Caio Araújo Neponoceno de Lima

# A Virtual Filesystem Layer implementation in the XV6 Operating System

Salvador - BA, Brazil

2016

Caio Araújo Neponoceno de Lima

# A Virtual Filesystem Layer implementation in the XV6 Operating System

A monograph submitted to the Department of Computer Science of the Federal University of Bahia in partial fulfilment of the requirements for the Degree of Bachelor in Computer Science.

Federal University of Bahia – UFBA

Institute of Mathematics

Department of Computer Science

Advisor: Maurício Pamplona Segundo

Salvador - BA, Brazil

2016

# Resumo

Este trabalho está apresentando uma implementação de um *Virtual File System* (VFS) que permite mais de um sistema de arquivos coexistir no XV6. Como uma prova de conceito, nós também apresentamos uma implementação básica do sistema de arquivos EXT2, um dos sistemas de arquivos mais populares entre usuários Linux na década de 90. O principal objetivo deste trabalho é documentar uma fonte de conhecimento simples e prática sobre desenvolvimento de sistemas de arquivos para ambientes Unix. Com a simplicidade do XV6, esta implementação do VFS torna possível adicionar suporte a novos sistemas de arquivos para o mesmo com o mínimo de esforço.

**Keywords**: VFS. Operating Systems. Filesystems. Unix.

# Abstract

This work is presenting an implementation of a Virtual Filesystem Layer (VFS) to enable more than one filesystem coexist over the XV6. As a proof of concept, we also present a basic implementation of EXT2 filesystem, one of the most used filesystem by Linux users in 90's years. The main objective of this work is to document a simple and practical source of knowledge about filesystem development over an Unix-like environment. With the simplicity of XV6, this VFS implementation makes possible to add new filesystems support with minimal effort.

**Keywords**: VFS. Operating Systems. Filesystems. Unix.

# Contents

# 1 Introduction

Basic concepts of operating systems are extremely important to computer developers, mainly because the majority of software is developed to run as an application in an operating system. In this scenario, the operating system is a bottleneck in terms of performance, since it manages all computer resources, such as primary memory and storage access. Nowadays, we have access to cutting-edge commercial operating systems like GNU/Linux or BSD, but their source code are hard to understand. To solve this disparity between practical knowledge about operating system engineering and theoretical concepts, the MIT CSAIL Parallel and Distributed Operating Systems Group developed the XV6.

XV6 is a re-implementation of Dennis Ritchie and Ken Thompson's Unix Version 6 (v6) that runs on modern x86-based multiprocessors and is written in ANSI C. It was developed in the summer of 2006 for MIT's operating systems course, *6.828: Operating System Engineering*. Unlike Unix, Linux and other modern operating systems, XV6's source code is simple enough to be covered in one semester and also allows to cover the main concepts of Unix-like operating systems.

The XV6 is not the unique operating system for education puporses. The Minix, a microkernel-based operating system, was developed by Tanenbaum (2007) to apply the concepts covered in his book. In its third version, the primary objective shifted from education to the creation of a highly reliable and self-healing microkernel operating system, so it is possible to find modern concepts of operating systems since there are a lot of work and academic research being done on it. The biggest difference between XV6 and Minix 3 is in terms of design, since the XV6 is a monolithic kernel and Minix 3 is a microkernel. Unlike XV6, Minix 3 has an extensive code base and implements advanced concepts, such as network stack, which requires a detailed investigation before modifying the source code.

The filesystem[1] implemented in XV6 is a simpler version of original Unix v6 filesystem. However, its design is totally coupled with the rest of the XV6 code, difficulting tasks that require changes in this part of kernel, such as more than one filesystem mounted at the same time. Thus, it is not simple to introduce modern filesystem development techniques in this version of XV6.

This work presents an implementation of a Virtual Filesystem Layer (VFS) to enable having more than one filesystem coexisting over the XV6 and to discuss important concepts that influences modern filesystem design and implementation. As a proof of

[1] According to Tanenbaum (2007), a filesystem is an important kernel component used to control how data is stored and retrieved from disk or other medias. It is the component that organizes the access to data with "files" and "folders".

concept, we also present a basic implementation of the Second Extended Filesystem (EXT2), one of the most basic filesystems used by Linux users.

Our main objective is to document and present a simple and practical source of knowledge about filesystem development in Unix-like environments. Due to the simplicity of XV6, this VFS implementation makes possible to support new filesystems with minimal effort.

The remainder of this monograph is organized as follows. We will discuss some VFS implementations from different operating systems in Chapter 2. Chapter 3, we will present the necessary changes in XV6's original source code that were made to implement a VFS. In Chapter 4, we will show our EXT2 implementation for the XV6 VFS and a strategy to implement new filesystems. We show an experiment to verify the operability of our implementation in Chapter 5, followed by our conclusions and future works in Chapter 6.

# 2 Related Works

Before having multiple coexisting filesystems, the design of a filesystem was coupled with the kernel. This approach worked well in the past because there was only one filesystem type used by all Unix-like operating system implementation. When BSD Fast File System was created in thw University of Berkley, the necessity to support new filesystems finally became real. It was the first registry of a different filesystem layout since Unix become popular. After that, different filesystems were created, and, as consequence, an abstract layer to manipulate filesystems became a must have feature in commercial kernels. Some examples of this abstract layer are the File System Switch (FSS) in Unix System V Revision 3.0, the Sun VFS/Vnode in SunOS, and the Linux VFS layer, which are discussed below.

## 2.1 File System Switch

In order to solve the problem of multiple filesystems coexisting in the kernel, the File System Switch (FSS) was introduced in the Unix System V Revision 3.0 (PATE; BOSCH, 2003). As stated in this work, *"one of the main goals of SVR3 was to provide a framework under which multiple different filesystem types could coexist at the same time. Thus each time a call is made to mount, the caller could specify the filesystem type."*

The major difference between the filesystem-independent layer of the kernel and the filesystem-dependent layer is the inclusion of an in-core *inode*[1]. The FSS implementation of the in-core inode contains fields that are abstract among different filesystem types (e.g. user and group IDs, file size and access permission), as well as the ability to reference data that is filesystem-specific (PATE; BOSCH, 2003). This way, each filesystem type can have a very different on-disk representation of its metadata/data and still be manipulated by the FSS layer.

The FSS design implements the abstract behavior over filesystem operations through a structure called *fstypsw*, where all filesystem-specific operations are defined. To support multiple filesystems, the kernel maintains a global array of this structure, where each entry represents a possible filesystem. Then, when a system call to access a file is made, the *inode* representing it in memory only needs to have an index of the correct filesystem to access the list of adequate operations. Filesystem-specific operations used by FSS are described in Table 1. As may be observed, the operations in this list are related to a group

[1] According to Tanenbaum (2007), an inode is the representation of a filesystem file used internally by Unix-like kernels to manipulate its information. There are two majors types of inode: the on-disk inode, which is the inode stored in disk, and the in-core inode, which is the representation of an on-disk inode in memory.

Table 1: List of operations for the File System Switch, as presented by Rodeh, Bacik and Mason (2013)

| **FSS Operation** | **Description** |
|---|---|
| fs_init | Each filesystem can specify a function that is called during kernel initialization allowing the filesystem to perform any initialization tasks prior the first *mount* call |
| fs_iread | Read the inode (during pathname resolution) |
| fs_iput | Release the inode |
| fs_iupdat | Update the inode timestamps |
| fs_readi | Called to read data from file |
| fs_writei | Called to write data to a file |
| fs_itrunc | Truncate a file |
| fs_statf | Return file information required by *stat()* |
| fs_namei | Called during pathname transversal |
| fs_mount | Called to mount a filesystem |
| fs_umount | Called to unmount a filesystem |
| fs_getinode | Allocate a file for a pipe |
| fs_openi | Call the device open routine |
| fs_closei | Call the devic close routine |
| fs_update | Sync the superblock to disk |
| fs_statfs | Used by *statfs()* and *ustat()* |
| fs_access | Check access permissions |
| fs_getdents | Read directory entries |
| fs_allocmap | Build a block list map for demanding page |
| fs_freemap | Frees the demand paging block list map |
| fs_readmap | Read a page using the block list map |
| fs_setattr | Set file attributes |
| fs_notify | Notify the filesystem when the file attributes change |
| fs_fcntl | Handle the *fcntl()* system call |
| fs_fsinfo | Return filesystem-specific information |
| fs_ioctl | Called in response to *ioctl()* system call |

of Unix-like system calls (e.g *mount*, *umount*, *statf* or *ioctl*) and their sub-procedures (e.g *fs_namei* or *fs_access*).

The FSS architecture is the first practical work to support multiple filesystems on Unix. The following architectures presented in this work were widely influenced by it, with some variations on layer organization or filesystem-dependent functions.

## 2.2 The Sun VFS/Vnode Architecture

This VFS implementation was developed for the Sun Microsystem's SunOS (KLEIMAN, 1986) and is grounded on the four following goals:

1. *The filesystem implementation should be clearly split into a filesystem independent*

*and filesystem-dependent layer. The interface between the two should be well defined.*

2. *It should support local disk filesystems such as the 4.2BSD Fast File System (FSS), non-UNIX like filesystems such as MS-DOS, stateless filesystems such as NFS, and stateful filesystems such as RFS.*

3. *It should be able to support the server side of remote filesystems such as NFS and RFS.*

4. *Filesystem operations across the interface should be atomic such that several operations do not need to be encompassed by locks.*

The main difference between FSS and VFS/Vnode is that the VFS/Vnode architecture is based on two majors structures, the *vfsops*, the interface for abstract filesystem operations, and the *vnops*, the interface to enable individual file operations. Figure 1 shows a diagram of the VFS/Vnode architecture, which describes a high level interation between SunOS' components. As may be seen, the VFS/VOP/veneer layer, which is the VFS/Vnode layer, separates other kernel components from supported filesystems. In addition, this diagram shows that the abstract layer can handle diskless filesystems (e.g Network File System (NFS), a network-based filesystem).

Since one goal of this architecture is to support diskless and non-Unix filesystems, the in-core *inode*, also named as *vnode*, stores only the common data among all filesystems, like the FSS in-core *inode* presented before. This structure stores all information required by the filesystem-independent layer and stores a pointer to a private data used by the filesystem-dependent layer.

The VFS layer (represented by the *vfsops*) is responsible to store all operations related to the specific filesystem. The set of operations stored by the VFS layer is described in Table 2.

The Vnode Layer (represented by *vnops*) is where all operations over files are stored. This set of operations is described in Table 3.

Table 3: List of *vnops* operations, as presented by Pate and Bosch (2003).

| **VNode Operation** | **Description** |
|---|---|
| vop_open | This function is only applicable to device special files, files in the namespace that represent hardware devices. It is called once the vnode has been returned from a prior call to voplookup |

| | |
|---|---|
| vop_close | This function is only applicable to device special files. It is called once the vnode has been returned from a prior call to voplookup. |
| vop_rdwr | Called to read from or write to a file. The information about the I/O is passed through the uio structure. |
| vop_ioctl | This call invokes an ioctl on the file, a function that can be passed to device drivers. |
| vop_select | This vnodeop implements select(). |
| vop_getattr | Called in response to system calls such as stat(), this vnodeop fills in a vattr structure, which can be returned to the caller via the stat structure. |
| vop_setattr | Also using the vattr structure, this vnodeop allows the caller to set various file attributes such as the file size, mode, user ID, group ID, and file times. |
| vop_access | This vnodeop allows the caller to check the file for read, write, and execute permissions. A cred structure that is passed to this function holds the credentials of the caller. |
| voplookup | This function replaces part of the old namei() implementation. It takes a directory vnode and a component name and returns the vnode for the component within the directory. |
| vop_create | This function creates a new file in the specified directory vnode. The file properties are passed in a vattr structure. |
| vop_remove | This function removes a directory entry. |
| vop_link | This function implements the link() system call. |
| vop_rename | This function implements the rename () system call. |
| vop_mkdir | This function implements the mkdir() system call. |
| vop_rmdir | This function implements the rmdir() system call. |
| vop_readdir | This function reads directory entries from the specified directory vnode. It is called in response to the getdents() system call. |
| vop_symlink | This function implements the symlink() system call. |
| vop_readlink | This function reads the contents of the symbolic link. |
| vop_fsync | This function flushes any modified file data in memory to disk. It is called in response to an fsync() system call. |
| vop_inactive | This function is called when the filesystem-independent layer of the kernel releases its last hold on the vnode. The filesystem can then free the vnode. |

| | |
|---|---|
| vop_bmap | This function is used for demand paging so that the virtual memory (VM) subsystem can map logical file offsets to physical disk offsets. |
| vop_strategy | This vnodeop is used by the VM and buffer cache layers to read blocks of a file into memory following a previous call to vop_bmap(). |
| vop_bread | This function reads a logical block from the specified vnode and returns a buffer from the buffer cache that references the data. |
| vop_brelse | This function releases the buffer returned by a previous call to vop_bread. |

Our XV6 VFS architecture was strongly influenciated by this implementation because of its simplicity and flexibility to support non-Unix and diskless filesystems.

## 2.3 The Linux VFS Layer

The Linux VFS implementation is one of the most successful in terms of design, mainly because of its complex requirements of portability and performance. A family of data structures represents the abstract file model. The Linux implementation also follows the design of generic structures for the data required by a filesystem-independent layer and maintain data and pointers to filesystem-dependent information.

The four primary structures types of Linux VFS are:

- *superblock*: This structure stores a superblock of a mounted filesystem and contains the global filesystem's metadata;
- *inode*: This structure stores a file and its metadata (e.g. access permissions);
- *dentry*: This structure represents an directory entry, which is a single component of a file;
- *file*: This structure represents an open file attached to a running process.

There is a pointer to a group of operations on each structure presented, which stores the filesystem-dependent functions. These structures describe methods called by the kernel when the filesystem-dependent function need to be executed:

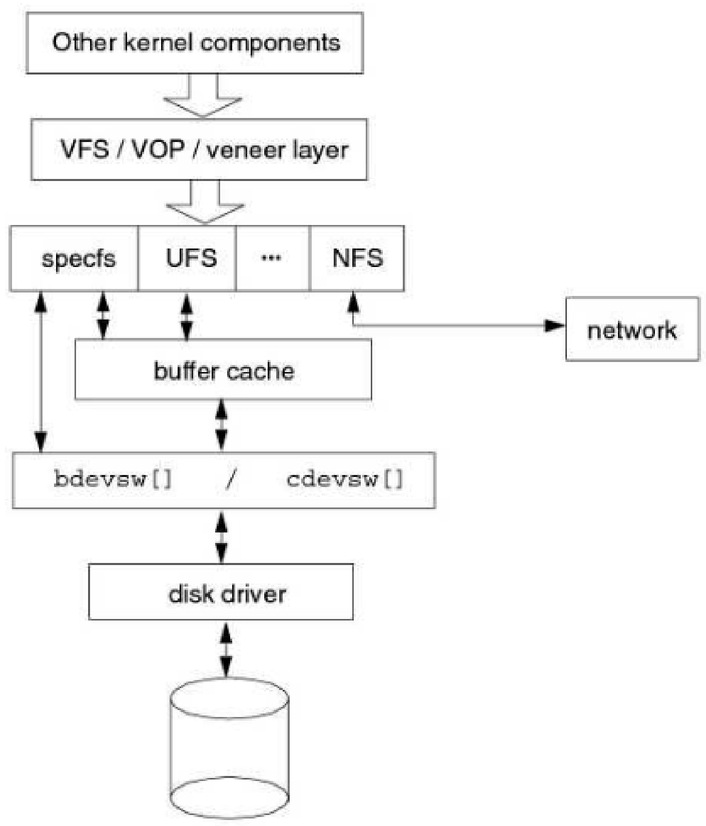


Figure 1: SunOS VFS/Vnode architecture, extracted from Pate and Bosch (2003)

- *super_operations*: contains methods the kernel invokes on a specific filesystem, such as write_inode().
- *inode_operations*: contains methods to operate over filesystem *inodes*, such as *create()* or *mkdir()*.
- *dentry_operations*: contains operations to manipulate a specific directory entry, such as *d_compare()*.
- *file_operations*: contains operations to manipulate opened files, such as *open()* and *close()*.

These structures and their operations are much more complex than previous presented VFSs. If you are interested in seeing what abstract information is covered by the Linux VFS, we recommend reading the Robert Love's work(LOVE, 2010).

The major advantage of Linux VFS is its constant evolution, because it is the biggest open source project in the world. Thus, there are some features presented in this VFS implementation that were not implemented in others such as *page cache* utilization.

Table 2: List of *vfsops* operations table, as presented by Pate and Bosch (2003)

| **VFS Operation** | **Description** |
|---|---|
| vfs_mount | This function is called to mount the filesystem correctly. |
| vfs_unmount | This function is called to unmount a filesystem. |
| vfs_root | This function returns the root vnode for this filesystem and is called during pathname resolution. |
| vfs_statfs | This function returns filesystem-specific information in response to the statfs() system call. This is used by commands such as df. |
| vfs_sync | This function flushes file data and filesystem structural data to disk, which provides a level of filesystem hardening by minimizing data loss in the event of a system crash. |
| vfs_fid | This function is used by NFS to construct a file handle for a specified vnode. |
| vfs_vget | This function is used by NFS to convert a file handle returned by a previous call to vfs_f id into a vnode on which further operations can be performed. |

# 3 XV6 VFS Implementation

In this section, we present our VFS implementation over XV6. Figure 2 shows the XV6 VFS architecture and it is possible to notice notice the influence of SunOS VFS/Vnode architecture. This is main contribution of this work, and almost everything that will be discussed later will refer the concepts and designs presented here.

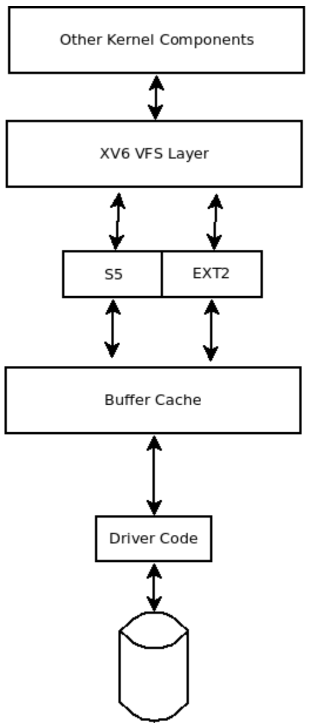


Figure 2: XV6 VFS architecture

## 3.1 XV6 VFS main structures

Our design for the XV6 VFS was strongly influenced by the SunOS VFS/Vnode (KLEIMAN, 1986) architecture presented in Section 2.2. There are three major structures to handle filesystem-independent information: *inode*, *superblock* and *filesystem_type*. Details of the implementation of these structures are given bellow.

### 3.1.1 inode

The *inode* structure is responsible for representing files (e.g. regular files and directories). Filesystem-related system calls that manipulate the *inode* structure has no knowledge of the filesystem type in use. Our *inode* structure is presented on Listing 3.1.

Listing 3.1: struct inode

```
// in-memory copy of an inode
struct inode {
  uint dev;                      // Minor Device number
  uint inum;                     // Inode number
  int ref;                       // Reference count
  int flags;                     // I_BUSY, I_VALID
  struct filesystem_type *fs_t;
  struct inode_operations *iops; // Specific inode operations
  void *i_private; // Filesystem-dependent information

  short type;            // File type
  short major;           // Major device number (T_DEV only)
  short minor;           // Minor device number (T_DEV only)
  short nlink;           // Number of links to inode in file system
  uint size;             // Size of file (bytes)
};
```

The meaning of each variable in Listing 3.1 is presented on Table 4.

### 3.1.2 superblock

The *superblock* is the structure responsible for storing filesystem metadata information like total filesystem size, available amount of storage and block size. Filesystem-related system calls manipulate the *superblock* as an abstract representation of the filesystem-independent structure. Different from the Linux *superblock* implementation, that contains much more fields, XV6's *superblock* stores only necessary information to manipulate basic filesystem operations like inode and block allocation, as presented in Listing 3.2.

