that's why both were affected. For more information about this, read section 18 further down.

17.2 Planning

The following table lists the estimated time for every phase and every task. We tried to estimate the needed time for every task on its own without taking into account the total amount of time for the whole project. After we had finished all estimations, we sum-up the time. The result was $445h$, which is what we had expected before. This time variance comparison is the basis for the project plan.

Table 17.1: Time variance comparison for all tasks

Task	**Milestone**	**Nominal** [h]	**Actual** [h]
IT Installation		**20**	**25**
Preparation work		**30**	**35**
Documentation		**115**	**135**
Project plan	1	10	20
Requirements	2	30	20
Recognizer		10	30
ext2		25	20
ext3		40	-
Project Management, etc.		-	45
Recognizer	3	**20**	**55**
Analysis & Design		5	25
Code		10	20
Test		5	10
ext2	4	**95**	**180**
Analysis & Design		20	40
Code		50	120
Test		25	20
ext3 read	removed	**70**	**-**
Analysis & Design		25	-
Code		30	-
Test		15	-
ext3 write	removed	**60**	**-**
Analysis & Design		15	-
Code		30	-
Test		15	-
Miscellaneous		**20**	**15**
Meetings		20	15

Fixed dates		**15**	**15**
Final release	5	15	15
Total		**445**	**460**

Chapter 18

Challenges and Problems

When we first had the idea of writing a file system driver for Windows, we could not determine the complexity of this task. We were completely inexperienced on this kind of software development. Furthermore, we were encouraged by the continuing studies diploma thesis where they wrote a read-only file system driver. We thought, it should easily be possible to finish a write function for ext2 and because this seemed not enough for a semester thesis, our objective was to include journalling support as well.

After we had finished collecting all information available and started reading, we realized that the problem might be more complex than we thought. First, we tried to understand the ext2 file system using a hexadecimal printout of a small partition and the book “Understanding the Linux Kernel” [1]. It took us about $1\frac{1}{2}$ weeks until we understood every bit on the partition.

After this, we tried to understand the basic concepts of a Windows driver. This turned out to be the hard part. Driver programming for Windows is very different from programming we have known and done before. To get some help on the design, we took the read-only driver mentioned before because this design had to be adopted and obeyed by our write extension. Soon we realized that these people from the continuing studies had the same problem we had. The problem is too complex to understand in such a short time. Even though, they could finish their read-only driver because they had a sample driver from which they could copy the code. Unfortunately this was impossible for our create and write dispatch routine. Although there was a driver from Matt Wu ([7]), we could not use his code because the basic design was completely different. Because we had to use the code from the read-only driver, we first had to understand the design and knew what already implemented functions were there for. This took us a long time and we never really finished this task because it was simply too much but we understood the parts which were fundamental for us.

Step for step, we tried to implement our functions (create and write). Because we could not copy anything from a sample driver, we had to write everything ourselves which is very time consuming, though very interesting.

All problems above lead to the result we achieved. The driver is not completely finished. Although this is kind of disappointing, we learned a lot. It is nearly impossible to complete a file system driver in this time without any sample driver which you can use as template except the programmer has wide experience in driver programming.

To conclude, we do not consider this work as unsuccessful as it might look like but rather we are glad we could make such a experience. It has enabled us to look more closely on how Windows works internally which can help us to conceive problems faster.

Chapter 19

Resources

Hardware	2 standard Siemens Scenic PC's (provided by HSR)
Operating System	Windows XP (SP1), Gentoo Linux 1.4-rc3
Tools	See tools list in Appendix A
Room	1.262

Chapter 20

Personal Reports

20.1 Michael Egli

Da ich mich allgemein sehr für Linux interessiere, fand ich das Projekt eine interessante Herausforderung. Interessant fand ich auch die Tatsache, dass das Projekt Einblick in einen Bereich gibt, mit dem sich verhältnismässig wenige Personen beschäftigen. Die Treiberprogrammierung unter Windows hat sich dann aber doch als komplizierter erwiesen, als wir zu Beginn dachten. Hinzu kam, dass es kaum brauchbare Dokumentationen zum Thema Dateisystemtreiber gibt. Trotzdem waren wir sehr motiviert und glaubten, die Ziele die wir uns steckten, erreichen zu können.

Nach und nach stellte sich heraus, dass es ein riesiger Zeitaufwand ist, bis man überhaupt die Konzepte der Treiberprogrammierung versteht. Da wir auch keinen Beispieltreiber hatten, an den wir uns halten konnten (den Treiber den es gibt passt nicht zum Design des bestehenden read-only Treibers), mussten wir alles selber programmieren. Das hat aus meiner Sicht den Vorteil, dass wir alles was mir machten verstehen mussten, aus der Sicht der Arbeit hat es aber den Nachteil, dass wir langsam voran kamen.