Listing 3.2: struct superblock

```
struct superblock {
  // Driver major number of block device this superblock is stored in.
  int major;
  // Driver major number of block device this superblock is stored in.
  int minor;
  // Block size of this superblock
  uint blocksize;
  // Filesystem-specific info
  void *fs_info;
  unsigned char s_blocksize_bits;
  // Superblock Falgs to map its usage
  int flags;
```

Table 4: Description of *inode* variables.

| **Inode field** | **Description** |
|---|---|
| dev | Stores the minor device number. In commercial operating systems, like Linux, it is prudent to store a structure representing the block device where this *inode* is stored on. We decided to store the minor device number because the XV6 does not have a block device structure, since it is planned to support only one type of block device (i.e. IDE hard disk). |
| inum | Stores the *inode* number, which is the identity of the *inode* used by all operations that manage it. It is heavily used by the *inode* cache. |
| ref | Tracks if this *inode* is in use. If its value is 0, the *inode* cache will reuse the space to store information of a new *inode*. |
| flags | Stores flags used internally by the VFS algorithms. |
| fs_t | Is a pointer to fast access filesystem-dependent operations. |
| iops | Points to filesystem-specific operations used by this *inode*. |
| i_private | Stores filesystem-specific data, and usually points to an filesystem-specific *inode* representation (e.g. "struct ext2_inode*" for an EXT2 filesystem). Its data is used on filesystem-specific functions (e.g. "int ext2_writei(struct inode *ip, char *src, uint off, uint n)" to write data into an *inode*). |
| type | Stores the file type, which can be T_DIR, T_FILE, T_DEV and T_MOUNT |
| major | Stores the device major number when file type equals T_DEV. |
| minor | Stores the device minor number when file type equals T_DEV. |
| nlink | Number of links pointing to this *inode*. |
| size | Stores the file size. |

```
};
```

Variables in Listing 3.2 are described with more details on Table 5.

### 3.1.3 *filesystem_type*

The XV6 VFS layer keeps a registration list containing all supported filesystems. To store it into the kernel in an organized way, there is a structure named *filesystem_type* that is responsible for storing important data about a filesystem, such as name, *inode* operations and global operations. This structure is defined in *src/vfs.h* and is shown in Listing 3.3.

Table 5: Description of *superblock* variables.

| **Superblock field** | **Description** |
|---|---|
| major | It is the block device major identifier. It is used internally by the kernel to correctly map what driver is used to access the block device this superblock is stored in. |
| minor | It is the block device minor identifier. It is used internally by the block device driver to correctly map the correct device this superblock is stored in. |
| blocksize | It is used to inform the kernel what is the size of the block for this filesystem. |
| fs_info | It is a generic pointer used to store the filesystem-specific information. |
| s_blocksize_bits | It is also used to inform the kernel what is the size of the block for this filesystem, but it is used by bitwise operations. |
| flags | It is used to store flags to internal kernel control. |

Listing 3.3: struct filesystem_type

```c
struct filesystem_type {
  char *name;
  struct vfs_operations *ops;
  struct inode_operations *iops;
  struct list_head fs_list;
};
```

There is a list pointing to other registered filesystem. To manipulate this list, there are two helper functions used internally by the kernel, both shown in Listing 3.4.

Listing 3.4: List of helper functions for filesystem_type

```c
int register_fs(struct filesystem_type *fs);
struct filesystem_type* getfs(const char *fs_name);
```

The *register_fs()* is used to install a new filesystem into the internal kernel filesystem list. The *getfs()* is used to retrieve a filesystem type named as *fs_name* (see the *mount* system call in Listing 3.10).

## 3.2 Filesystem-specific operations

Our XV6 VFS implementation offers an interface between the implemented filesystem and the kernel code through the use of two structures, named *vfs_operations* and *inode_operations*.

### 3.2.1 *vfs_operations*

This structure stores operations that affect the entire filesystem, and after almost every operations, there is a change over the state of the filesystem. This structure is a list of function pointers and is one of the main components that make the kernel use filesystem operations with a satisfactory abstraction level. It is stored in a structure that manages general filesystem information, such as the *filesystem_type* in Section 3.1.3 or the *inode* in Section 3.1.1.

The operations stored by this structure are presented in Listing 3.5 and detailed in Table 6.

Listing 3.5: struct *vfs_operations*

```

struct vfs_operations {
  int           (*fs_init)(void);
  int           (*mount)(struct inode *devi, struct inode *ip);
  int           (*unmount)(struct inode *);
  struct inode* (*getroot)(int, int);
  void          (*readsb)(int dev, struct superblock *sb);
  struct inode* (*ialloc)(uint dev, short type);
  uint          (*balloc)(uint dev);
  void          (*bzero)(int dev, int bno);
  void          (*bfree)(int dev, uint b);
  void          (*brelse)(struct buf *b);
  void          (*bwrite)(struct buf *b);
  struct buf*   (*bread)(uint dev, uint blockno);
  int           (*namecmp)(const char *s, const char *t);
};
```

Table 6: Description of *vfs_operations* functions.

| **vfs_operations field** | **Description** |
|---|---|
| fs_init(void) | This operation is called when the filesystem is being loaded by the kernel (i.e. when the kernel is bootstrapping itself). Its main purpose is to provide an interface where filesystem developers can initialize data structures internally used by their code. |

| | |
|---|---|
| mount(struct inode *devi, struct inode *ip) | This operation is almost self-explained. This function is called when a new instance of the filesystem is being mounted. Usually, a lot of operations are handled in a mount operation, such as read the *superblock*, initialize device structures like filesystem logger, and add an entry on *mount_table*. Function parameters are the device inode being mounted (*devi*) and the directory where the filesystem is going to be mounted (ip). |
| unmount(struct inode *ip) | This operation is called when the filesystem is going to be unmounted. The parameter *ip* is the directory being unmounted. |
| getroot(int major, int minor) | This operation is responsible to allocate, read, fill ande return the necessary information of the root *inode*, including the filesystem-specific information. The parameters *major* and *minor* are used to read the root *inode* from the correct device. |
| readsb(int dev, struct superblock *sb) | This operation reads the *superblock* from the device and store it on memory to be manipulated by the kernel. The parameter *dev* indicate the correct device and *sb* is a pointer to an already allocated superblock structure. In this version, the implementation is using only the *dev* as parameter to identify what is the block device the *superblock* is being read, because we are considering that the XV6 is hardcoded to support only one type of block device. To support more than one type of block device, this parameter shall be a *struct block_device*. |
| ialloc(uint dev, short type) | This operation allocates new *inodes* for this filesystem. This function searches for a free inode in the filesystem (sometimes there is an *inode* table) and return it to the kernel. When the *inode* is allocated, the disk is updated to avoid double allocation and data losses. The parameter *dev* indicates which device this *inode* is located and *type* is the *inode* type (e.g. directory, regular file, device file). |
| balloc(uint dev) | This operation allocates a block. This function searches for an available block in the filesystem, set the information to avoid double allocation, and return the block number. The parameter *dev* is the device identifier. |

| | |
|---|---|
| bzero(uint dev, int bno) | This operation is used to fill and write the block *bno* from *dev* device with zero and persist this information. |
| bfree(uint dev, uint b) | This function is the opposite of *balloc*, where a block *b* from the device *dev* is released to be reused in the future. This function is usually called by system calls thaat delete files or directories such as *unlink* and *rmdir*. |
| brelse(struct buf *b) | This operation is used by the kernel to release the buffer *b* and make its space available to other process that are trying to get a buffer cache. This function is used to let the filesystem layer handle buffer release operations, but it is usually set as the default internal *brelse* function. |
| bwrite(struct buf *b) | This operation writes the content stored on buffer *b* in the block device. The purpose of this operation is to let filesystem developers handle the *bwrite* operation and implement custom behaviors. However, the default internal *bwrite* function is usually used. |
| bread(uint dev, uint blockno) | This operation is used by the kernel to reaad the block with number *blockno* from the device *dev*. The purpose of this operation is to enable filesystem developers handle the *bread* operation and implement custom behaviors. Usually, the default internal bwrite function is used. |
| namecmp(const char *s, const char *t) | This operation defines how the way directory entries' names will be compared. For example, there are some filesystem with case-sensitive and non-case-sensitive rules. |

### 3.2.2 inode_operations

This structure is the main glue between the filesystem code and *inode* operations performed by the kernel. Its operations manipulate the filesystem-dependent *inode* information and return messages of success or failure to the abstract kernel code (i.e. normally filesystem-related system calls). Operations like reading or writing to a file, reading directory entries from a folder and directory lookup are stored in *inode_operations*. This structure is presented in Listing 3.6 and its operations are detailed in Table 7.

Listing 3.6: struct *inode_operations*

```
struct inode_operations {
  struct inode* (*dirlookup)(struct inode *dp, char *name, uint *off);
  void (*iupdate)(struct inode *ip);
  void (*itrunc)(struct inode *ip);
  void (*cleanup)(struct inode *ip);
  uint (*bmap)(struct inode *ip, uint bn);
  void (*ilock)(struct inode* ip);
  void (*iunlock)(struct inode* ip);
  void (*stati)(struct inode *ip, struct stat *st);
  int (*readi)(struct inode *ip, char *dst, uint off, uint n);
  int (*writei)(struct inode *ip, char *src, uint off, uint n);
  int (*dirlink)(struct inode *dp, char *name, uint inum, uint type);
  int (*unlink)(struct inode *dp, uint off);
  int (*isdirempty)(struct inode *dp);
};
```

Table 7: Description of *vfs_operations* fuctions.

| **inode_operations field** | **Description** |
|---|---|
| dirlookup(struct inode *dp, char *name, uint *off) | This operation has a big role in making the way kernel manages *inodes* highly abstract. This function verifies if a directory entry named *name* is a child of the folder *dp*. If this entry exists, this function reads and returns the corresponding *inode* and set the parameter *off* with the byte offset of the entry in the *dp* data. The internal path to *inode* translator kernel function *namex()*, presented in Listing 3.13, is the main caller of this operation. |
| iupdate(struct inode *ip) | This function is the implementation of the *inode* update operation performed by kernel in arbitrary system calls (e.g. *mkdir*). It updates the in-disk *inode* with the data stored in the in-memory *inode ip*. The pointer *inode->i_private* is used to store the filesystem-dependent information to be written on disk. |
| itrunc(struct inode *ip) | This operation cleans all the information stored in an *inode ip*. |
| cleanup(struct inode *ip) | This operation is called by the kernel when an *inode* is being released, because there is no reference to it. Filesystems should free the *inode* and its blocks, and optionally, but strongly recommended, erase its content. |

| | |
|---|---|
| bmap(struct inode *ip, uint bn) | This is one of the most important functions used by the VFS layer, because it translates the block *bn* of the *inode ip* into the block number in the filesystem. If there is no allocated block for *bn*, the *bmap* function allocates one block and returns it. The return value is the filesystem block number. |
| ilock(struct inode* ip) | This function is called by the kernel VFS layer when it is necessary to lock the access to the *inode ip*. If the *inode ip* is already locked, the caller process sleeps and waits until *ip* becomes available. It is important to the kernel synchronization mechanism. |
| iunlock(struct inode* ip) | It is the opposite of *ilock*. This function unlocks the *inode ip* and wakes up all process waiting for *ip*. As almost all unlock functions do the same operations, there is an internal kernel function called *generic_iunlock* that can be used instead. |
| stati(struct inode *ip, struct stat *st)) | This operation is called by the *stati* system call and is responsible for filling the parameter *st* with information from the *inode ip*. |
| readi(struct inode *ip, char *dst, uint off, uint n) | This function implements the *read* system call and is internally used by the kernel when is necessary to read a directory entry. The *read* operation tranfers *n* bytes from *inode ip* starting from the byte offset *off* to the *dst* buffer. The function returns the number of bytes read. |
| writei(struct inode *ip, char *src, uint off, uint n) | This function implements the *write* system call and is internally used by the kernel when is necessary to write to a directory entry. The write operation is performed on *inode ip* starting from the byte offset *off*, where *n* bytes from the buffer *src* will be written. The function returns the number of bytes written. |
| dirlink(struct inode *dp, char *name, uint inum, uint type) | This function is called by *mkdir* and *creat* system calls. Its purpose is to add the *inode inum* of type *type* to the directories map *dp* with name *name*. It keeps the filesystem's hierarchical structure updated. |
| unlink(struct inode *dp, uint off) | This function is called by *rmdir* and *rm* system calls. Its objective is to remove the directory entry located starting in the offset byte *off* and freeing it from the inode *dp*. |

| | |
|---|---|
| isdirempty(struct inode *dp) | This function checks if the directory *dp* does not contain directory entries. |

## 3.3 The *mount* system call

The filesystem hierarchy is the interface that a process uses to access files. This abstraction is powerful because application developers do not need to think or even know how its application data will be stored in a block device.

The XV6, as an Unix-like operating system, implements file access through filesystem hierarchy and originally supports only one block device. Being able attach more than one block device is not a VFS feature, but without it VFS would not be useful enough. So, it was necessary to implement a prototype *mount* system call to support multiple block devices.

The *mount* is the operation used to attach a new block device to the filesystem hierarchy. This way, application developers have a high level of abstraction to manipulate data from one block device to another. The XV6 *mount* system call is defined as:

*int **mount**(char *special_device_file, char *mount_point_directory, char *filesystem_type)*

where *special_device_file* is the file specifying which disk will be mounted, *mount_point_directory* is the directory where the new filesystem will be mounted in, and *filesystem_type* is a valid and supported type of the filesystem to be mounted. Different from our implementation, commercial operating systems also offer an interface to pass options to *mount* operations, such as *read-only* or *no-recovery* flags.

To exemplify the *mount* operation, lets consider we have a block device called */dev/hdc* with an EXT2 filesystem, and we want to mount it on */mnt* directory. To do that, we should make the following call:

***mount**("/dev/hdc", "/mnt", "ext2")*

After the *mount* call, it is possible to access the */dev/hdc* filesystem tree through */mnt*. Figure 3 shows how the filesystem hierarchy will look like after the mount operation.

### 3.3.1 Mount table

When the operating system supports the mount operation, some data-structures and changes over methods that translate path names into *inodes* (i.e. *namex* function in Listing 3.13) are required because there are cases where the path translation needs to cross *mount* points.

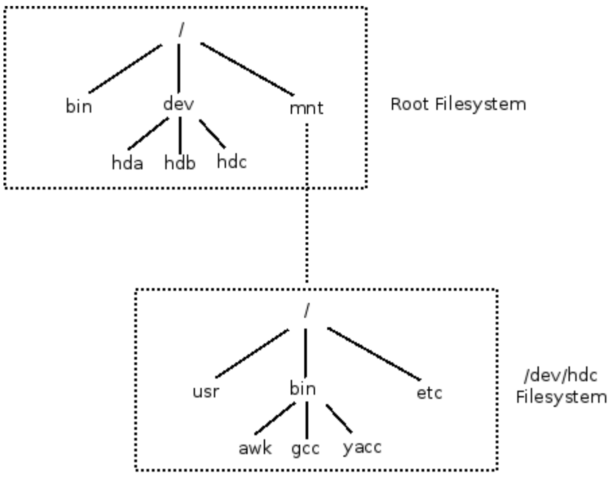


Figure 3: Filesystem tree with /dev/hdc device mounted on /mnt. Based on Bach (1986)

The *mount table* is responsible for storing information about mounted filesystems. Our implementation over the XV6 is based on the work of Bach (1986) and represents the *mount table* using two major structures: *mntentry*, representing each table entry; and the global structure *mtable*, representing the table itself. Both *mntentry* and *mtable* are respectively shown in Listings 3.7 and 3.8.

Listing 3.7: struct mntentry

```
// Mount Table Entry
struct mntentry {
  struct inode *m_inode;
  struct inode *m_rtinode;
  void *pdata;
  int dev;
  int flag;
};
```

Listing 3.8: struct mtable

```
// Mount Table Structure
struct {
  struct spinlock lock;
  struct mntentry mpoint[MOUNTSIZE];
} mtable;
```

Each entry in the mount table stores the information presented in Table 8.

The global mount table representation stores an array of mount entries with size *MOUNTSIZE* and a spin lock to be used internally by the kernel to control concurrent access. There are some utility functions defined to manipulate this table, as presented in Listing 3.9.

Table 8: Description of *mntentry* variables.

| mntentry field | Description |
|---|---|
| m_inode | A pointer to the inode that is the mount point named *m_inode*. ("*/mnt*" of the root filesystem in Figure 3). |
| m_rtinode | A pointer to the inode that is the root of the mounted filesystem. ("*/*" of the "*/dev/hdc*" filesystem in Figure 3). |
| pdata | A pointer to entry's private data (normally it is the superblock). |
| dev | The block device identifier. |
| flags | A flag member for internal kernel manipulation. |

Listing 3.9: List of helper functions for the mount table.

```
// Utility functions
struct inode* mtablertinode(struct inode * ip);
struct inode* mtablemntinode(struct inode * ip);
int isinoderoot(struct inode* ip);
void mountinit(void);
```

The function *mtablertinode()* returns the root *inode* of the mounted filesystem in which *ip* is a mount point. The function *mtablemntinode()* returns the *inode* of the mount point where the root *inode ip* is mounted. Both functions are used by the modified implementation of the *namex* function, shown in Listing 3.13.

The *mount* system call was implemented in *src/sysfile.c*. We present the main idea of this system call in Listing 3.10, omitting error checking to improve readability.

Listing 3.10: Simplified version of the mount system call.

```
int sys_mount(void) {
  char *devf; char *path; char *fstype;
  struct inode *ip, *evi;
  // Handle syscall arguments
  if (argstr(0, &devf) < 0 ||
      argstr(1, &path) < 0 ||
      argstr(2, &fstype) < 0) {
    return -1;
  }
  // Get inodes
  if ((ip = namei(path)) == 0 ||
      (devi = namei(devf)) == 0) {
    return -1;
  }

  struct filesystem_type *fs_t = getfs(fstype);
17    // Open the device and check if everything is ok.
18    bdev_open(devi);
19    // Add this to a list of filesystem type
20    putvfsonlist(devi->major, devi->minor, fs_t);
21    // Call specific fs mount operation.
22    fs_t->ops->mount(devi, ip);
23    //Turn the current ip into a mount point
24    ip->type = T_MOUNT;
25    return 0;
26  }
```

From Line 2 to Line 14, this function is setting up the local variables by parsing the system call arguments and getting the necessary information to perform the *mount* operation. Line 16 checks if the *fstype* is supported by the kernel. Line 18 opens the device to be mounted and checks if it is possible to access the hardware without error. There is a registration of this mount operation indicating the device identifiers (i.e. major and minor numbers) and the filesystem type in Line 20. Line 22 is using the VFS layer to call the filesystem-specific *mount* operation. This operation will read the device *superblock* and request a *mount entry* on the *mount table*. To complete the operation, the inode representing the mount point is marked as *T_MOUNT*.

Filesystem-specific *mount* operations change for different filesystem types. To help the comprehension of the *mount* system call, a simple version of the filesystem-specific *mount* operation for the XV6's default filesystem, that we named S5, is shown in Listing 3.11. You can find the complete implementation in *src/s5.c* (see Appendix B).

Listing 3.11: S5's mount operation handler.

```
int s5_mount(struct inode *devi, struct inode *ip) {
  struct mntentry *mp;
  s5_ops.readsb(devi->minor, &sb[devi->minor]);
  struct inode *devrtip = s5_ops.getroot(devi->major, devi->minor);
  for (mp = &mtable.mpoint[0]; mp < &mtable.mpoint[MOUNTSIZE]; mp++) {
    // This slot is available
    if (mp->flag == 0) {
found_slot:
      mp->dev = devi->minor;
      mp->m_inode = ip;
      mp->pdata = &sb[devi->minor];
      mp->flag |= M_USED;
      mp->m_rtinode = devrtip;
      initlog(devi->minor);
      return 0;
16       } else {
17         // The disk is already mounted
18         if (mp->dev == devi->minor) {
19           return -1;
20         }
21         if (ip->dev == mp->m_inode->dev &&
22             ip->inum == mp->m_inode->inum)
23           goto found_slot;
24       }
25     }
26     return -1;
27   }
```

Almost every operation performed in *s5_mount* can be shared with other filesystems. The *superblock* is read in Line 3 through the *s5_ops* structure, which is global in *src/s5.c*. Line 4 reads the root inode of the device being mounted. The loop between Lines 5 and 25 searches for a empty entry on mount table and, when it finds one, the *mntentry* is set. Line 14 initializes the log system for this filesystem. In addition, this implementation does not enable a device be mounted twice, as may be seen in Line 18. Finally, Line 21 checks if the mount point is already an entry and updates it to point to the new device to be mounted.

To exemplify the cases discussed in Section 3.3, Firgure 4 illustrates the relationship between *Inode Table* and *Mount Table*. The XV6 implement this diagram with the support of the *mtablemntinode()* and *mtablertinode()* functions, both presented in Listing 3.9.

### 3.3.2 Modifications over the XV6

The *mount* operation required changes over the XV6 code to be implemented. The first change was over the IDE driver code, because it was hard coded to support just two IDE disk and both slots were already in use as boot disk and root XV6's filesystem. We changed the driver to use the Slave Bus (TECHNOLOGY, 1993) and now it is possible to attach 4 IDE devices on XV6. Also, *namex* and *iget* functions were changed to support path translation with crossing mount points.

The updated *iget* function checks if the required inode is a mount point (i.e. its type is *T_MOUNT*). If it is true, it finds the *mount table entry* for this *inode*, then get the root *inode* of the mounted filesystem using the *mtablertinode()*, and return it as the requested *inode*. This algorithm ensures that a path translation crossing mount points follows the direction from the mount point to the mounted filesystem correctly. Listing 3.12 shows the updated version of *iget()* function.

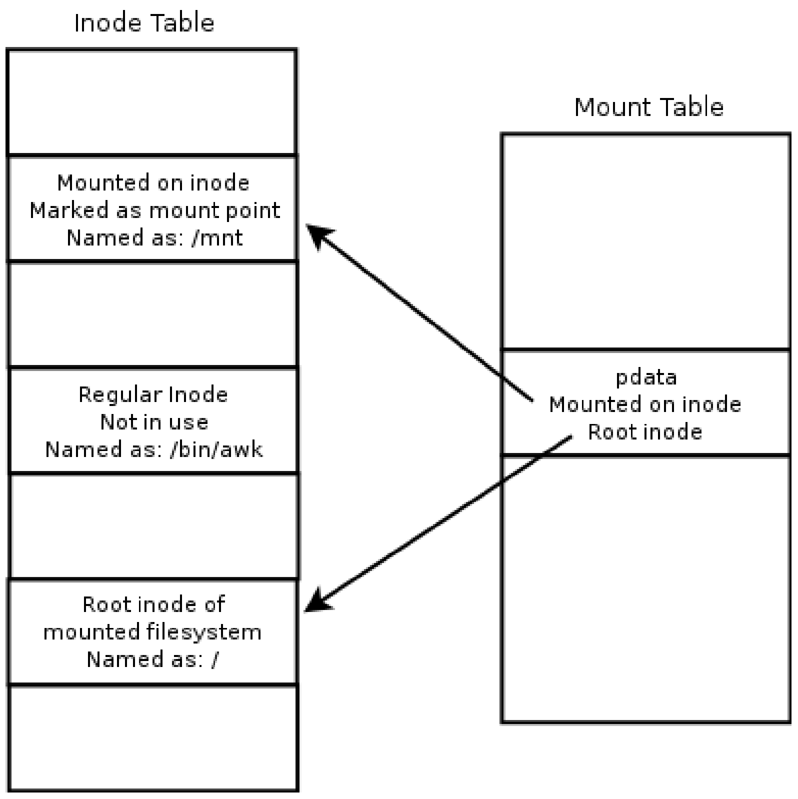


Figure 4: Diagram showing the relation between *Inode Table* and *Mount Table* based on Bach (1986).

Listing 3.12: iget() - updated function supporting path translation with crossing mount points.

```
struct inode* iget(uint dev, uint inum,
                   int (*fill_inode)(struct inode *)){
  struct inode *ip, *empty;
  struct filesystem_type *fs_t;
  acquire(&icache.lock);
  empty = 0;

  // Is the inode already cached?
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      // If the current inode is an mount point
      if (ip->type == T_MOUNT) {
        struct inode *rinode = mtablertinode(ip);
        if (rinode == 0) {
          panic("Invalid␣Inode␣on␣Mount␣Table");
        }
        rinode->ref++;
        release(&icache.lock);
        return rinode;
      }
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }
  // Recycle an inode cache entry.
  if(empty == 0)
    panic("iget:␣no␣inodes");
  fs_t = getvfsentry(IDEMAJOR, dev)->fs_t;
  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->flags = 0;
  ip->fs_t = fs_t;
  ip->iops = fs_t->iops;
  release(&icache.lock);
  if (!fill_inode(ip)) {
    panic("Error␣on␣fill␣inode");
  }
  return ip;
}
```

This implementation of the *iget()*, however, does not handle path translation in the opposite direction (i.e. coming from the mounted device to the mount point direction). This case can happen when you go back in the filesystem tree (i.e. the path has ".."). To handle this problem, the *namex* function had to be modified, as presented in Listing 3.13.

Listing 3.13: namex() - updated function supporting path translation with crossing mount points.

```
static struct inode*
namex(char *path, int nameiparent, char *name) {
  struct inode *ip, *next, *ir;
  if(*path == '/')
    ip = rootfs->fs_t->ops->getroot(IDEMAJOR, ROOTDEV);
  else
    ip = idup(proc->cwd);
  while((path = skipelem(path, name)) != 0){
```

```
9        ip->iops->ilock(ip);
10       if(ip->type != T_DIR){
11         iunlockput(ip);
12         return 0;
13       }
14       if(nameiparent && *path == '\0'){
15         // Stop one level early.
16         ip->iops->iunlock(ip);
17         return ip;
18       }
19     component_search:
20       if((next = ip->iops->dirlookup(ip, name, 0)) == 0){
21         iunlockput(ip);
22         return 0;
23       }
24       ir = next->fs_t->ops->getroot(IDEMAJOR, next->dev);
25       if (next->inum == ir->inum  &&
26           isinoderoot(ip) &&
27           (strncmp(name, "..", 2) == 0)) {
28         struct inode *mntinode = mtablemntinode(ip);
29         iunlockput(ip);
30         ip = mntinode;
31         ip->iops->ilock(ip);
32         ip->ref++;
33         goto component_search;
34       }
35       iunlockput(ip);
36       ip = next;
37     }
38     if(nameiparent){
39       iput(ip);
40       return 0;
41     }
42     return ip;
43   }
```

The modification to support crossing mount points in the *namex* function was made between Lines 20 and 35. It checks if the reached *inode* is a root *inode* and if the previous path component is "..". If it is true, it is a root *inode* from a mounted filesystem and we have to find the mount point *inode* into the mount table (Line 29) to look for the ".." directory entry from it. This is the main reason why the mount point is required to be a folder, once all folders contain at least "." and ".." directory entries. This change ensures that a path translation crossing a mount point from the mounted filesystem to the mount point will be performed correctly.

### 3.3.3 The block device-filesystem mapping

Some kernel operations must be aware about what is the filesystem type for the block device it is manipulating (see Listing 3.12 which requires this information to setup the *inode* correctly). To avoid wasting time looking for this information in the mount table, XV6 VFS contains an object called *vfsmlist* that is responsible for storing a list of *vfs*. This structure is defined on *src/vfs.h* and is shown in Listing **??**.

Listing 3.14: strcut vfs

```c
struct vfs {
  int major;
  int minor;
  int flag;
  struct filesystem_type *fs_t;
  struct list_head fs_next; // Next mounted on vfs
};
```

With this structure, it is possible to map a device through its major and minor identifiers into a filesystem type. It is important to say that this mapping should be implemented using hash, but as the performance is not the major concern of this work, we used a linked list instead.

Listing 3.15: List of helper functions for vfs.

```c
struct vfs* getvfsentry(int major, int minor);
int putvfsonlist(int major, int minor, struct filesystem_type *fs_t);
```

To abstract the use of this list, XV6 VFS offers two helpers function: *getvfsentry()*, used to retrive a *vfs* reference for a device and *putvfsonlist()*, used to link device to its filesystem type.

# 4 Implementing a new filesystem on XV6

In this chapter, we describe the necessary steps to implement a new filesystem using our XV6 VFS. To illustrate this process, we use a basic version of EXT2 (CARD; TS'O; TWEEDIE, 2010). All listings presented in this chapter can be found in *src/ext2.c* and *src/ext2.h* (see Apendix C). Before discussing implementation, we present an overview of EXT2's concepts. To a complete documentation of EXT2's design and implementation, please read the official documentation (POIRIER, 2011).

## 4.1 The EXT2 filesystem

EXT2 is a block-based filesystem implemented by Rémy Card, Theodore Ts'o and Stephen Tweedie to substitute the Extended Filesystem, maintaining old internal structures while providing new functionalities. It was first released on January 1993 as part of the Linux kernel and was further used as the standard filesystem in different major Linux distributions. Structures from EXT4 and EXT3 were strongly influenced by the EXT2 internal structure. Unlike recent filesystems, EXT2 does not support any optimization feature, such as journaling, journal checksums or extents. However, due its simplicity, it is a good start point for filesystem developers.

### 4.1.1 EXT2 disk organization

The EXT2 disk organization is strongly based on the layout of the BSD filesystem (MCKUSICK et al., 1984). Unlike previous filesystems, EXT2 is physically divided in block groups to improve sequential access, since it allows to allocate related data, such as directories and files, physically near each other. The physical structure of an EXT2 filesystem is illustrated in Figure 5.

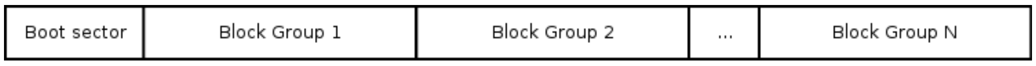


Figure 5: EXT2 filesystem architecture.

There is a boot sector in the first 1024 bytes. Then, follows N block groups, where each block group is divided in blocks with 1 up to 8KB. The number of block groups, inodes and blocks is varies depending on the partition size and the block size. These parameters can be configured when the filesystem is being installed in a device using the *mkfs.ext2* utility.

The first block group contains a copy of important filesystem control information, such as superblock and filesystem descriptors (e.g. block group descriptor table) as well as part of the filesystem itself, with a block bitmap, an inode bitmap, a piece of the inode table and data blocks, as shown in Figure 6.

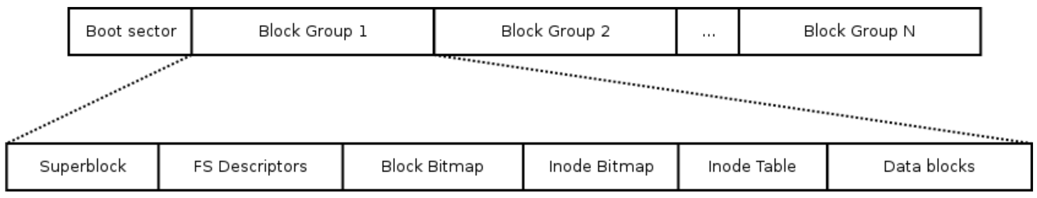


Figure 6: Layout of the first EXT2 Block Group.

Table 9 shows a layout of a 20MB EXT2 filesystem with block size of 1KB and block group size of 8MB. As may be observed, there are backups for *superblock* and *filesystem descriptors*. They increase the reliability, since they make it possible to recover the original data in case of corruption.

Table 9: Sample 20MB Ext2 filesystem using 1KiB block size. Based on Poirier (2011)

| **Block Offset** | **Length** | **Description** |
|---|---|---|
| byte 0 | 512 bytes | boot record (if present) |
| byte 512 | 512 bytes | additional boot record data (if present) |
| – block group 1, blocks 1 to 8192 – | | |
| byte 1024 | 1024 bytes | superblock |
| block 2 | 1 block | filesystem descriptor table |
| block 3 | 1 block | block bitmap |
| block 4 | 1 block | inode bitmap |
| block 5 | 214 blocks | inode table |
| block 219 | 7974 blocks | data blocks |
| – block group 2, blocks 8193 to 16384 – | | |
| block 8193 | 1 block | superblock backup |
| block 8194 | 1 block | filesystem descriptor table backup |
| block 8195 | 1 block | block bitmap |
| block 8196 | 1 block | inode bitmap |
| block 8197 | 214 blocks | inode table |
| block 8408 | 7974 blocks | data blocks |
| – block group 3, blocks 16385 to 24576 – | | |
| block 16385 | 1 block | block bitmap |
| block 16386 | 1 block | inode bitmap |
| block 16387 | 214 blocks | inode table |
| block 16601 | 3879 blocks | data blocks |

The layout of a disk is predictable when block size, number of blocks per group, inodes per group are known as well. These information are located in the superblock structure, and the EXT2 implementations use these values to compute the correct offset of an inode entry on the *inode table*, to find a specific data block, and so on.

### 4.1.2 Important EXT2 structures

Every filesystem requires internal and specific structure representations to enable its data manipulation with a certain level of abstraction. These structures must stricty follow the documented layout, because they are used to access the raw data from block devices, and if wrong operations are performed, they may end up corrupting the filesystem.

The main structures in EXT2 are *ext2_superblock* for EXT2 *superblock* manipulation, *ext2_inode* for EXT2 *inode* manipulation, *ext2_dir_entry_2* for EXT2 directory entry manipulation, and *ext2_block_group_desc* for EXT2 block group manipulation.

#### 4.1.2.1 *ext2_superblock*

The structure *ext2_superblock* is used to manage EXT2's physical superblock. The stored information is presented in Listing 4.1, and its complete documentation is available on the EXT2 documentation (POIRIER, 2011).

Listing 4.1: struct *ext2_superblock*

```
struct ext2_superblock {
  uint32 s_inodes_count;    /* Inodes count */
  uint32 s_blocks_count;    /* Blocks count */
  uint32 s_r_blocks_count;  /* Reserved blocks count */
  uint32 s_free_blocks_count;  /* Free blocks count */
  uint32 s_free_inodes_count;  /* Free inodes count */
  uint32 s_first_data_block;  /* First Data Block */
  uint32 s_log_block_size;  /* Block size */
  uint32 s_log_frag_size;  /* Fragment size */
  uint32 s_blocks_per_group;  /* # Blocks per group */
  uint32 s_frags_per_group;  /* # Fragments per group */
  uint32 s_inodes_per_group;  /* # Inodes per group */
  uint32 s_mtime;    /* Mount time */
  uint32 s_wtime;    /* Write time */
  uint16 s_mnt_count;    /* Mount count */
  uint16 s_max_mnt_count;  /* Maximal mount count */
  uint16 s_magic;    /* Magic signature */
  uint16 s_state;    /* File system state */
  uint16 s_errors;    /* Behaviour when detecting errors */
  uint16 s_minor_rev_level;   /* minor revision level */
  uint32 s_lastcheck;    /* time of last check */
  uint32 s_checkinterval;  /* max. time between checks */
  uint32 s_creator_os;    /* OS */
  uint32 s_rev_level;    /* Revision level */
  uint16 s_def_resuid;    /* Default uid for reserved blocks */
  uint16 s_def_resgid;    /* Default gid for reserved blocks */
  uint32 s_first_ino;     /* First non-reserved inode */
  uint16 s_inode_size;     /* size of inode structure */
  uint16 s_block_group_nr;   /* block group # of this superblock */
  uint32 s_feature_compat;   /* compatible feature set */
  uint32 s_feature_incompat;   /* incompatible feature set */
  uint32 s_feature_ro_compat;   /* readonly-compatible feature set */
  uint8  s_uuid[16];    /* 128-bit uuid for volume */
  char   s_volume_name[16];   /* volume name */
  char   s_last_mounted[64];   /* directory where last mounted */
  uint32 s_algorithm_usage_bitmap; /* For compression */
  uint8  s_prealloc_blocks;  /* Nr of blocks to try to preallocate*/
  uint8  s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
  uint16 s_padding1;
  uint8  s_journal_uuid[16];  /* uuid of journal superblock */
  uint32 s_journal_inum;    /* inode number of journal file */
  uint32 s_journal_dev;    /* device number of journal file */
  uint32 s_last_orphan;    /* start of list of inodes to delete */
  uint32 s_hash_seed[4];    /* HTREE hash seed */
  uint8  s_def_hash_version;  /* Default hash version to use */
  uint8  s_reserved_char_pad;
  uint16 s_reserved_word_pad;
  uint32 s_default_mount_opts;
  uint32 s_first_meta_bg;   /* First metablock block group */
  uint32 s_reserved[190];  /* Padding to the end of the block */
};
```

#### 4.1.2.2 *ext2_inode*

The structure *ext2_inode* keeps track of every directory, regular file, symbolic link or special file stored in the filesystem. It stores their location, size, type and access rights. Filenames are not stored in the inode itself, since this information is contained in directory entries. Listing 4.2 presents this structure.

Listing 4.2: struct ext2_inode

```
struct ext2_inode {
  uint16 i_mode;  /* File mode */
  uint16 i_uid;   /* Low 16 bits of Owner Uid */
  uint32 i_size;  /* Size in bytes */
  uint32 i_atime; /* Access time */
  uint32 i_ctime; /* Creation time */
  uint32 i_mtime; /* Modification time */
```

```
uint32 i_dtime; /* Deletion Time */
uint16 i_gid;   /* Low 16 bits of Group Id */
uint16 i_links_count; /* Links count */
uint32 i_blocks; /* Blocks count */
uint32 i_flags;  /* File flags */
union {
  struct {
    uint32  l_i_reserved1;
  } linux1;
  struct {
    uint32  h_i_translator;
  } hurd1;
  struct {
    uint32  m_i_reserved1;
  } masix1;
} osd1;     /* OS dependent 1 */
uint32 i_block[EXT2_N_BLOCKS];  /* Pointers to blocks */
uint32 i_generation;  /* File version (for NFS) */
uint32 i_file_acl;    /* File ACL */
uint32 i_dir_acl;     /* Directory ACL */
uint32 i_faddr;       /* Fragment address */
union {
  struct {
    uint8  l_i_frag;  /* Fragment number */
    uint8  l_i_fsize; /* Fragment size */
    uint16 i_pad1;
    uint16 l_i_uid_high;  /* these 2 fields    */
    uint16 l_i_gid_high;  /* were reserved2[0] */
    uint32 l_i_reserved2;
  } linux2;
  struct {
    uint8  h_i_frag;  /* Fragment number */
    uint8  h_i_fsize; /* Fragment size */
    uint16 h_i_mode_high;
    uint16 h_i_uid_high;
    uint16 h_i_gid_high;
    uint32 h_i_author;
  } hurd2;
  struct {
    uint8  m_i_frag;  /* Fragment number */
    uint8  m_i_fsize; /* Fragment size */
    uint16 m_pad1;
```

```
      uint32 m_i_reserved2[2];
    } masix2;
  } osd2;   /* OS dependent 2 */
};
```

It is important to say that, even if the EXT2 implementation does not use some members of this structure, it is necessary to keep them to avoid corrupting the filesystem meta-data.

#### 4.1.2.3 *ext2_dir_entry_2*

EXT2's directory entries are stored by a linked list, and each entry contains the *inode* number, total entry length, name length, file type and filename. Listing 4.3 presents this structure.

Listing 4.3: struct ext2_dir_entry_2

```
struct ext2_dir_entry_2 {
  uint32 inode;      /* Inode number */
  uint16 rec_len;    /* Directory entry length */
  uint8  name_len;   /* Name length */
  uint8  file_type;
  char   name[];     /* File name, up to EXT2_NAME_LEN */
};
```

#### 4.1.2.4 struct *ext2_block_group_desc*

This structure stores the description of block groups. It provides the location of *inode bitmap*, *inode table*, block bitmap, free blocks count and other useful information to manage each block group. Its instances are stored by the *Block Group Descriptor Table*, an array stored immediately after the superblock (see Figure 6 and Table 9). Listing 4.4 presents this structure.