Auch wenn wir unsere Ziele teilweise nicht erreicht haben, war das Projekt für mich sehr lehrreich und ich glaube auch, dass die Ziele die wir uns gesteckt haben unrealistisch war. Die Ziele wären vielleicht dann realistisch, wenn jemand schon *sehr* viel Erfahrung mit Treiberprogrammierung hat, was bei uns nicht der Fall war.

Die Arbeit mit Marc wahr sehr angenehm und ich würde sie jederzeit wieder mit ihm machen. Auch als wir x-treme programming praktizierten hat es gut funktioniert. Meiner Meinung nach sogar besser und produktiver, als wenn jeder an seinem Stück Code arbeitete.

20.2 Marc Winiger

Da ich mich sehr für das Opensource Betriebssystem Linux interessiere, nebenbei jedoch meistens trotzdem noch ein Windows installiert habe, war ich sofort sehr begeistert vom Thema dieser Arbeit. Ich sah es als grosse Herausforderung, mich mit der Systemprogrammierung von Windows auseinander zusetzen. Allen Beteiligten, dem Team sowie dem Betreuer, war klar, dass es eine schwierige Aufgabe sein wird, jedoch waren wir sehr motiviert und glaubten daran, dass es möglich sein wird, das Ziel zu erreichen. Ich finde es sehr schade, dass es uns, trotz dem absehbaren sehr grossen Aufwand, von der Abteilung Informatik leider nicht erlaubt wurde, die Arbeit in einem Dreierteam anzugehen.

Die ersten zwei Wochen nach der Planung des Projekts war es meine Aufgabe einen File System Recognizer zu schreiben, der in der Lage ist, ein Ext2 oder Ext3 Dateisystem auf der Festplatte zu erkennen und diese auch voneinander zu unterscheiden, um dann den benötigten Dateisystem Treiber zu laden. Leider stellte sich sehr schnell heraus, dass die Treiberprogrammierung unter Windows ein sehr kompliziertes Thema ist und sehr viel Zeit benötigt wird, um sich einzuarbeiten. So konnten wir uns leider nicht auf die Implementation des Ext Dateisystems konzentrieren, was mich etwas mehr interessiert hätte. Stattdessen hatten wir mit der Windows Treiberprogrammierung zu kämpfen.

Nachdem ich einen bestehenden Recognizer soweit angepasst hatte, dass er unseren Anforderungen entsprach, widmete ich mich ebenfalls dem eigentlichen Dateisystem Treiber, in welchen sich Michael bereits eingearbeitet hatte. Wir entschieden uns die Arbeit von nun an nach dem Prinzip X-treme Programming fortzuführen. Aus meiner Sicht war dies die beste Lösung. Einerseits konnte ich mich so, im doch ziemlich umfangreichen Code, relativ schnell zurechtfinden. Andererseits haben wir auftretende Probleme und Fragen immer sofort diskutiert. Gerade dies war ein grosser Zeitvorteil, weil es sich vorwiegend um Probleme handelte, bei denen man alleine unter Umständen Stunden braucht, um sie zu lösen, der andere aber eventuell gleich eine gute Idee für einen Lösungsansatz hat.

Trotz dem teilweisen Misserfolg, war es für mich eine sehr interessante und lehrreiche Arbeit, die uns einen sehr tiefen Einblick in Funktionsweisen vom Windows System gab. Es wäre für mich eine persönliche Genugtuung gewesen, wenn wir es nur geschafft hätten eine leere Datei oder ein Verzeichnis zu schreiben. Leider hat uns Windows bis zum letzten Moment nur mit Bluescreens "belohnt".

Dass Michael mich als seinen Team-Partner auswählte, als wir erfahren hatten, dass die Arbeit nur als Zweierteam durchgeführt werden darf, hatte mich sehr gefreut, denn wir haben uns erst durch dieses Projekt näher kennen gelernt. Ich habe sehr gerne mit ihm zusammen gearbeitet und wir verstehen uns auch privat bestens, so dass wir manchmal nach getaner Arbeit die Rapperswiler “Wirtschaft(en)” etwas unterstützen konnten.