Listing 4.4: struct ext2_block_group_descriptor

```
struct ext2_group_desc {
  uint32 bg_block_bitmap;       /* Blocks bitmap block */
  uint32 bg_inode_bitmap;       /* Inodes bitmap block */
  uint32 bg_inode_table;        /* Inodes table block */
  uint16 bg_free_blocks_count;  /* Free blocks count */
  uint16 bg_free_inodes_count;  /* Free inodes count */
  uint16 bg_used_dirs_count;    /* Directories count */
  uint16 bg_pad;
  uint32 bg_reserved[3];
};
```

## 4.2 Implementation strategy for new filesystems

The task of adding a new filesystem to the kernel is trivial in XV6 VFS. The triviality, however, is not extended to the internal filesystem implementation. It is not a wise idea to implement all filesystem features and only after that start the validation phase. Thus, a good strategy is to create all filesystem-dependent operations as empty operations that call the *panic* function. After this step, configure and register the new filesystem and start coding one operation at time for each system call.

This strategy will let you test and debug your code in parts and locate errors with more precision, thanks to granularity of the code.

Since the EXT2 implementation is not the objective of this work, readers are referred to the Linux kernel implementation (POIRIER, 2011) or to check the file *src/ext2.c* for further details.

## 4.3 *vfs_operations* for EXT2

As discussed in Section 3.2.1, it is necessary to create an object pointing to the filesystem-dependent general operations. Listing 4.5 shows the *vfs_operations* structure for EXT2 as an example.

Listing 4.5: vfs_operations instance for EXT2.

```c
struct vfs_operations ext2_ops = {
  .fs_init = &ext2fs_init,
  .mount   = &ext2_mount,
  .unmount = &ext2_unmount,
  .getroot = &ext2_getroot,
  .readsb  = &ext2_readsb,
  .ialloc  = &ext2_ialloc,
  .balloc  = &ext2_balloc,
  .bzero   = &ext2_bzero,
  .bfree   = &ext2_bfree,
  .brelse  = &brelse,
  .bwrite  = &bwrite,
  .bread   = &bread,
  .namecmp = &ext2_namecmp
};
```

Each EXT2-specific function starts with *"ext2_"*. It is an important convention to follow when programming in C because it works similar to a namespace and avoids compilation errors due to multiple definition of identifiers. Operations *brelse*, *bwrite* and *bread* are pointing to internal kernel generic operations because they do not need a filesystem-specific implementation.

## 4.4 *inode_operations* for EXT2

As stated in Section 3.2.2, it is also necessary to create an instance of *inode_operations* pointing to filesystem-dependent inode operations. Listing 4.6 shows the *inode_operations* structure for EXT2.

Listing 4.6: inode_operations instance for EXT2.

```c
struct inode_operations ext2_iops = {
  .dirlookup  = &ext2_dirlookup,
  .iupdate    = &ext2_iupdate,
  .itrunc     = &ext2_itrunc,
  .cleanup    = &ext2_cleanup,
  .bmap       = &ext2_bmap,
  .ilock      = &ext2_ilock,
  .iunlock    = &generic_iunlock,
  .stati      = &generic_stati,
  .readi      = &generic_readi,
  .writei     = &ext2_writei,
  .dirlink    = &ext2_dirlink,
  .unlink     = &ext2_unlink,
  .isdirempty = &ext2_isdirempty
};
```

As may be seen, the structure *inode_operations* also defines some generic operations: *iunlock*, *stati* and *readi*. The same naming convention was used for filesystem-specific functions.

## 4.5 Configuring and registering the structure *filesystem_type*

One of the most important steps to support a new filesystem in our XV6 VFS is to create the structure *filesystem_type* and populate its variables. It is implemented it on *src/ext2.c* because there is no reason to keep a filesystem-specific code global. It basically points to the structures in Sections 4.3 and 4.4 and stores the filesystem name, as be seen in Listing 4.7.

Listing 4.7: filesystem_type instance for EXT2.

```c
struct filesystem_type ext2fs = {
  .name = "ext2",
  .ops = &ext2_ops,
  .iops = &ext2_iops
};
```

Then it is necessary to inform the kernel that there is a new filesystem to be supported. To implement this, we use the function *register_fs()*, which was introduced in Section 3.1.3. Our EXT2 implementation contains a function named *initext2fs(void)* to initialize all internal data, including the registration of the filesystem, as shown in Listing 4.8.

Listing 4.8: initext2fs() -EXT2 initialization function.

```
int initext2fs(void) {
  initlock(&ext2_sb_pool.lock, "ext2_sb_pool");
  return register_fs(&ext2fs);
}
```

Unlike the Linux kernel, we cannot load modules dynamically on XV6, so the function *initext2fs()* had to be hardcoded into the kernel initialization code. To organize filesystem initializations, the function *initfss()* was created as in Listing 4.9.

Listing 4.9: Kernel function to initialize filesystems.

```
static void initfss(void) {
  if (inits5fs() != 0) // init s5 fs
    panic("S5␣not␣registered");
  if (initext2fs() != 0) // init ext2 fs
    panic("ext2␣not␣registered");
}
```

Following these steps, the mount system call presented in Section 3.3 wil be able to support devices formatted with the EXT2 filesystem.

## 4.6 Final remarks

Our EXT2 implementation was based on Linux's version. However, many changes were necessary because of the internal kernel data manipulation in XV6. The first change was to remove byte endianness compatibility present in Linux. As XV6 is designed to run into a *X86* architecture, we removed all instructions to convert the byte endianness. Also, another change was the usage of *buffer_head* structure to manage blocks of the filesystem in XV6, which diverge from Linux's implementation that uses *page cache*. It is important to know that the structures and functions presented in Sections 4.3 and 4.4 can be implemented based on available implementations of the desired filesystem.

# 5 XV6 VFS evaluation

## 5.1 Methodology

In this chapter, we evaluate the VFS operability on XV6. With our EXT2 implementation, it is possible to check the interoperability with other operating systems that support this filesystem. The environment used to perform this experiment was a virtual machine running *Debian 3.2.65-1* with *Linux kernel 3.2.0-4-amd64*, which was used to compile XV6's code and to create EXT2 filesystems with *mkfs.ext2*. The XV6 runs in an i386 machine emulated with *qemu-system-i386* (check Appendix A to configure an execution environment).

The major goal of this experiment is to show that our XV6 VFS works properly and that EXT2 block devices are totally operational. To this end, operations like create and read directories, read directory entries, write or delete files are going to be performed on XV6, and, after that, the filesystem is going to be mounted on Linux to check if they were performed correctly. Then, the reverse direction is going to be considered. To do that, the filesystem is going to be modified on Linux and then mounted on XV6 to check if it is possible to successfully access the modifications. Considering that Linux's EXT2 implementation is a widely used and stable commercial filesystem, used as base for filesystems like EXT3 and EXT4, its implementation using our XV6 VFS implementation validates our architectural design.

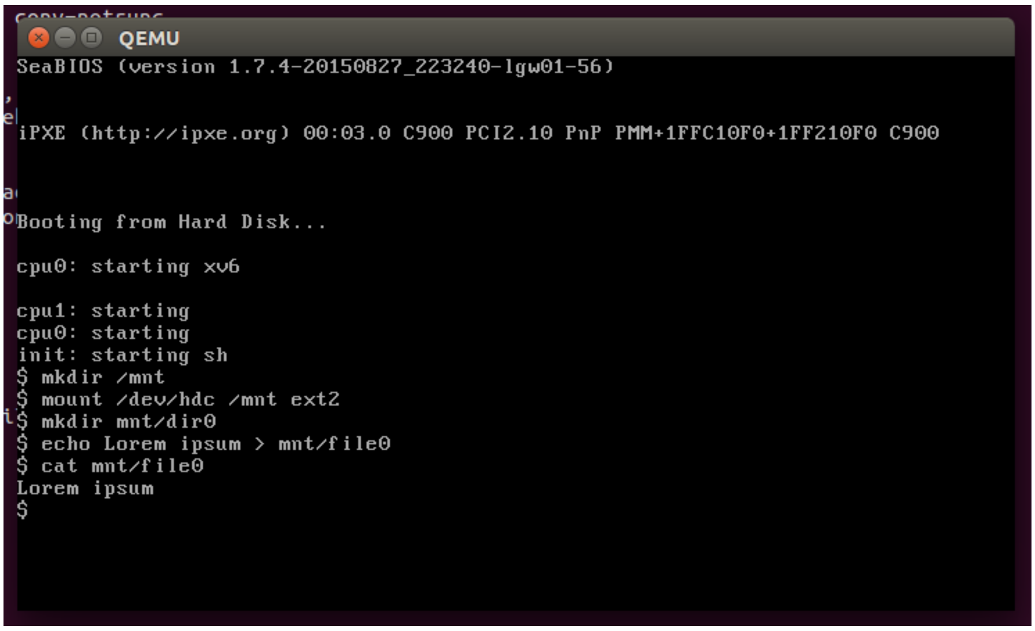
## 5.2 Experiments and results

XV6's terminal does not support script automation, so our experiments hat to be performed manually. In the first part of this evaluation, we ran commands shown in Listing 5.1, and the obtained results are presented in Figure 7.

Listing 5.1: List of commands that modify the EXT2 filesystem on XV6.

```
mkdir /mnt
mount /dev/hdc /mnt ext2
mkdir mnt/dir0
echo Lorem ipsum > mnt/file0
cat mnt/file0
```

These commands may look simple, but a lot of work was done by XV6 to process them. The first two commands are necessary to mount the EXT2 filesystem in a directory. The *mount* program, which execute the *mount* system call, requires three parameters: the



```
SeaBIOS (version 1.7.4-20150827_223240-lgw01-56)


iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+1FFC10F0+1FF210F0 C900


Booting from Hard Disk...

cpu0: starting xv6

cpu1: starting
cpu0: starting
init: starting sh
$ mkdir /mnt
$ mount /dev/hdc /mnt ext2
$ mkdir mnt/dir0
$ echo Lorem ipsum > mnt/file0
$ cat mnt/file0
Lorem ipsum
$
```

Figure 7: Execution of commands that modify the EXT2 filesystem on XV6.

device to be mounted, the directory where the device will be mounted, and the type of the filesystem contained in this device. After that, a directory named *dir0* is created on this device. Finally, to test if file writing is being done correctly, Line 4 writes *"Lorem ipsum"* into *file0* and Line 5 reads its contents to check if the write operation was successful.

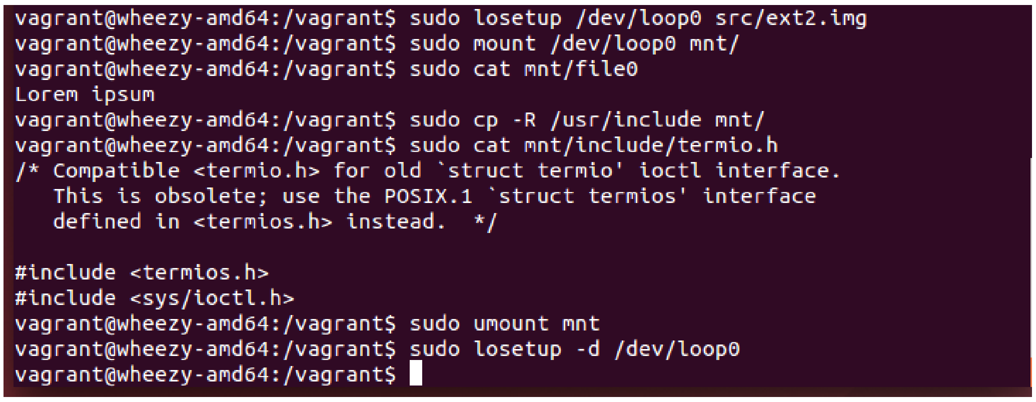
To make sure these modifications are correct in the device, we mounted this filesystem on Linux and ran another sequence of commands, as shown in Listing 5.2 and Figure 8.

Listing 5.2: List of commands that modify the EXT2 filesystem on Linux.

```
1 sudo losetup /dev/loop0 src/ext2.img
2 sudo mount /dev/loop0 mnt/
3 sudo cat mnt/file0
4 sudo cp -R /usr/include mnt/
5 sudo cat mnt/include/termio.h
6 sudo umount mnt
7 sudo losetup -d /dev/loop0
```

Line 1 uses tool of Linux kernel to copy the filesystem image located at *src/ext2.img* to a *loop device* to allow mounting this image as a virtual block device. Line 2 mounts the filesystem image into *mnt/*. Line 3 prints the content of *file0* in the terminal. Line 4 copies a complex directory structure located in */usr/include/* to *mnt/*. Lines 5 and 6 finish the manipulation of the EXT2 image.

Finally, we check if the manipulations made on Linux can be correctly loaded on XV6. This is done with the commands in Listing 5.3, and the results are shown in Figure 9.



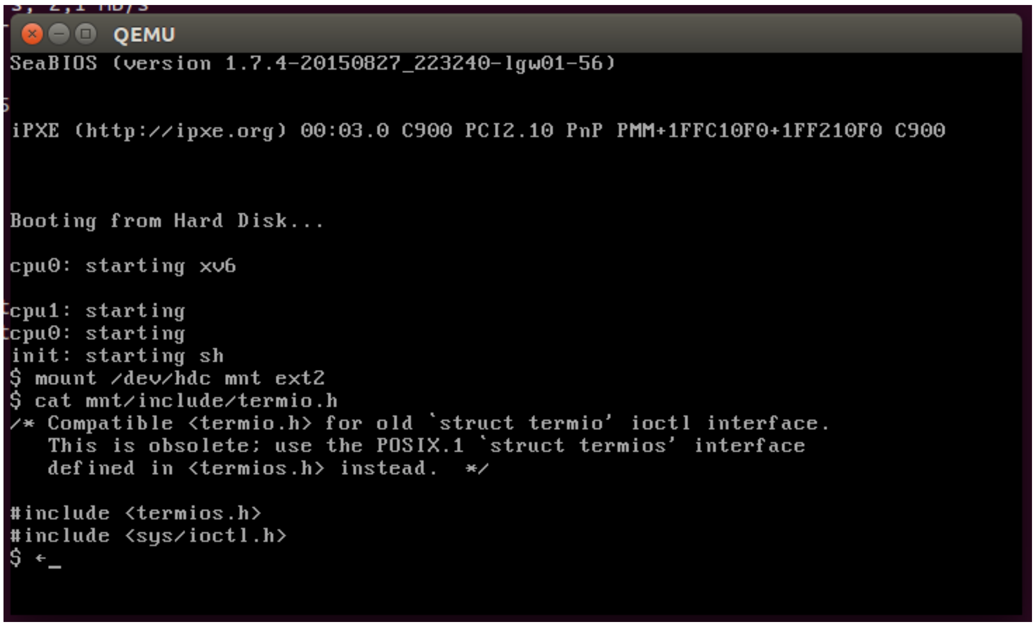
```
vagrant@wheezy-amd64:/vagrant$ sudo losetup /dev/loop0 src/ext2.img
vagrant@wheezy-amd64:/vagrant$ sudo mount /dev/loop0 mnt/
vagrant@wheezy-amd64:/vagrant$ sudo cat mnt/file0
Lorem ipsum
vagrant@wheezy-amd64:/vagrant$ sudo cp -R /usr/include mnt/
vagrant@wheezy-amd64:/vagrant$ sudo cat mnt/include/termio.h
/* Compatible <termio.h> for old `struct termio' ioctl interface.
   This is obsolete; use the POSIX.1 `struct termios' interface
   defined in <termios.h> instead.  */

#include <termios.h>
#include <sys/ioctl.h>
vagrant@wheezy-amd64:/vagrant$ sudo umount mnt
vagrant@wheezy-amd64:/vagrant$ sudo losetup -d /dev/loop0
vagrant@wheezy-amd64:/vagrant$
```

Figure 8: Execution of commands that modify the EXT2 filesystem on XV6.

Listing 5.3: List of commands to verify modifications in the EXT2 filesystem on XV6.

```
1 mount /dev/hdc /mnt ext2
2 cat mnt/include/termio.h
```



```
SeaBIOS (version 1.7.4-20150827_223240-lgw01-56)


iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+1FFC10F0+1FF210F0 C900



Booting from Hard Disk...

cpu0: starting xv6

cpu1: starting
cpu0: starting
init: starting sh
$ mount /dev/hdc mnt ext2
$ cat mnt/include/termio.h
/* Compatible <termio.h> for old `struct termio' ioctl interface.
   This is obsolete; use the POSIX.1 `struct termios' interface
   defined in <termios.h> instead.  */

#include <termios.h>
#include <sys/ioctl.h>
$ ←_
```

Figure 9: Execution of commands to verify modifications in the EXT2 filesystem on XV6.

After Line 2, it is possible verify that the content of the file *"mnt/include/termio.h"* was correctly printed in the terminal because we see the same output on Linux (see Figure 8). It is important to say that the command *cat* uses the system call *read*, which is implemented using two major VFS operations: *readi()* and *dirlookup()* (see section 3.2.2).

With these experiments, we were able to validate the major features of the EXT2 filesystem running on XV6. Unmodifed native XV6 commands were used to manipulate the filesystem, commands that were implemented using XV6 system calls. This behavior could be achieved only because our XV6 VFS implementation does not change the system call interface, just their internal functions, achieving the desired behavior of a VFS implementation in terms of abstraction. In addition, this experiment shows that our VFS allows adding new filesystems to XV6 without compromising its operation.

# 6 Conclusion

This work revealed to us one of the major advantages of a VFS, the interoperability of filesystems among operating systems. The compatibility with more than one filesystem is a very important feature in a modern operating system, what gave us the intuition that VFS design and implementation is a topic that should be taught to operating system engineers.

The major contribution of this work is the implementation and documentation of a simple, but powerful VFS layer in an operating system that is designed for academic purposes, making it a great start point for operating system developers. Our EXT2 implementation indicates that the VFS design achieved the desired abstraction. Another important contribution of our work is porting XV6's filesystem to run in the VFS layer, which is also part of the validation, since all operations performed in the root filesystem are using the VFS layer. In addition, it shows that our XV6 VFS allows using more than one type of filesystem at the same time.

## 6.1 Limitations and future works

As a future work, it would be interesting to implement a non Unix-like filesystem to validate the power of the VFS architecture presented in this work. The same thing is applied to diskless filesystems like *procfs*, *sysfs* or *NFS*.

In addition, there are some limitations in this XV6 VFS implementation that can improve its abstraction level. The first improvement that can be done is to make the VFS compatible with extent-based filesystem. In our implementation, we assume that all filesystems are block-based, which is not true for modern filesystems like BTRFS, EXT4 or ZFS.

Second, it is necessary to implement a new system call to read directory entries. Without this system call, it is not possible implement programs like *ls* in an elegant way.

Third, XV6 should be modified to enable global system time access in order to update the last time an *inode* is modified, and to improve the memory usage of Block I/O subsystem, since it is currently consuming 4KB per block on buffer cache even if the block is smaller than that.

Finally, it is important to create operations that support Access Control List patterns on the VFS layer. It was not implemented in this version because there is only one user on XV6, the root user.

# Bibliography

BACH, M. J. *The Design of the UNIX Operating System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986. ISBN 0-13-201799-7. Referenced 2 times in pages 27 and 31.

CARD, R.; TS'O, T.; TWEEDIE, S. Design and implementation of the second extended filesystem. *Proceedings of the First Dutch International Symposium on Linux*, 2010. Available from Internet: <http://web.mit.edu/tytso/www/linux/ext2intro.html>. Referenced in page 35.

KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in sun unix. In: . [S.l.: s.n.], 1986. p. 238–247. Referenced 2 times in pages 10 and 17.

LOVE, R. *Linux Kernel Development*. 3rd. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0672329468, 9780672329463. Referenced in page 14.

MCKUSICK, M. K. et al. A fast file system for unix. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 2, n. 3, p. 181–197, ago. 1984. ISSN 0734-2071. Available from Internet: <http://doi.acm.org/10.1145/989.990>. Referenced in page 35.

PATE, S.; BOSCH, F. V. D. *UNIX Filesystems: Evolution, Design and Impemenation*. New York, NY, USA: John Wiley & Sons, Inc., 2003. ISBN 0471164836. Referenced 4 times in pages 9, 11, 14, and 15.

POIRIER, D. The second extended file system: Internal layout. 2011. Available from Internet: <http://www.nongnu.org/ext2-doc/ext2.pdf>. Referenced 4 times in pages 35, 36, 37, and 41.

RODEH, O.; BACIK, J.; MASON, C. Btrfs: The linux b-tree filesystem. *Trans. Storage*, ACM, New York, NY, USA, v. 9, n. 3, p. 9:1–9:32, ago. 2013. ISSN 1553-3077. Available from Internet: <http://doi.acm.org/10.1145/2501620.2501623>. Referenced in page 10.

TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. Referenced 2 times in pages 7 and 9.

TECHNOLOGY, I. S. Ata interface reference manual. 1993. Available from Internet: <ftp://ftp.seagate.com/acrobat/reference/111-1c.pdf>. Referenced in page 30.

# Appendix

# APPENDIX A – Configuring an environment to build XV6

There are two ways to build and run the xv6 OS. The first one is using a pre-configured vagrant machine and we recommend follow this way to avoid headaches. The second solution we recommend follow the official documentation that can be found in https://pdos.csail.mit.edu/6.828/2014/tools.html

Required software:

- Vagrant: https://www.vagrantup.com/downloads.html
- git: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
- QEMU : https://en.wikibooks.org/wiki/QEMU/Installing_QEMU

To configure an environment using Vagrant, the first thing to do is download and install the Vagrant software. You can follow the isntructions available in Vagrant's link given above. If you are not use to use vagrant, don't worry. All configurations are already done on this Vagrantfile (https://gist.github.com/caiolima/fdc6974c1fec0e57caac). The workflow using the Vagrant is simple: share the folder where the XV6's source code is beteween Vagrant machine and host operating system. This way, you can decide your own development environment (IDE, Editor, etc.).

After install and configure the vagrant environment, you need to download our XV6 code and place it in the shared folder between Vagrant machine and your host operating system. It is recommended clone the source code on the same folder where the Vagrantfile is placed. You can find this repository on https://github.com/caiolima/xv6-public or download the code using the following command:

```
git clone https://github.com/caiolima/xv6-public.git
```

After the clone command, your folder tree should have *Vagrantfile* and *xv6-public* folder. Our example we placed these files in *˜/xv6-dev* folder.

Now, we are almost in the end of the build step. We need to run the commands:

```
cd ~/xv6-dev
vagrant up
vagrant ssh
```

The first time you run *vagrant up* you will need to wait for a while because the Vagrant need to download the virtual machine image. After the download, the load time will be faster.

When you successfully ssh into the vagrant virtual machine, run the following commands to build the xv6 OS:

```
cd /vagrant/xv6-public/src
make
```

To create an valid EXT2 filesystem, you need to run the following commands on /vagrant/xv6-public/src folder:

```
dd if=/dev/zero of=ext2.img bs=30M count=1
sudo mkfs.ext2 -b 1024 -T "EXT2_TEST" ext2.img
```

To run and test the XV6, you need to open the terminal application of your host operating system and run:

```
make qemu
```

Finally, the QEMU will start and boot the XV6.

# APPENDIX B – src/s5.c

```c
// It is the s5 filesystem implementation

#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "vfs.h"
#include "buf.h"
#include "file.h"
#include "vfsmount.h"
#include "s5.h"

/*
 * Its is a pool to allocate s5 inodes structs.
 * We use it becase we don't have a kmalloc function.
 * With an kmalloc implementatios, it need to be removed.
 */
static struct {
  struct spinlock lock;
  struct s5_inode s5_i_entry[NINODE];
} s5_inode_pool;

struct s5_inode*
alloc_s5_inode()
{
  struct s5_inode *ip;

  acquire(&s5_inode_pool.lock);
  for (ip = &s5_inode_pool.s5_i_entry[0]; ip < &s5_inode_pool.s5_i_entry[NINODE]; ip++) {
    if (ip->flag == S5_INODE_FREE) {
      ip->flag |= S5_INODE_USED;
      release(&s5_inode_pool.lock);

      return ip;
    }
  }
  release(&s5_inode_pool.lock);

  return 0;
}

static struct {
  struct spinlock lock;
  struct s5_superblock sb[MAXVFSSIZE];
} s5_sb_pool; // It is a Pool of S5 Superblock Filesystems

struct s5_superblock*
alloc_s5_sb()
{
  struct s5_superblock *sb;

  acquire(&s5_sb_pool.lock);
  for (sb = &s5_sb_pool.sb[0]; sb < &s5_sb_pool.sb[MAXVFSSIZE]; sb++) {
    if (sb->flags == S5_SB_FREE) {
      sb->flags |= S5_SB_USED;
      release(&s5_sb_pool.lock);

      return sb;
    }
  }
  release(&s5_sb_pool.lock);

  return 0;
}

struct vfs_operations s5_ops = {
  .fs_init  = &s5fs_init,
  .mount    = &s5_mount,
  .unmount  = &s5_unmount,
  .getroot  = &s5_getroot,
  .readsb   = &s5_readsb,
  .ialloc   = &s5_ialloc,
  .balloc   = &s5_balloc,
  .bzero    = &s5_bzero,
  .bfree    = &s5_bfree,
  .brelse   = &brelse,
  .bwrite   = &bwrite,
  .bread    = &bread,
  .namecmp  = &s5_namecmp
};

struct inode_operations s5_iops = {
  .dirlookup  = &s5_dirlookup,
  .iupdate    = &s5_iupdate,
  .itrunc     = &s5_itrunc,
  .cleanup    = &s5_cleanup,
  .bmap       = &s5_bmap,
  .ilock      = &s5_ilock,
  .iunlock    = &generic_iunlock,
  .stati      = &generic_stati,
  .readi      = &s5_readi,
  .writei     = &s5_writei,
  .dirlink    = &generic_dirlink,
  .unlink     = &s5_unlink,
  .isdirempty = &s5_isdirempty
};

struct filesystem_type s5fs = {
  .name = "s5",
  .ops = &s5_ops,
  .iops = &s5_iops
};

int
inits5fs(void)
{
  initlock(&s5_sb_pool.lock, "s5_sb_pool");
  initlock(&s5_inode_pool.lock, "s5_inode_pool");
  return register_fs(&s5fs);
}

```

```c
int
s5fs_init(void)
{
  return 0;
}

int
s5_mount(struct inode *devi, struct inode *ip)
{
  struct mntentry *mp;

  // Read the Superblock
  s5_ops.readsb(devi->minor, &sb[devi->minor]);

  // Read the root device
  struct inode *devrtip = s5_ops.getroot(devi->major, devi->minor);

  acquire(&mtable.lock);
  for (mp = &mtable.mpoint[0]; mp < &mtable.mpoint[MOUNTSIZE]; mp++) {
    // This slot is available
    if (mp->flag == 0) {
found_slot:
      mp->dev = devi->minor;
      mp->m_inode = ip;
      mp->pdata = &sb[devi->minor];
      mp->flag |= M_USED;
      mp->m_rtinode = devrtip;

      release(&mtable.lock);

      initlog(devi->minor);
      return 0;
    } else {
      // The disk is already mounted
      if (mp->dev == devi->minor) {
        release(&mtable.lock);
        return -1;
      }

      if (ip->dev == mp->m_inode->dev && ip->inum == mp->m_inode->inum)
        goto found_slot;
    }
  }
  release(&mtable.lock);

  return -1;
}

int
s5_unmount(struct inode *devi)
{
  return 0;
}

struct inode *
s5_getroot(int major, int minor)
{
  return s5_iget(minor, ROOTINO);
}

```

```
void
s5_readsb(int dev, struct superblock *sb)
{
  struct buf *bp;
  struct s5_superblock *s5sb;

  if((sb->flags & SB_NOT_LOADED) == 0) {
    s5sb = alloc_s5_sb(); // Allocate a new S5 sb struct to the superblock.
  } else{
    s5sb = sb->fs_info;
  }

  // These sets are needed because of bread
  sb->major = IDEMAJOR;
  sb->minor = dev;
  sb->blocksize = BSIZE;

  bp = s5_ops.bread(dev, 1);
  memmove(s5sb, bp->data, sizeof(*s5sb) - sizeof(s5sb->flags));
  s5_ops.brelse(bp);

  sb->fs_info = s5sb;
}

struct inode*
s5_ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;
  struct s5_superblock *s5sb;

  s5sb = sb[dev].fs_info;

  for(inum = 1; inum < s5sb->ninodes; inum++){
    bp = s5_ops.bread(dev, IBLOCK(inum, (*s5sb)));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp);   // mark it allocated on the disk
      s5_ops.brelse(bp);
      return s5_iget(dev, inum);
    }
    s5_ops.brelse(bp);
  }
  panic("ialloc: no inodes");
}

uint
s5_balloc(uint dev)
{
  int b, bi, m;
  struct buf *bp;
  struct s5_superblock *s5sb;

  s5sb = sb[dev].fs_info;
  bp = 0;
  for (b = 0; b < s5sb->size; b += BPB) {
    bp = s5_ops.bread(dev, BBLOCK(b, (*s5sb)));
    for (bi = 0; bi < BPB && b + bi < s5sb->size; bi++) {
      m = 1 << (bi % 8);
      if ((bp->data[bi/8] & m) == 0) {  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        s5_ops.brelse(bp);
        s5_ops.bzero(dev, b + bi);
        return b + bi;
      }
    }
    s5_ops.brelse(bp);
  }
  panic("balloc:␣out␣of␣blocks");
}

void
s5_bzero(int dev, int bno)
{
  struct buf *bp;

  bp = s5_ops.bread(dev, bno);
  memset(bp->data, 0, BSIZE);
  log_write(bp);
  s5_ops.brelse(bp);
}

void
s5_bfree(int dev, uint b)
{
  struct buf *bp;
  int bi, m;
  struct s5_superblock *s5sb;

  s5sb = sb[dev].fs_info;
  s5_ops.readsb(dev, &sb[dev]);
  bp = s5_ops.bread(dev, BBLOCK(b, (*s5sb)));
  bi = b % BPB;
  m = 1 << (bi % 8);
  if((bp->data[bi/8] & m) == 0)
    panic("freeing␣free␣block");
  bp->data[bi/8] &= ~m;
  log_write(bp);
  s5_ops.brelse(bp);
}

struct inode*
s5_dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type == T_FILE || dp->type == T_DEV)
    panic("dirlookup␣not␣DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(s5_iops.readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink␣read");
    if(de.inum == 0)
      continue;
    if(s5_ops.namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return s5_iget(dp->dev, inum);
    }
  }

  return 0;
}

void
s5_iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;
  struct s5_superblock *s5sb;
  struct s5_inode *s5ip;

  s5ip = ip->i_private;
  s5sb = sb[ip->dev].fs_info;
  bp = s5_ops.bread(ip->dev, IBLOCK(ip->inum, (*s5sb)));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  memmove(dip->addrs, s5ip->addrs, sizeof(s5ip->addrs));
  log_write(bp);
  s5_ops.brelse(bp);
}

void
s5_itrunc(struct inode *ip)
{
  int i, j;
  struct buf *bp;
  uint *a;
  struct s5_inode *s5ip;

  s5ip = ip->i_private;

  for(i = 0; i < NDIRECT; i++){
    if(s5ip->addrs[i]){
      s5_ops.bfree(ip->dev, s5ip->addrs[i]);
      s5ip->addrs[i] = 0;
    }
  }

  if(s5ip->addrs[NDIRECT]){
    bp = s5_ops.bread(ip->dev, s5ip->addrs[NDIRECT]);
    a = (uint*)bp->data;
    for (j = 0; j < NINDIRECT; j++) {
      if (a[j])
        s5_ops.bfree(ip->dev, a[j]);
    }
    s5_ops.brelse(bp);
    s5_ops.bfree(ip->dev, s5ip->addrs[NDIRECT]);
    s5ip->addrs[NDIRECT] = 0;
```

```c
  }

  ip->size = 0;
  s5_iops.iupdate(ip);
}

void
s5_cleanup(struct inode *ip)
{
  memset(ip->i_private, 0, sizeof(struct s5_inode));
}

uint
s5_bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;
  struct s5_inode *s5ip;

  s5ip = ip->i_private;

  if(bn < NDIRECT){
    if((addr = s5ip->addrs[bn]) == 0)
      s5ip->addrs[bn] = addr = s5_ops.balloc(ip->dev);
    return addr;
  }
  bn -= NDIRECT;

  if(bn < NINDIRECT){
    // Load indirect block, allocating if necessary.
    if((addr = s5ip->addrs[NDIRECT]) == 0)
      s5ip->addrs[NDIRECT] = addr = s5_ops.balloc(ip->dev);
    bp = s5_ops.bread(ip->dev, addr);
    a = (uint*)bp->data;
    if((addr = a[bn]) == 0){
      a[bn] = addr = s5_ops.balloc(ip->dev);
      log_write(bp);
    }
    s5_ops.brelse(bp);
    return addr;
  }

  panic("bmap:␣out␣of␣range");
}

void
s5_ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;
  struct s5_superblock *s5sb;
  struct s5_inode *s5ip;

  s5ip = ip->i_private;

  s5sb = sb[ip->dev].fs_info;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquire(&icache.lock);
  while (ip->flags & I_BUSY)
    sleep(ip, &icache.lock);
  ip->flags |= I_BUSY;
  release(&icache.lock);

  if (!(ip->flags & I_VALID)) {
    bp = s5_ops.bread(ip->dev, IBLOCK(ip->inum, (*s5sb)));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(s5ip->addrs, dip->addrs, sizeof(s5ip->addrs));
    s5_ops.brelse(bp);
    ip->flags |= I_VALID;
    if (ip->type == 0)
      panic("ilock: no type");
  }
}

int
s5_readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
      return -1;
    return devsw[ip->major].read(ip, dst, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = ip->fs_t->ops->bread(ip->dev, ip->iops->bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    ip->fs_t->ops->brelse(bp);
  }
  return n;
}

int
s5_writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip, src, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > MAXFILE*BSIZE)
    return -1;

  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = s5_ops.bread(ip->dev, s5_iops.bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(bp->data + off%BSIZE, src, m);
    log_write(bp);
    s5_ops.brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    s5_iops.iupdate(ip);
  }
  return n;
}

int
s5_isdirempty(struct inode *dp)
{
  int off;
  struct dirent de;

  for(off=2*sizeof(de); off<dp->size; off+=sizeof(de)){
    if(s5_iops.readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("isdirempty: readi");
    if(de.inum != 0)
      return 0;
  }
  return 1;
}

int
s5_unlink(struct inode *dp, uint off)
{
  struct dirent de;

  memset(&de, 0, sizeof(de));
  if(dp->iops->writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    return -1;

  return 0;
}

int
s5_namecmp(const char *s, const char *t)
{
  return strncmp(s, t, DIRSIZ);
}

int
s5_fill_inode(struct inode *ip) {
  struct s5_inode *s5ip;

  s5ip = alloc_s5_inode();
  if (!s5ip) {
    panic("No s5 inode available");
  }

  ip->i_private = s5ip;

  return 1;
}

struct inode*
s5_iget(uint dev, uint inum)
{
  return iget(dev, inum, &s5_fill_inode);
}
```

# APPENDIX C – src/ext2.c

```
#include "types.h"
#include "defs.h"
#include "param.h"
#include "stat.h"
#include "mmu.h"
#include "proc.h"
#include "spinlock.h"
#include "vfs.h"
#include "buf.h"
#include "file.h"
#include "vfsmount.h"
#include "ext2.h"
#include "find_bits.h"

#define in_range(b, first, len) ((b) >= (first) && (b) <= (first) + (len) - 1)
#define ext2_find_next_zero_bit find_next_zero_bit
#define ext2_test_bit test_bit
#define ext2_set_bit_atomic test_and_set_bit
#define ext2_clear_bit_atomic test_and_clear_bit

static int ext2_block_to_path(struct inode *inode,
                    long i_block, int offsets[4], int *boundary);

static struct ext2_inode * ext2_get_inode(struct superblock *sb,
                                   uint ino, struct buf **bh);

static struct buf * read_block_bitmap(struct superblock *sb,
                                   unsigned int block_group);

static void group_adjust_blocks(struct superblock *sb, int group_no,
                               struct ext2_group_desc *desc, struct buf *bh,
                               int count);

typedef struct {
  uint32 *p;
  uint32 key;
  struct buf *bh;
} Indirect;

static inline void
add_chain(Indirect *p, struct buf *bh, uint32 *v)
{
  p->key = *(p->p = v);
  p->bh = bh;
}

static inline int verify_chain(Indirect *from, Indirect *to)
{
  while (from <= to && from->key == *from->p)
    from++;
  return (from > to);
}