Appendix A

Tools

A.1 Developing

MS Visual Studio 6 Enterprise Edition (SP 5) , C/C++ IDE

MS IFS-DDK Kit 2600 , Driver Development Kit for installable file system drivers for Windows XP

MS MSDN 6.0a , Microsoft Developer Network Documentation

A.2 Versioning

Concurrent Versions System (CVS) 1.11.5 , CVS server on a Linux Server
http://www.cvshome.org/

TortoiseCVS 1.2.2 , CVS GUI with CVSNT 1.11.1.3 Client
http://www.tortoisecvs.org/

A.3 Documentation

TeXnicCenter 1 Beta 6.01 , really nice LaTeX-Editor
http://www.toolscenter.org/products/texniccenter/

MiKTeX-pdfTeX 2.2.1 (1.10a) , Typesetting tool with PDF output

Appendix B

Glossary

A

APC “Asynchronous Procedure Call”

B

Big-Endian refers to the most significant byte first order in which bytes of a multibyte value (such as a 32-bit dword value) are stored. For example a value of 0x0006FC7B would be stored in a file as: 0x00, 0x06, 0xFC, 0x7B. Many Motorola processors (Macintosh) use Big-Endian. The opposite byte ordering method is called Little Endian.

BDL “Block Description List”.

C

CCB “Context Control Block” represents an open instance of an on-disk object.

CDO “Control Device Objects” represent an entire file system.

D

DCB “Directory Control Block” is an internal NT file system structure in which a file system maintains state for an open instance of a directory file.

DDK Windows Driver Development Kit. Used for developing any drivers for Windows.

DPC “Deferred Procedure Call”

E

ext2/3 “Extended File System” is the file system often used on Linux system. Version 3 includes journalling.

F

FAT “File Allocation Table” is a file system generally used with Windows9x/Me and DOS.

FCB “File Control Block” represents an on-disk object in system memory.

FDO “Functional Device Object” is the root object of a storage device stack.

I

IFS “installable file system” is the possibility to load additional file system drivers in Windows XP.

IRP “I/O request packet“. An IRP is the basic I/O Manager structure used to communicate with drivers and to allow drivers to communicate with each other. Each IRP request has a major and a minor function.

IRP Major request Each major request can be mapped to a function to be executed on such a request. This function has to handle the minor request.

IRP Minor request is a sub-request. The major request decides on this value what there is to do.

IRQL “Interrupt Request Level”

ISR “Interrupt Service Routine”

L

LBN “Logical block number” identifies a physical block on a disk, using a logical address rather than physical disk values (for cylinder, track, and sector). For a disk with N blocks (in other words, sectors), the corresponding LBNs are numbered 0 through $(N - 1)$.

Link A hard- or soft-link on an ext2/3 partition.

Little-Endian refers to the least significant byte first order in which bytes of a multi-byte value (such as a 32-bit dword value) are stored. For example a value 0x0006FC7B would be stored in a file as: 0x7B, 0xFC, 0x06, 0x00. Intel processors (PC) use Little-Endian. The opposite byte ordering method is called Big Endian.

M

Magic number Identification bytes in the super block of a ext2 or ext3 partition.

MCB "Map control block" is a structure used by file systems in mapping the VBNs for a file to the corresponding LBNs on the disk.

MDL "Memory Descriptor List"

N

NTFS "New Technology File System" is a file system generally used with WindowsNT/2k/XP.

P

PDO "Physical Device Object" are the leaves of the tree with a FDO root object.

P

RtlXxx "Run-Time Library routines".

S

Super block Second block of the partition, right after the boot block. It contains all information about the partition.

V

VBN "Virtual Block Number" identifies a block (in other words, sector) relative to the start of a file. For a file with N blocks of data, the corresponding VBNs are numbered 1 through N.

VCB "Volume Control Block". An internal file system structure in which a file system maintains state about a mounted volume.

VDO "Volume Device Objects" represent mounted volumes.

VMM "Virtual Memory Manager"

VPB “Volume Parameter Block” creates a logical association between a physical disk device object and a logical volume device object.

Bibliography

[1] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, Sebastopol, CA 95472, 1 edition, 2000. ISBN 0-596-00002-2. 536 pp.

[2] Dave Hitz et al. Merging NT and UNIX Filesystem Permissions. URL http://www.usenix.org/publications/library/proceedings/lisa97/failsafe/usenix-nt98/full_papers/hitz/hitz_html/hitz.html.

[3] Manoj Joseph. Ext2 File System Driver for Windows NT. URL http://sourceforge.net/projects/winext2fsd.

[4] Daniel Lüönd, Daniel Zwirner, and Jürg Krebs. Ext2 Dateisystemtreiber für Windows XP, 2002. URL http://sourceforge.net/projects/ext2forxp/.

[5] John Newbigin. EXT2 IFS for Windows NT. URL http://uranus.it.swin.edu.au/~jn/linux/ext2ifs.htm.

[6] Richard Stallman et al. GNU Coding Standards. URL http://www.gnu.org/prep/standards_toc.html.

[7] Matt Wu. Ext2 File System Driver. URL http://sys.xiloo.com/projects/projects.htm#ext2fsd.