```

```
static struct {
  struct spinlock lock;
  struct ext2_inode_info ei[NINODE];
} ext2_ei_pool; // It is a Pool of S5 Superblock Filesystems

struct ext2_inode_info*
alloc_ext2_inode_info()
{
  struct ext2_inode_info *ei;

  acquire(&ext2_ei_pool.lock);
  for (ei = &ext2_ei_pool.ei[0]; ei < &ext2_ei_pool.ei[NINODE]; ei++) {
    if (ei->flags == INODE_FREE) {
      ei->flags |= INODE_USED;
      release(&ext2_ei_pool.lock);

      return ei;
    }
  }
  release(&ext2_ei_pool.lock);

  return 0;
}

static struct {
  struct spinlock lock;
  struct ext2_sb_info sb[MAXVFSSIZE];
} ext2_sb_pool; // It is a Pool of S5 Superblock Filesystems

struct ext2_sb_info*
alloc_ext2_sb()
{
  struct ext2_sb_info *sb;

  acquire(&ext2_sb_pool.lock);
  for (sb = &ext2_sb_pool.sb[0]; sb < &ext2_sb_pool.sb[MAXVFSSIZE]; sb++) {
    if (sb->flags == SB_FREE) {
      sb->flags |= SB_USED;
      release(&ext2_sb_pool.lock);

      return sb;
    }
  }
  release(&ext2_sb_pool.lock);

  return 0;
}

struct vfs_operations ext2_ops = {
  .fs_init = &ext2fs_init,
  .mount   = &ext2_mount,
  .unmount = &ext2_unmount,
  .getroot = &ext2_getroot,
  .readsb  = &ext2_readsb,
  .ialloc  = &ext2_ialloc,
  .balloc  = &ext2_balloc,
  .bzero   = &ext2_bzero,
  .bfree   = &ext2_bfree,
  .brelse  = &brelse,
  .bwrite  = &bwrite,
  .bread    = &bread,
  .namecmp = &ext2_namecmp
};

struct inode_operations ext2_iops = {
  .dirlookup  = &ext2_dirlookup,
  .iupdate    = &ext2_iupdate,
  .itrunc     = &ext2_itrunc,
  .cleanup    = &ext2_cleanup,
  .bmap       = &ext2_bmap,
  .ilock      = &ext2_ilock,
  .iunlock    = &generic_iunlock,
  .stati      = &generic_stati,
  .readi      = &generic_readi,
  .writei     = &ext2_writei,
  .dirlink    = &ext2_dirlink,
  .unlink     = &ext2_unlink,
  .isdirempty = &ext2_isdirempty
};

struct filesystem_type ext2fs = {
  .name = "ext2",
  .ops = &ext2_ops,
  .iops = &ext2_iops
};

int
initext2fs(void)
{
  initlock(&ext2_sb_pool.lock, "ext2_sb_pool");
  /* initlock(&ext2_inode_pool.lock, "ext2_inode_pool"); */
  return register_fs(&ext2fs);
}

int
ext2fs_init(void)
{
  return 0;
}

int
ext2_mount(struct inode *devi, struct inode *ip)
{
  struct mntentry *mp;

  // Read the Superblock
  ext2_ops.readsb(devi->minor, &sb[devi->minor]);

  // Read the root device
  struct inode *devrtip = ext2_ops.getroot(devi->major, devi->minor);

  acquire(&mtable.lock);
  for (mp = &mtable.mpoint[0]; mp < &mtable.mpoint[MOUNTSIZE]; mp++) {
    // This slot is available
    if (mp->flag == 0) {
found_slot:
      mp->dev = devi->minor;
      mp->m_inode = ip;
      mp->pdata = &sb[devi->minor];
      mp->flag |= M_USED;
```

```c
      mp->m_rtinode = devrtip;

      release(&mtable.lock);

      return 0;
    } else {
      // The disk is already mounted
      if (mp->dev == devi->minor) {
        release(&mtable.lock);
        return -1;
      }

      if (ip->dev == mp->m_inode->dev && ip->inum == mp->m_inode->inum)
        goto found_slot;
    }
  }
  release(&mtable.lock);

  return -1;
}

int
ext2_unmount(struct inode *devi)
{
  panic("ext2␣unmount␣op␣not␣defined");
  return 0;
}

struct inode *
ext2_getroot(int major, int minor)
{
  return ext2_iget(minor, EXT2_ROOT_INO);
}

static inline int
test_root(int a, int b)
{
  int num = b;

  while (a > num)
    num *= b;
  return num == a;
}

static int
ext2_group_sparse(int group)
{
  if (group <= 1)
    return 1;
  return (test_root(group, 3) || test_root(group, 5) ||
      test_root(group, 7));
}

/**
 * ext2_bg_has_super - number of blocks used by the superblock in group
 * @sb: superblock for filesystem
 * @group: group number to check
 *
 * Return the number of blocks used by the superblock (primary or backup)
 * in this group. Currently this will be only 0 or 1.
 */
int
ext2_bg_has_super(struct superblock *sb, int group)
{
  if (EXT2_HAS_RO_COMPAT_FEATURE(sb, EXT2_FEATURE_RO_COMPAT_SPARSE_SUPER)&&
      !ext2_group_sparse(group))
    return 0;
  return 1;
}

struct ext2_group_desc *
ext2_get_group_desc(struct superblock * sb,
                    unsigned int block_group,
                    struct buf ** bh)
{
  unsigned long group_desc;
  unsigned long offset;
  struct ext2_group_desc * desc;
  struct ext2_sb_info *sbi = EXT2_SB(sb);

  if (block_group >= sbi->s_groups_count) {
    panic("Block group # is too large");
  }

  group_desc = block_group >> EXT2_DESC_PER_BLOCK_BITS(sb);
  offset = block_group & (EXT2_DESC_PER_BLOCK(sb) - 1);
  if (!sbi->s_group_desc[group_desc]) {
    panic("Accessing a group descriptor not loaded");
  }

  desc = (struct ext2_group_desc *) sbi->s_group_desc[group_desc]->data;
  if (bh) {
    *bh = sbi->s_group_desc[group_desc];
  }
  return desc + offset;
}

static unsigned long
descriptor_loc(struct superblock *sb,
               unsigned long logic_sb_block,
               int nr)
{
  unsigned long bg, first_meta_bg;
  int has_super = 0;

  first_meta_bg = EXT2_SB(sb)->s_es->s_first_meta_bg;

  if (!EXT2_HAS_INCOMPAT_FEATURE(sb, EXT2_FEATURE_INCOMPAT_META_BG) ||
      nr < first_meta_bg)
    return (logic_sb_block + nr + 1);
  bg = EXT2_SB(sb)->s_desc_per_block * nr;
  if (ext2_bg_has_super(sb, bg))
    has_super = 1;

  return ext2_group_first_block_no(sb, bg) + has_super;
}

void
ext2_readsb(int dev, struct superblock *sb)
{
  struct buf *bp;
  struct ext2_sb_info *sbi;
  struct ext2_superblock *es;
  uint32 blocksize = EXT2_MIN_BLKSIZE;
  int db_count, i;
  unsigned long block;
  unsigned long logic_sb_block = 1;
  unsigned long offset = 0;

  if((sb->flags & SB_NOT_LOADED) == 0) {
    sbi = alloc_ext2_sb(); // Allocate a new S5 sb struct to the superblock.
  } else{
    sbi = sb->fs_info;
  }

  // These sets are needed because of bread
  sb->major = IDEMAJOR;
  sb->minor = dev;
  sb_set_blocksize(sb, blocksize);
  sb->fs_info = sbi;

  bp = ext2_ops.bread(dev, logic_sb_block); // Read the 1024 bytes starting from the byte 1024
  es = (struct ext2_superblock *)bp->data;

  sbi->s_es = es;
  sbi->s_sbh = bp;
  if (es->s_magic != EXT2_SUPER_MAGIC) {
    ext2_ops.brelse(bp);
    panic("Try to mount a non ext2 fs as an ext2 fs");
  }

  blocksize = EXT2_MIN_BLKSIZE << es->s_log_block_size;

  /* If the blocksize doesn't match, re-read the thing.. */
  if (sb->blocksize != blocksize) {
    ext2_ops.brelse(bp);

    sb_set_blocksize(sb, blocksize);

    logic_sb_block = EXT2_MIN_BLKSIZE / blocksize;
    offset = EXT2_MIN_BLKSIZE % blocksize;
    bp = ext2_ops.bread(dev, logic_sb_block);

    if (!bp) {
      panic("Error on second ext2 superblock read");
    }

    es = (struct ext2_superblock *) (((char *)bp->data) + offset);
    sbi->s_es = es;

    if (es->s_magic != EXT2_SUPER_MAGIC) {
      panic("error: ext2 magic mismatch");
    }
  }

  if (es->s_rev_level == EXT2_GOOD_OLD_REV) {
    sbi->s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
    sbi->s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
  } else {
    sbi->s_inode_size = es->s_inode_size;
    sbi->s_first_ino = es->s_first_ino;
  }

  sbi->s_blocks_per_group = es->s_blocks_per_group;
  sbi->s_inodes_per_group = es->s_inodes_per_group;

  sbi->s_inodes_per_block = sb->blocksize / sbi->s_inode_size;
  sbi->s_itb_per_group = sbi->s_inodes_per_group / sbi->s_inodes_per_block;
  sbi->s_desc_per_block = sb->blocksize / sizeof(struct ext2_group_desc);

  sbi->s_addr_per_block_bits = ilog2(EXT2_ADDR_PER_BLOCK(sb));
  sbi->s_desc_per_block_bits = ilog2(EXT2_DESC_PER_BLOCK(sb));

  if (sbi->s_blocks_per_group > sb->blocksize * 8) {
    panic("error:␣#blocks␣per␣group␣too␣big");
  }

  if (sbi->s_inodes_per_group > sb->blocksize * 8) {
    panic("error:␣#inodes␣per␣group␣too␣big");
  }

  sbi->s_groups_count = ((es->s_blocks_count -
                          es->s_first_data_block - 1)
                            / sbi->s_blocks_per_group) + 1;
  db_count = (sbi->s_groups_count + sbi->s_desc_per_block - 1) /
                sbi->s_desc_per_block;

  if (db_count > EXT2_MAX_BGC) {
    panic("error:␣not␣enough␣memory␣to␣storage␣s_group_desc.␣Consider␣change␣the␣EXT2_MAX_BGC␣constan
  }

  /* bgl_lock_init(sbi->s_blockgroup_lock); */

  for (i = 0; i < db_count; i++) {
    block = descriptor_loc(sb, logic_sb_block, i);
    sbi->s_group_desc[i] = ext2_ops.bread(dev, block);
    if (!sbi->s_group_desc[i]) {
      panic("Error␣on␣read␣ext2␣␣group␣descriptor");
    }
  }

  sbi->s_gdb_count = db_count;
}

/*
 * Read the inode allocation bitmap for a given block_group, reading
 * into the specified slot in the superblock's bitmap cache.
 *
 * Return buffer_head of bitmap on success or NULL.
 */
static struct buf *
read_inode_bitmap(struct superblock * sb, unsigned long block_group)
{
  struct ext2_group_desc *desc;
  struct buf *bh = 0;

  desc = ext2_get_group_desc(sb, block_group, 0);
  if (!desc)
    panic("error␣on␣read␣ext2␣inode␣bitmap");

  bh = ext2_ops.bread(sb->minor, desc->bg_inode_bitmap);
  if (!bh)
    panic("error on read ext2 inode bitmap");
  return bh;
}

/**
 * It is a dummy implementation of ialloc.
 * Current Linux implementation uses an heuristic to alloc inodes
 * in the best place.
 * Our implementation will take an linear search over the inode bitmap
 * and get the first free inode.
 */
struct inode*
ext2_ialloc(uint dev, short type)
{
  int i, group;
  unsigned long ino;
  struct ext2_sb_info *sbi;
  struct buf *bitmap_bh = 0;
  struct buf *bh2;
  struct buf *ibh;
  struct ext2_group_desc *gdp;
  struct ext2_inode *raw_inode;

  sbi = EXT2_SB(&sb[dev]);

  group = 0;
  for(i = 0; i < sbi->s_groups_count; i++) {
    gdp = ext2_get_group_desc(&sb[dev], group, &bh2);

    if (bitmap_bh)
      ext2_ops.brelse(bitmap_bh);

    bitmap_bh = read_inode_bitmap(&sb[dev], group);
    ino = 0;

repeat_in_this_group:
    ino = ext2_find_next_zero_bit((unsigned long *)bitmap_bh->data,
                                  EXT2_INODES_PER_GROUP(&sb[dev]), ino);
    if (ino >= EXT2_INODES_PER_GROUP(&sb[dev])) {
      if (++group == sbi->s_groups_count)
        group = 0;
      continue;
    }
    if (ext2_set_bit_atomic(ino, (unsigned long *)bitmap_bh->data)) {
      /* we lost this inode */
      if (++ino >= EXT2_INODES_PER_GROUP(&sb[dev])) {
        /* this group is exhausted, try next group */
        if (++group == sbi->s_groups_count)
          group = 0;
        continue;
      }
      /* try to find free inode in the same group */
      goto repeat_in_this_group;
    }
    goto got;
  }

  /*
```

```
   * Scanned all blockgroups.
   */
  panic("no␣space␣to␣alloc␣inode");

got:
  ext2_ops.bwrite(bitmap_bh);
  ext2_ops.brelse(bitmap_bh);

  ino += group * EXT2_INODES_PER_GROUP(&sb[dev]) + 1;
  if (ino < EXT2_FIRST_INO(&sb[dev]) || ino > sbi->s_es->s_inodes_count) {
    panic("ext2␣invalid␣inode␣number␣allocated");
  }

  /* spin_lock(sb_bgl_lock(sbi, group)); */
  gdp->bg_free_inodes_count -= 1;
  /* spin_unlock(sb_bgl_lock(sbi, group)); */

  ext2_ops.bwrite(bh2);

  raw_inode = ext2_get_inode(&sb[dev], ino, &ibh);

  // Erase the current inode
  memset(raw_inode, 0, sbi->s_inode_size);
  // Translate the xv6 to inode type type
  if (type == T_DIR) {
    raw_inode->i_mode = S_IFDIR;
  } else if (type == T_FILE) {
    raw_inode->i_mode = S_IFREG;
  } else {
    // We did not treat char and block devices with difference.
    panic("ext2:␣invalid␣inode␣mode");
  }

  ext2_ops.bwrite(ibh);
  ext2_ops.brelse(ibh);

  return ext2_iget(dev, ino);
}

uint
ext2_balloc(uint dev)
{
  panic("ext2␣balloc␣op␣not␣defined");
}

void
ext2_bzero(int dev, int bno)
{
  panic("ext2␣bzero␣op␣not␣defined");
}

void
ext2_bfree(int dev, uint b)
{
  panic("ext2␣bfree␣op␣not␣defined");
}

struct inode*
ext2_dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum, currblk;
  struct ext2_dir_entry_2 *de;
  struct buf *bh;
  int namelen = strlen(name);

  for (off = 0; off < dp->size;) {
    currblk = off / sb[dp->dev].blocksize;

    bh = ext2_ops.bread(dp->dev, ext2_iops.bmap(dp, currblk));

    de = (struct ext2_dir_entry_2 *) (bh->data + (off % sb[dp->dev].blocksize));

    if(de->inode == 0 || de->name_len != namelen) {
      off += de->rec_len;
      ext2_ops.brelse(bh);
      continue;
    }

    if(strncmp(name, de->name, de->name_len) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de->inode;
      ext2_ops.brelse(bh);
      return ext2_iget(dp->dev, inum);
    }
    off += de->rec_len;
    ext2_ops.brelse(bh);
  }

  return 0;
}

void
ext2_iupdate(struct inode *ip)
{
  struct buf *bp;
  struct ext2_inode_info *ei;
  struct ext2_inode *raw_inode;

  ei = ip->i_private;
  raw_inode = ext2_get_inode(&sb[ip->dev], ip->inum, &bp);

  raw_inode->i_mode = ei->i_ei.i_mode;
  raw_inode->i_blocks = ei->i_ei.i_blocks;
  raw_inode->i_links_count = ip->nlink;
  memmove(raw_inode->i_block, ei->i_ei.i_block, sizeof(ei->i_ei.i_block));
  raw_inode->i_size = ip->size;

  ext2_ops.bwrite(bp);
  ext2_ops.brelse(bp);
}

/**
 * ext2_free_blocks() -- Free given blocks and update quota and i_blocks
 * @inode:     inode
 * @block:     start physical block to free
 * @count:     number of blocks to free
 */
void
```

```
ext2_free_blocks(struct inode * inode, unsigned long block,
                  unsigned long count)
{
  struct buf *bitmap_bh = 0;
  struct buf * bh2;
  unsigned long block_group;
  unsigned long bit;
  unsigned long i;
  unsigned long overflow;
  struct superblock * superb = &sb[inode->dev];
  struct ext2_sb_info * sbi = EXT2_SB(&sb[inode->dev]);
  struct ext2_group_desc * desc;
  struct ext2_superblock * es = sbi->s_es;
  unsigned freed = 0, group_freed;

  if (block < es->s_first_data_block ||
      block + count < block ||
      block + count > es->s_blocks_count) {
    panic("ext2␣free␣blocks␣in␣not␣datazone");
  }

do_more:
  overflow = 0;
  block_group = (block - es->s_first_data_block) / EXT2_BLOCKS_PER_GROUP(superb);
  bit = (block - es->s_first_data_block) % EXT2_BLOCKS_PER_GROUP(superb);
  /*
   * Check to see if we are freeing blocks across a group
   * boundary.
   */
  if (bit + count > EXT2_BLOCKS_PER_GROUP(superb)) {
    overflow = bit + count - EXT2_BLOCKS_PER_GROUP(superb);
    count -= overflow;
  }
  if (bitmap_bh)
    brelse(bitmap_bh);

  bitmap_bh = read_block_bitmap(superb, block_group);
  if (!bitmap_bh)
    goto error_return;

  desc = ext2_get_group_desc(superb, block_group, &bh2);
  if (!desc)
    goto error_return;

  if (in_range (desc->bg_block_bitmap, block, count) ||
      in_range (desc->bg_inode_bitmap, block, count) ||
      in_range (block, desc->bg_inode_table,
                sbi->s_itb_per_group) ||
      in_range (block + count - 1, desc->bg_inode_table,
                sbi->s_itb_per_group)) {
    panic("Freeing␣blocks␣on␣system␣zone");
    goto error_return;
  }

  for (i = 0, group_freed = 0; i < count; i++) {
    if (!ext2_clear_bit_atomic(bit + i, (unsigned long *)bitmap_bh->data)) {
      panic("ext2␣bit␣already␣cleared␣for␣block");
    } else {
      group_freed++;
    }
  }

  ext2_ops.bwrite(bitmap_bh);
  group_adjust_blocks(superb, block_group, desc, bh2, group_freed);
  freed += group_freed;

  if (overflow) {
    block += count;
    count = overflow;
    goto do_more;
  }
error_return:
  ext2_ops.brelse(bitmap_bh);
}

/**
 * ext2_free_data - free a list of data blocks
 * @inode: inode we are dealing with
 * @p: array of block numbers
 * @q: points immediately past the end of array
 *
 * We are freeing all blocks referred from that array (numbers are
 * stored as little-endian 32-bit) and updating @inode->i_blocks
 * appropriately.
 */
static inline void
ext2_free_data(struct inode *inode, uint32 *p, uint32 *q)
{
  unsigned long block_to_free = 0, count = 0;
  unsigned long nr;

  for ( ; p < q ; p++) {
    nr = *p;
    if (nr) {
      *p = 0;
      /* accumulate blocks to free if they're contiguous */
      if (count == 0)
        goto free_this;
      else if (block_to_free == nr - count)
        count++;
      else {
        ext2_free_blocks(inode, block_to_free, count);
        /* mark_inode_dirty(inode); */
free_this:
        block_to_free = nr;
        count = 1;
      }
    }
  }
  if (count > 0) {
    ext2_free_blocks(inode, block_to_free, count);
    /* mark_inode_dirty(inode); */
  }
}

/**
 * ext2_free_branches - free an array of branches
 * @inode: inode we are dealing with
 * @p: array of block numbers
 * @q: pointer immediately past the end of array
```

```c
 * @depth: depth of the branches to free
 */
static void
ext2_free_branches(struct inode *inode, uint32 *p, uint32 *q, int depth)
{
  struct buf * bh;
  unsigned long nr;

  if (depth--) {
    int addr_per_block = EXT2_ADDR_PER_BLOCK(&sb[inode->dev]);
    for ( ; p < q ; p++) {
      nr = *p;
      if (!nr)
        continue;
      *p = 0;
      bh = ext2_ops.bread(inode->dev, nr);
      /*
       * A read failure? Report error and clear slot
       * (should be rare).
       */
      if (!bh) {
        panic("ext2␣block␣read␣failure");
        continue;
      }
      ext2_free_branches(inode,
                         (uint32*)bh->data,
                         (uint32*)bh->data + addr_per_block,
                         depth);
      ext2_ops.brelse(bh);
      ext2_free_blocks(inode, nr, 1);
      /* mark_inode_dirty(inode); */
    }
  } else {
    ext2_free_data(inode, p, q);
  }
}

static void
ext2_release_inode(struct superblock *sb, int group, int dir)
{
  struct ext2_group_desc * desc;
  struct buf *bh;

  desc = ext2_get_group_desc(sb, group, &bh);
  if (!desc) {
    panic("Error␣on␣get␣group␣descriptor");
    return;
  }

  /* spin_lock(sb_bgl_lock(EXT2_SB(sb), group)); */
  desc->bg_free_inodes_count += 1;
  if (dir)
    desc->bg_used_dirs_count -= 1;
  /* spin_unlock(sb_bgl_lock(EXT2_SB(sb), group)); */
  ext2_ops.bwrite(bh);
}

/*
 * NOTE! When we get the inode, we're the only people
 * that have access to it, and as such there are no
 * race conditions we have to worry about. The inode
 * is not on the hash-lists , and it cannot be reached
 * through the filesystem because the directory entry
 * has been deleted earlier.
 *
 * HOWEVER: we must make sure that we get no aliases ,
 * which means that we have to call "clear_inode()"
 * _before_ we mark the inode not in use in the inode
 * bitmaps. Otherwise a newly created file might use
 * the same inode number (not actually the same pointer
 * though), and then we'd have two inodes sharing the
 * same inode number and space on the harddisk.
 */
void
ext2_free_inode (struct inode * inode)
{
  struct superblock *superb = &sb[inode->dev];
  int is_directory;
  unsigned long ino;
  struct buf *bitmap_bh;
  unsigned long block_group;
  unsigned long bit;
  struct ext2_superblock * es;
  struct ext2_inode_info *ei;

  ino = inode->inum;
  ei = inode->i_private;

  es = EXT2_SB(superb)->s_es;
  is_directory = S_ISDIR(ei->i_ei.i_mode);

  if (ino < EXT2_FIRST_INO(superb) ||
      ino > es->s_inodes_count) {
    panic("ext2␣reserved␣or␣non␣existent␣inode");
    return;
  }

  block_group = (ino - 1) / EXT2_INODES_PER_GROUP(superb);
  bit = (ino - 1) % EXT2_INODES_PER_GROUP(superb);
  bitmap_bh = read_inode_bitmap(superb, block_group);
  if (!bitmap_bh)
    return;

  /* Ok, now we can actually update the inode bitmaps.. */
  if (!ext2_clear_bit_atomic(bit, (void *) bitmap_bh->data))
    panic("ext2␣bit␣already␣cleared");
  else
    ext2_release_inode(superb, block_group, is_directory);

  ext2_ops.bwrite(bitmap_bh);
  ext2_ops.brelse(bitmap_bh);
}

void
ext2_itrunc(struct inode *ip)
{
  uint32 *i_data;
  int offsets[4];
  uint32 nr = 0;
  int n;
  long iblock;
  unsigned blocksize;
  blocksize = sb[ip->dev].blocksize;
  iblock = (blocksize - 1) >> EXT2_BLOCK_SIZE_BITS(&sb[ip->dev]);
  n = ext2_block_to_path(ip, iblock, offsets, 0);

  struct ext2_inode_info *ei = ip->i_private;

  i_data = ei->i_ei.i_block;

  if (n == 0)
    return;

  /* lock block here */

  if (n == 1) {
    ext2_free_data(ip, i_data+offsets[0],
                       i_data + EXT2_NDIR_BLOCKS);
  }

  /* Kill the remaining (whole) subtrees */
  switch (offsets[0]) {
    default:
      nr = i_data[EXT2_IND_BLOCK];
      if (nr) {
        i_data[EXT2_IND_BLOCK] = 0;
        /* mark_inode_dirty(inode); */
        ext2_free_branches(ip, &nr, &nr+1, 1);
      }
    case EXT2_IND_BLOCK:
      nr = i_data[EXT2_DIND_BLOCK];
      if (nr) {
        i_data[EXT2_DIND_BLOCK] = 0;
        /* mark_inode_dirty(inode); */
        ext2_free_branches(ip, &nr, &nr+1, 2);
      }
    case EXT2_DIND_BLOCK:
      nr = i_data[EXT2_TIND_BLOCK];
      if (nr) {
        i_data[EXT2_TIND_BLOCK] = 0;
        /* mark_inode_dirty(inode); */
        ext2_free_branches(ip, &nr, &nr+1, 3);
      }
    case EXT2_TIND_BLOCK:
      ;
  }

  // unlock the inode here
  ext2_free_inode(ip);

  ext2_iops.iupdate(ip);
}

void
ext2_cleanup(struct inode *ip)
{
  memset(ip->i_private, 0, sizeof(struct ext2_inode_info));
}

/**
```

```
 * ext2_block_to_path - parse the block number into array of offsets
 * @inode: inode in question (we are only interested in its superblock)
 * @i_block: block number to be parsed
 * @offsets: array to store the offsets in
 * @boundary: set this non-zero if the referred-to block is likely to be
 *            followed (on disk) by an indirect block.
 * To store the locations of file's data ext2 uses a data structure common
 * for UNIX filesystems - tree of pointers anchored in the inode, with
 * data blocks at leaves and indirect blocks in intermediate nodes.
 * This function translates the block number into path in that tree -
 * return value is the path length and @offsets[n] is the offset of
 * pointer to (n+1)th node in the nth one. If @block is out of range
 * (negative or too large) warning is printed and zero returned.
 *
 * Note: function doesn't find node addresses, so no IO is needed. All
 * we need to know is the capacity of indirect blocks (taken from the
 * superblock).
 */

/*
 * Portability note: the last comparison (check that we fit into triple
 * indirect block) is spelled differently, because otherwise on an
 * architecture with 32-bit longs and 8Kb pages we might get into trouble
 * if our filesystem had 8Kb blocks. We might use long long, but that would
 * kill us on x86. Oh, well, at least the sign propagation does not matter -
 * i_block would have to be negative in the very beginning, so we would not
 * get there at all.
 */

static int
ext2_block_to_path(struct inode *inode,
                   long i_block, int offsets[4], int *boundary)
{
  int ptrs = EXT2_ADDR_PER_BLOCK(&sb[inode->dev]);
  int ptrs_bits = EXT2_ADDR_PER_BLOCK_BITS(&sb[inode->dev]);
  const long direct_blocks = EXT2_NDIR_BLOCKS,
        indirect_blocks = ptrs,
        double_blocks = (1 << (ptrs_bits * 2));
  int n = 0;
  int final = 0;

  if (i_block < 0) {
    panic("block_to_path␣invalid␣block␣num");
  } else if (i_block < direct_blocks) {
    offsets[n++] = i_block;
    final = direct_blocks;
  } else if ((i_block -= direct_blocks) < indirect_blocks) {
    offsets[n++] = EXT2_IND_BLOCK;
    offsets[n++] = i_block;
    final = ptrs;
  } else if ((i_block -= indirect_blocks) < double_blocks) {
    offsets[n++] = EXT2_DIND_BLOCK;
    offsets[n++] = i_block >> ptrs_bits;
    offsets[n++] = i_block & (ptrs - 1);
    final = ptrs;
  } else if (((i_block -= double_blocks) >> (ptrs_bits * 2)) < ptrs) {
    offsets[n++] = EXT2_TIND_BLOCK;
    offsets[n++] = i_block >> (ptrs_bits * 2);
    offsets[n++] = (i_block >> ptrs_bits) & (ptrs - 1);
    offsets[n++] = i_block & (ptrs - 1);
    final = ptrs;
  } else {
    panic("This␣block␣is␣out␣of␣bounds␣from␣this␣ext2␣fs");
  }

  if (boundary)
    *boundary = final - 1 - (i_block & (ptrs - 1));

  return n;
}

static void
ext2_update_branch(struct inode *inode, uint bn, Indirect *chain)
{
  int ptrs = EXT2_ADDR_PER_BLOCK(&sb[inode->dev]);
  int ptrs_bits = EXT2_ADDR_PER_BLOCK_BITS(&sb[inode->dev]);
  const long direct_blocks = EXT2_NDIR_BLOCKS,
        indirect_blocks = ptrs,
        double_blocks = (1 << (ptrs_bits * 2));
  struct ext2_inode_info *ei;

  ei = inode->i_private;

  // Update inode block
  if (bn < 0) {
    panic("block_to_path␣invalid␣block␣num");
  } else if (bn < direct_blocks) {
    if (ei->i_ei.i_block[bn] == 0)
      ei->i_ei.i_block[bn] = chain[0].key;
  } else if ((bn -= direct_blocks) < indirect_blocks) {
    if (ei->i_ei.i_block[EXT2_IND_BLOCK] == 0)
      ei->i_ei.i_block[EXT2_IND_BLOCK] = chain[0].key;
  } else if ((bn -= indirect_blocks) < double_blocks) {
    if (ei->i_ei.i_block[EXT2_DIND_BLOCK] == 0)
      ei->i_ei.i_block[EXT2_DIND_BLOCK] = chain[0].key;
  } else if (((bn -= double_blocks) >> (ptrs_bits * 2)) < ptrs) {
    if (ei->i_ei.i_block[EXT2_TIND_BLOCK] == 0)
      ei->i_ei.i_block[EXT2_TIND_BLOCK] = chain[0].key;
  } else {
    panic("This␣block␣is␣out␣of␣bounds␣from␣this␣ext2␣fs");
  }

  return;
}

/**
 * ext2_get_branch - read the chain of indirect blocks leading to data
 * @inode: inode in question
 * @depth: depth of the chain (1 - direct pointer, etc.)
 * @offsets: offsets of pointers in inode/indirect blocks
 * @chain: place to store the result
 * @err: here we store the error value
 *
 * Function fills the array of triples <key, p, bh> and returns %NULL
 * if everything went OK or the pointer to the last filled triple
 * (incomplete one) otherwise. Upon the return chain[i].key contains
 * the number of (i+1)-th block in the chain (as it is stored in memory,
 * i.e. little-endian 32-bit), chain[i].p contains the address of that
 * number (it points into struct inode for i==0 and into the bh->b_data
 * for i>0) and chain[i].bh points to the buffer_head of i-th indirect
 * block for i>0 and NULL for i==0. In other words, it holds the block
 * numbers of the chain, addresses they were taken from (and where we can
 * verify that chain did not change) and buffer_heads hosting these
 * numbers.
 *
 * Function stops when it stumbles upon zero pointer (absent block)
 *  (pointer to last triple returned, *@err == 0)
 * or when it gets an IO error reading an indirect block
 *  (ditto, *@err == -EIO)
 * or when it notices that chain had been changed while it was reading
 *  (ditto, *@err == -EAGAIN)
 * or when it reads all @depth-1 indirect blocks successfully and finds
 * the whole chain, all way to the data (returns %NULL, *err == 0).
 */
static Indirect *ext2_get_branch(struct inode *inode,
                                 int depth,
                                 int *offsets,
                                 Indirect chain[4])
{
  Indirect *p = chain;
  struct buf *bh;
  struct ext2_inode_info *ei = inode->i_private;

  add_chain (chain, 0, ei->i_ei.i_block + *offsets);
  if (!p->key)
    goto no_block;
  while (--depth) {
    bh = ext2_ops.bread(inode->dev, p->key);
    if (!bh)
      panic("error␣on␣ext2_get_branch");
    if (!verify_chain(chain, p))
      panic("ext2_get_branch␣chain␣changed");
    add_chain(++p, bh, (uint32*)bh->data + *++offsets);
    if (!p->key)
      goto no_block;
  }
  return 0;

no_block:
  return p;
}

/**
 * ext2_find_near - find a place for allocation with sufficient locality
 * @inode: owner
 * @ind: descriptor of indirect block.
 *
 * This function returns the preferred place for block allocation.
 * It is used when heuristic for sequential allocation fails.
 * Rules are:
 *   + if there is a block to the left of our position - allocate near it.
 *   + if pointer will live in indirect block - allocate near that block.
 *   + if pointer will live in inode - allocate in the same cylinder group.
 *
 * In the latter case we colour the starting block by the callers PID to
 * prevent it from clashing with concurrent allocations for a different inode
 * in the same block group.   The PID is used here so that functionally related
 * files will be close-by on-disk.
 *
 * Caller must make sure that @ind is valid and will stay that way.
 */

static ext2_fsblk_t ext2_find_near(struct inode *inode, Indirect *ind)
{
  struct ext2_inode_info *ei = inode->i_private;
  uint32 *start = ind->bh ? (uint32 *) ind->bh->data : ei->i_ei.i_block;
  uint32 *p;
  ext2_fsblk_t bg_start;
  ext2_fsblk_t colour;
  ext2_grpblk_t i_block_group;

  /* Try to find previous block */
  for (p = ind->p - 1; p >= start; p--)
    if (*p)
      return *p;

  /* No such thing, so let's try location of indirect block */
  if (ind->bh)
    return ind->bh->blockno;

  /*
   * It is going to be referred from inode itself? OK, just put it into
   * the same cylinder group then.
   */
  i_block_group = (inode->inum - 1) / EXT2_INODES_PER_GROUP(&sb[inode->dev]);
  bg_start = ext2_group_first_block_no(&sb[inode->dev], i_block_group);
  colour = (proc->pid % 16) *
    (EXT2_BLOCKS_PER_GROUP(&sb[inode->dev]) / 16);
  return bg_start + colour;
}

static inline ext2_fsblk_t ext2_find_goal(struct inode *inode, long block,
    Indirect *partial)
{
  return ext2_find_near(inode, partial);
}

/**
 * ext2_blks_to_allocate: Look up the block map and count the number
 * of direct blocks need to be allocated for the given branch.
 *
 * @branch: chain of indirect blocks
 * @k: number of blocks need for indirect blocks
 * @blks: number of data blocks to be mapped.
 * @blocks_to_boundary:  the offset in the indirect block
 *
 * return the total number of blocks to be allocate, including the
 * direct and indirect blocks.
 */
static int
ext2_blks_to_allocate(Indirect * branch, int k, unsigned long blks,
                      int blocks_to_boundary)
{
  unsigned long count = 0;

  /*
   * Simple case, [t,d]Indirect block(s) has not allocated yet
   * then it's clear blocks on that path have not allocated
   */
  if (k > 0) {
    /* right now don't hanel cross boundary allocation */
    if (blks < blocks_to_boundary + 1)
      count += blks;
    else
      count += blocks_to_boundary + 1;
    return count;
  }

  count++;
  while (count < blks && count <= blocks_to_boundary
      && *(branch[0].p + count) == 0) {
    count++;
  }
  return count;
}

/*
 * Read the bitmap for a given block_group,and validate the
 * bits for block/inode/inode tables are set in the bitmaps
 *
 * Return buffer_head on success or NULL in case of failure.
 */
static struct buf *
read_block_bitmap(struct superblock *sb, unsigned int block_group)
{
  struct ext2_group_desc * desc;
  struct buf * bh;
  ext2_fsblk_t bitmap_blk;

  desc = ext2_get_group_desc(sb, block_group, 0);
  if (!desc)
    return 0;
  bitmap_blk = desc->bg_block_bitmap;
  bh = ext2_ops.bread(sb->minor, bitmap_blk);
  if (!bh) {
    return 0;
  }

  /* ext2_valid_block_bitmap(sb, desc, block_group, bh); */
  /*
   * file system mounted not to panic on error, continue with corrupt
   * bitmap
   */
  return bh;
}

/**
 * bitmap_search_next_usable_block()
 * @start:  the starting block (group relative) of the search
 * @bh:     bufferhead contains the block group bitmap
 * @maxblocks:  the ending block (group relative) of the reservation
 *
 * The bitmap search --- search forward through the actual bitmap on disk until
 * we find a bit free.
 */
static ext2_grpblk_t
bitmap_search_next_usable_block(ext2_grpblk_t start, struct buf *bh,
                                ext2_grpblk_t maxblocks)
{
  ext2_grpblk_t next;

  next = ext2_find_next_zero_bit((unsigned long *)bh->data, maxblocks, start);
  if (next >= maxblocks)
    return -1;
  return next;
}

/**
 * find_next_usable_block()
 * @start:  the starting block (group relative) to find next
 *     allocatable block in bitmap.
 * @bh:    bufferhead contains the block group bitmap
 * @maxblocks:  the ending block (group relative) for the search
 *
 * Find an allocatable block in a bitmap.  We perform the "most
 * appropriate allocation" algorithm of looking for a free block near
 * the initial goal; then for a free byte somewhere in the bitmap;
 * then for any free bit in the bitmap.
 */
static ext2_grpblk_t
find_next_usable_block(int start, struct buf *bh, int maxblocks)
{
  ext2_grpblk_t here, next;
  char *p, *r;

  if (start > 0) {
    /*
     * The goal was occupied; search forward for a free
     * block within the next XX blocks.
     *
     * end_goal is more or less random, but it has to be
     * less than EXT2_BLOCKS_PER_GROUP. Aligning up to the
     * next 64-bit boundary is simple..
     */
    ext2_grpblk_t end_goal = (start + 63) & ~63;
    if (end_goal > maxblocks)
      end_goal = maxblocks;
    here = ext2_find_next_zero_bit((unsigned long *)bh->data, end_goal, start);
    if (here < end_goal)
      return here;
  }

  here = start;
  if (here < 0)
    here = 0;

  p = ((char *)bh->data) + (here >> 3);
  r = memscan(p, 0, ((maxblocks + 7) >> 3) - (here >> 3));
  next = (r - ((char *)bh->data)) << 3;

  if (next < maxblocks && next >= here)
    return next;

  here = bitmap_search_next_usable_block(here, bh, maxblocks);
  return here;
}

/**
 * ext2_try_to_allocate()
 * @sb:    superblock
 * @group:   given allocation block group
 * @bitmap_bh:   bufferhead holds the block bitmap
 * @grp_goal:   given target block within the group
 * @count:   target number of blocks to allocate
 * @my_rsv:   reservation window
 *
 * Attempt to allocate blocks within a give range. Set the range of allocation
 * first, then find the first free bit(s) from the bitmap (within the range),
 * and at last, allocate the blocks by claiming the found free bit as allocated.
 *
 * To set the range of this allocation:
 *  if there is a reservation window, only try to allocate block(s)
 *  from the file's own reservation window;
 *  Otherwise, the allocation range starts from the give goal block,
 *  ends at the block group's last block.
 *
 * If we failed to allocate the desired block then we may end up crossing to a
 * new bitmap.
 */
static int
ext2_try_to_allocate(struct superblock *sb, int group,
    struct buf *bitmap_bh, ext2_grpblk_t grp_goal,
    unsigned long *count)
{
  ext2_grpblk_t start, end;
  unsigned long num = 0;

  if (grp_goal > 0)
    start = grp_goal;
  else
    start = 0;
  end = EXT2_BLOCKS_PER_GROUP(sb);

repeat:
  if (grp_goal < 0) {
    grp_goal = find_next_usable_block(start, bitmap_bh, end);
    if (grp_goal < 0)
      goto fail_access;

    int i;

    for (i = 0; i < 7 && grp_goal > start &&
        !ext2_test_bit(grp_goal - 1, (unsigned long *)bitmap_bh->data);
        i++, grp_goal--)
      ;
  }
  start = grp_goal;

  if (ext2_set_bit_atomic(grp_goal,
                          (unsigned long *)bitmap_bh->data)) {
    /*
     * The block was allocated by another thread, or it was
     * allocated and then freed by another thread
     */
    start++;
    grp_goal++;
    if (start >= end)
      goto fail_access;
    goto repeat;
  }
  num++;
  grp_goal++;
  while (num < *count && grp_goal < end &&
         !ext2_set_bit_atomic(grp_goal, (unsigned long*)bitmap_bh->data)) {
    num++;
    grp_goal++;
  }
  *count = num;
  return grp_goal - num;
fail_access:
  *count = num;
  return -1;
}

static void
group_adjust_blocks(struct superblock *sb, int group_no,
                    struct ext2_group_desc *desc, struct buf *bh,
                    int count)
{
  if (count) {
    /* struct ext2_sb_info *sbi = EXT2_SB(sb); */
    unsigned free_blocks;

    /* spin_lock(sb_bgl_lock(sbi, group_no)); */
    free_blocks = desc->bg_free_blocks_count;
    desc->bg_free_blocks_count = free_blocks + count;
    /* spin_unlock(sb_bgl_lock(sbi, group_no)); */
    ext2_ops.bwrite(bh);
  }
}

/*
 * ext2_new_blocks() -- core block(s) allocation function
 * @inode:  file inode
 * @goal:  given target block(filesystem wide)
 * @count:  target number of blocks to allocate
 * @errp:  error code
 *
 * ext2_new_blocks uses a goal block to assist allocation.  If the goal is
 * free, or there is a free block within 32 blocks of the goal, that block
 * is allocated.  Otherwise a forward search is made for a free block; within
 * each block group the search first looks for an entire free byte in the block
 * bitmap, and then for any free bit if that fails.
 * This function also updates quota and i_blocks field.
 */
ext2_fsblk_t
ext2_new_blocks(struct inode *inode, ext2_fsblk_t goal,
                unsigned long *count, int *errp)
{
  struct buf *bitmap_bh = 0;
  struct buf *gdp_bh;
  int group_no;
  ext2_grpblk_t grp_target_blk; /* blockgroup relative goal block */
  ext2_grpblk_t grp_alloc_blk; /* blockgroup-relative allocated block*/
  ext2_fsblk_t ret_block;  /* filesyetem-wide allocated block */
  int bgi;   /* blockgroup iteration index */
  ext2_grpblk_t free_blocks; /* number of free blocks in a group */
  struct superblock *superb;
  struct ext2_group_desc *gdp;
  struct ext2_superblock *es;
  struct ext2_sb_info *sbi;
  unsigned long ngroups;
  unsigned long num = *count;

  *errp = -1;
  superb = &sb[inode->dev];

  sbi = EXT2_SB(superb);
  es = sbi->s_es;

  /* if (!ext2_has_free_blocks(sbi)) { */
  /*   *errp = -ENOSPC; */
  /*   goto out; */
  /* } */

  /*
   * First, test whether the goal block is free.
   */
  if (goal < es->s_first_data_block ||
      goal >= es->s_blocks_count) {
    goal = es->s_first_data_block;
  }

  group_no = (goal - es->s_first_data_block) / EXT2_BLOCKS_PER_GROUP(superb);
retry_alloc:
  gdp = ext2_get_group_desc(superb, group_no, &gdp_bh);
  if (!gdp)
    goto io_error;

  free_blocks = gdp->bg_free_blocks_count;

  if (free_blocks > 0) {
    grp_target_blk = ((goal - es->s_first_data_block) %
                      EXT2_BLOCKS_PER_GROUP(superb));
    bitmap_bh = read_block_bitmap(superb, group_no);
    if (!bitmap_bh)
      goto io_error;
    grp_alloc_blk = ext2_try_to_allocate(superb, group_no,
                                         bitmap_bh, grp_target_blk, &num);
    if (grp_alloc_blk >= 0)
      goto allocated;
  }

  ngroups = EXT2_SB(superb)->s_groups_count;

  /*
   * Now search the rest of the groups.  We assume that
   * group_no and gdp correctly point to the last group visited.
   */
  for (bgi = 0; bgi < ngroups; bgi++) {
    group_no++;
    if (group_no >= ngroups)
      group_no = 0;
    gdp = ext2_get_group_desc(superb, group_no, &gdp_bh);
    if (!gdp)
      goto io_error;

    free_blocks = gdp->bg_free_blocks_count;
    /*
     * skip this group (and avoid loading bitmap) if there
     * are no free blocks
     */
    if (!free_blocks)
      continue;

    ext2_ops.brelse(bitmap_bh);
    bitmap_bh = read_block_bitmap(superb, group_no);
    if (!bitmap_bh)
      goto io_error;
    /*
     * try to allocate block(s) from this group, without a goal(-1).
     */
    grp_alloc_blk = ext2_try_to_allocate(superb, group_no,
                                         bitmap_bh, -1, &num);
    if (grp_alloc_blk >= 0)
      goto allocated;
  }

  goto out;

allocated:

  ret_block = grp_alloc_blk + ext2_group_first_block_no(superb, group_no);

  if (in_range(gdp->bg_block_bitmap, ret_block, num) ||
      in_range(gdp->bg_inode_bitmap, ret_block, num) ||
      in_range(ret_block, gdp->bg_inode_table,
               EXT2_SB(superb)->s_itb_per_group)         ||
      in_range(ret_block + num - 1, gdp->bg_inode_table,
               EXT2_SB(superb)->s_itb_per_group)) {
    goto retry_alloc;
  }

  if (ret_block + num - 1 >= es->s_blocks_count) {
    panic("Error on ext2 block alloc");
  }

  group_adjust_blocks(superb, group_no, gdp, gdp_bh, -num);

  ext2_ops.bwrite(bitmap_bh);

  *errp = 0;
  ext2_ops.brelse(bitmap_bh);
  /* if (num < *count) { */
  /*   dquot_free_block_nodirty(inode, *count-num); */
  /*   mark_inode_dirty(inode); */
  /*   *count = num; */
  /* } */
  return ret_block;

io_error:
  *errp = -2;
out:
  /*
   * Undo the block allocation
   */
  /* if (!performed_allocation) { */
  /*   dquot_free_block_nodirty(inode, *count); */
  /*   mark_inode_dirty(inode); */
  /* } */
```

```c
  ext2_ops.brelse(bitmap_bh);
  return 0;
}

/**
 * ext2_alloc_blocks: multiple allocate blocks needed for a branch
 * @indirect_blks: the number of blocks need to allocate for indirect
 *    blocks
 *
 * @new_blocks: on return it will store the new block numbers for
 * the indirect blocks(if needed) and the first direct block,
 * @blks: on return it will store the total number of allocated
 *   direct blocks
 */
static int
ext2_alloc_blocks(struct inode *inode,
                  ext2_fsblk_t goal, int indirect_blks, int blks,
                  ext2_fsblk_t new_blocks[4], int *err)
{
  int target;
  unsigned long count = 0;
  int index = 0;
  ext2_fsblk_t current_block = 0;
  int ret = 0;

  /*
   * Here we try to allocate the requested multiple blocks at once,
   * on a best-effort basis.
   * To build a branch, we should allocate blocks for
   * the indirect blocks(if not allocated yet), and at least
   * the first direct block of this branch.  That's the
   * minimum number of blocks need to allocate(required)
   */
  target = blks + indirect_blks;

  while (1) {
    count = target;
    /* allocating blocks for indirect blocks and direct blocks */
    current_block = ext2_new_blocks(inode,goal,&count,err);
    if (*err)
      goto failed_out;

    target -= count;
    /* allocate blocks for indirect blocks */
    while (index < indirect_blks && count) {
      new_blocks[index++] = current_block++;
      count--;
    }

    if (count > 0)
      break;
  }

  /* save the new block number for the first direct block */
  new_blocks[index] = current_block;

  /* total number of blocks allocated for direct blocks */
  ret = count;
  *err = 0;
  return ret;
```

```
failed_out:
  panic("ext2␣error␣on␣ext2_alloc_blocks");
  return ret;
}

/**
 * ext2_alloc_branch - allocate and set up a chain of blocks.
 * @inode: owner
 * @num: depth of the chain (number of blocks to allocate)
 * @offsets: offsets (in the blocks) to store the pointers to next.
 * @branch: place to store the chain in.
 *
 * This function allocates @num blocks, zeroes out all but the last one,
 * links them into chain and (if we are synchronous) writes them to disk.
 * In other words, it prepares a branch that can be spliced onto the
 * inode. It stores the information about that chain in the branch[], in
 * the same format as ext2_get_branch() would do. We are calling it after
 * we had read the existing part of chain and partial points to the last
 * triple of that (one with zero ->key). Upon the exit we have the same
 * picture as after the successful ext2_get_block(), except that in one
 * place chain is disconnected - *branch->p is still zero (we did not
 * set the last link), but branch->key contains the number that should
 * be placed into *branch->p to fill that gap.
 *
 * If allocation fails we free all blocks we've allocated (and forget
 * their buffer_heads) and return the error value the from failed
 * ext2_alloc_block() (normally -ENOSPC). Otherwise we set the chain
 * as described above and return 0.
 */

static int
ext2_alloc_branch(struct inode *inode,
                  int indirect_blks, int *blks, ext2_fsblk_t goal,
                  int *offsets, Indirect *branch)
{
  int blocksize = sb[inode->dev].blocksize;
  int i, n = 0;
  int err = 0;
  struct buf *bh;
  int num;
  ext2_fsblk_t new_blocks[4];
  ext2_fsblk_t current_block;

  num = ext2_alloc_blocks(inode, goal, indirect_blks,
      *blks, new_blocks, &err);
  if (err)
    return err;

  branch[0].key = new_blocks[0];
  /*
   * metadata blocks and data blocks are allocated.
   */
  for (n = 1; n <= indirect_blks;  n++) {
    /*
     * Get buffer_head for parent block, zero it out
     * and set the pointer to new one, then send;
     * parent to disk.
     */
    bh = ext2_ops.bread(inode->dev, new_blocks[n-1]);
    if (!bh) {
```

```c
      goto failed;
    }
    branch[n].bh = bh;
    memset(bh->data, 0, blocksize);
    branch[n].p = (uint32 *) bh->data + offsets[n];
    branch[n].key = new_blocks[n];
    *branch[n].p = branch[n].key;
    if ( n == indirect_blks) {
      current_block = new_blocks[n];
      /*
       * End of chain, update the last new metablock of
       * the chain to point to the new allocated
       * data blocks numbers
       */
      for (i=1; i < num; i++)
        *(branch[n].p + i) = ++current_block;
    }
    ext2_ops.bwrite(bh);
  }
  *blks = num;
  return err;

failed:
  panic("ext2 error on allocate blocks branch");
  return err;
}

uint
ext2_bmap(struct inode *ip, uint bn)
{
  /* struct buf *bp; */
  int depth;
  Indirect chain[4];
  Indirect *partial;
  int offsets[4];
  int indirect_blks;
  uint blkn;
  int blocks_to_boundary;
  ext2_fsblk_t goal;
  int count;
  unsigned long maxblocks;
  int err;

  depth = ext2_block_to_path(ip, bn, offsets, &blocks_to_boundary);

  if (depth == 0)
    panic("Wrong depth value");

  partial = ext2_get_branch(ip, depth, offsets, chain);

  if (!partial) {
    goto got_it;
  }

  maxblocks = sb[ip->dev].blocksize >> EXT2_BLOCK_SIZE_BITS(&sb[ip->dev]);

  // The requested block is not allocated yet
  goal = ext2_find_goal(ip, bn, partial);

  /* the number of blocks need to allocate for [d,t]indirect blocks */
  indirect_blks = (chain + depth) - partial - 1;

  indirect_blks = (chain + depth) - partial - 1;
  /*
   * Next look up the indirect map to count the totoal number of
   * direct blocks to allocate for this branch.
   */
  count = ext2_blks_to_allocate(partial, indirect_blks,
      maxblocks, blocks_to_boundary);

  err = ext2_alloc_branch(ip, indirect_blks, &count, goal,
      offsets + (partial - chain), partial);

  if (err < 0)
    panic("error␣on␣ext2_alloc_branch");

got_it:
  blkn = chain[depth-1].key;
  ext2_update_branch(ip, bn, chain);

  /* Clean up and exit */
  partial = chain + depth - 1;  /* the whole chain */
/* cleanup: */
  while (partial > chain) {
    brelse(partial->bh);
    partial--;
  }

  return blkn;
}

void
ext2_ilock(struct inode *ip)
{
  struct buf *bp;
  struct ext2_inode *raw_inode;
  struct ext2_inode_info *ei;

  ei = ip->i_private;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquire(&icache.lock);
  while (ip->flags & I_BUSY)
    sleep(ip, &icache.lock);
  ip->flags |= I_BUSY;
  release(&icache.lock);

  if (!(ip->flags & I_VALID)) {
    raw_inode = ext2_get_inode(&sb[ip->dev], ip->inum, &bp);
    // Translate the inode type to xv6 type
    if (S_ISDIR(raw_inode->i_mode)) {
      ip->type = T_DIR;
    } else if (S_ISREG(raw_inode->i_mode)) {
      ip->type = T_FILE;
    } else if (S_ISCHR(raw_inode->i_mode) || S_ISBLK(raw_inode->i_mode)) {
      ip->type = T_DEV;
    } else {
      panic("ext2:␣invalid␣file␣mode");
    }
    ip->nlink = raw_inode->i_links_count;
    ip->size = raw_inode->i_size;
    memmove(&ei->i_ei, raw_inode, sizeof(ei->i_ei));

    ext2_ops.brelse(bp);
    ip->flags |= I_VALID;
    if (ip->type == 0)
      panic("ext2␣ilock:␣no␣type");
  }
}

int
ext2_writei(struct inode *ip, char *src, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if (ip->type == T_DEV) {
    if (ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].write)
      return -1;
    return devsw[ip->major].write(ip, src, n);
  }

  if (off > ip->size || off + n < off)
    return -1;

  // TODO: Verify the max file size

  for (tot = 0; tot < n; tot += m, off += m, src += m) {
    bp = ext2_ops.bread(ip->dev, ext2_iops.bmap(ip, off / sb[ip->dev].blocksize));
    m = min(n - tot, sb[ip->dev].blocksize - off % sb[ip->dev].blocksize);
    memmove(bp->data + off % sb[ip->dev].blocksize, src, m);
    ext2_ops.bwrite(bp);
    ext2_ops.brelse(bp);
  }

  if(n > 0 && off > ip->size){
    ip->size = off;
    ext2_iops.iupdate(ip);
  }

  return n;
}

/*
 * Return the offset into page ‘page_nr’ of the last valid
 * byte in that page, plus one.
 */
static unsigned
ext2_last_byte(struct inode *inode, unsigned long page_nr)
{
  unsigned last_byte = inode->size;
  last_byte -= page_nr * sb[inode->dev].blocksize;
  if (last_byte > sb[inode->dev].blocksize)
    last_byte = sb[inode->dev].blocksize;
  return last_byte;
}


```

```c
int
ext2_dirlink(struct inode *dp, char *name, uint inum, uint type)
{
  int namelen = strlen(name);
  struct buf *bh;
  unsigned chunk_size = sb[dp->dev].blocksize;
  unsigned reclen = EXT2_DIR_REC_LEN(namelen);
  unsigned short rec_len, name_len;
  char *dir_end;
  struct ext2_dir_entry_2 *de;
  int n;
  int numblocks = (dp->size + chunk_size - 1) / chunk_size;
  char *kaddr;

  if (ext2_iops.dirlookup(dp, name, 0) != 0) {
    return -1;
  }

  for (n = 0; n <= numblocks; n++) {
    bh = ext2_ops.bread(dp->dev, ext2_iops.bmap(dp, n));
    kaddr = (char *) bh->data;
    de = (struct ext2_dir_entry_2 *) kaddr;
    dir_end = kaddr + ext2_last_byte(dp, n);
    kaddr += chunk_size - reclen;

    while ((char *)de <= kaddr) {
      if ((char *)de == dir_end) {
        /* We hit i_size */
        name_len = 0;
        rec_len = chunk_size;
        de->rec_len = chunk_size;
        de->inode = 0;
        goto got_it;
      }

      if (de->rec_len == 0) {
        return -1;
      }

      name_len = EXT2_DIR_REC_LEN(de->name_len);
      rec_len = de->rec_len;
      if (!de->inode && rec_len >= reclen)
              goto got_it;
      if (rec_len >= name_len + reclen)
              goto got_it;
      de = (struct ext2_dir_entry_2 *) ((char *) de + rec_len);
    }

    ext2_ops.brelse(bh);
  }

  return -1;

got_it:
  if (de->inode) {
    struct ext2_dir_entry_2 *de1 = (struct ext2_dir_entry_2 *) ((char *) de + name_len);
    de1->rec_len = rec_len - name_len;
    de->rec_len = name_len;
    de = de1;
  }
  de->name_len = namelen;
  strncpy(de->name, name, namelen);
  de->inode = inum;

  // Translate the xv6 to inode type type
  if (type == T_DIR) {
    de->file_type = EXT2_FT_DIR;
  } else if (type == T_FILE) {
    de->file_type = EXT2_FT_REG_FILE;
  } else {
    // We did not treat char and block devices with difference.
    panic("ext2:␣invalid␣inode␣mode");
  }

  ext2_ops.bwrite(bh);
  ext2_ops.brelse(bh);

  if ((n + 1) * chunk_size > dp->size) {
    dp->size += rec_len;
    ext2_iops.iupdate(dp);
  }

  return 0;
}

int
ext2_isdirempty(struct inode *dp)
{
  struct buf *bh;
  unsigned long i;
  char *kaddr;
  struct ext2_dir_entry_2 *de;
  int chunk_size = sb[dp->dev].blocksize;
  int numblocks = (dp->size + chunk_size - 1) / chunk_size;

  for (i = 0; i < numblocks; i++) {
    bh = ext2_ops.bread(dp->dev, ext2_iops.bmap(dp, i));

    if (!bh) {
      panic("ext2_isemptydir␣error");
    }

    kaddr = (char *)bh->data;
    de = (struct ext2_dir_entry_2 *)kaddr;
    kaddr += ext2_last_byte(dp, i) - EXT2_DIR_REC_LEN(1);

    while ((char *)de <= kaddr) {
      if (de->rec_len == 0) {
        goto not_empty;
      }
      if (de->inode != 0) {
        /* check for . and .. */
        if (de->name[0] != '.')
          goto not_empty;
        if (de->name_len > 2)
          goto not_empty;
        if (de->name_len < 2) {
          if (de->inode != dp->inum)
            goto not_empty;
        } else if (de->name[1] != '.')
```

```
          goto not_empty;
      }
      de = (struct ext2_dir_entry_2 *)((char *)de + de->rec_len);
    }
    ext2_ops.brelse(bh);
  }
  return 1;

not_empty:
  ext2_ops.brelse(bh);
  return 0;
}

int
ext2_unlink(struct inode *dp, uint off)
{
  struct buf *bh;
  uint bn, offset;
  struct ext2_dir_entry_2 *dir;
  int chunk_size;

  chunk_size = sb[dp->dev].blocksize;
  bn = off / sb[dp->dev].blocksize;
  offset = off % sb[dp->dev].blocksize;
  bh = ext2_ops.bread(dp->dev, ext2_iops.bmap(dp, bn));

  dir = (struct ext2_dir_entry_2 *)(bh->data + offset);
  char *kaddr = (char *)bh->data;

  unsigned from = ((char*)dir - kaddr) & ~(chunk_size - 1);
  unsigned to = ((char *)dir - kaddr) + dir->rec_len;

  struct ext2_dir_entry_2 *pde = 0;
  struct ext2_dir_entry_2 *de = (struct ext2_dir_entry_2 *) (kaddr + from);

  while ((char*)de < (char*)dir) {
    if (de->rec_len == 0)
      panic("ext2_unlink␣invalid␣dir␣content");
    pde = de;
    de = (struct ext2_dir_entry_2 *)((char *)de + de->rec_len);
  }

  if (pde) {
    from = (char*)pde - (char *)bh->data;
    pde->rec_len = to - from;
  }

  dir->inode = 0;

  ext2_ops.bwrite(bh);
  ext2_ops.brelse(bh);

  return 0;
}

int
ext2_namecmp(const char *s, const char *t)
{
  unsigned short slen = strlen(s), tlen = strlen(t);
  unsigned short size = slen;

  if (slen != tlen)
    return -1;

  if (tlen > slen)
    size = tlen;

  return strncmp(s, t, size);
}

static struct ext2_inode *
ext2_get_inode(struct superblock *sb, uint ino, struct buf **bh)
{
  struct buf * bp;
  unsigned long block_group;
  unsigned long block;
  unsigned long offset;
  struct ext2_group_desc *gdp;
  struct ext2_inode *raw_inode;

  if ((ino != EXT2_ROOT_INO && ino < EXT2_FIRST_INO(sb)) ||
       ino > EXT2_SB(sb)->s_es->s_inodes_count)
    panic("Ext2␣invalid␣inode␣number");

  block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
  gdp = ext2_get_group_desc(sb, block_group, 0);
  if (!gdp)
    panic("Invalid␣group␣descriptor␣at␣ext2_get_inode");

  /*
   * Figure out the offset within the block group inode table
   */
  offset = ((ino - 1) % EXT2_INODES_PER_GROUP(sb)) * EXT2_INODE_SIZE(sb);
  block = gdp->bg_inode_table +
    (offset >> EXT2_BLOCK_SIZE_BITS(sb));

  if (!(bp = ext2_ops.bread(sb->minor, block)))
    panic("Error␣on␣read␣the␣␣block␣inode");

  offset &= (EXT2_BLOCK_SIZE(sb) - 1);
  raw_inode = (struct ext2_inode *)(bp->data + offset);
  if (bh)
    *bh = bp;

  return raw_inode;
}

/**
 * Its is called because the icache lookup failed
 */
int
ext2_fill_inode(struct inode *ip) {
  struct ext2_inode_info *ei;
  struct ext2_inode *raw_inode;
  struct buf *bh;

  ei = alloc_ext2_inode_info();

  if (ei == 0)
    panic("No␣memory␣to␣alloc␣ext2_inode");

  raw_inode = ext2_get_inode(&sb[ip->dev], ip->inum, &bh);
  memmove(&ei->i_ei, raw_inode, sizeof(ei->i_ei));
  ip->i_private = ei;

  ext2_ops.brelse(bh);

  // Translate the inode type to xv6 type
  if (S_ISDIR(ei->i_ei.i_mode)) {
    ip->type = T_DIR;
  } else if (S_ISREG(ei->i_ei.i_mode)) {
    ip->type = T_FILE;
  } else if (S_ISCHR(ei->i_ei.i_mode) || S_ISBLK(ei->i_ei.i_mode)) {
    ip->type = T_DEV;
  } else {
    panic("ext2: invalid file mode");
  }

  ip->nlink = ei->i_ei.i_links_count;
  ip->size  = ei->i_ei.i_size;
  return 1;
}

struct inode*
ext2_iget(uint dev, uint inum)
{
  return iget(dev, inum, &ext2_fill_inode);
}
```