The Brainix Manual

The Brainix Team

Copyleft © Draft date June 22, 2007

Contents

Part I

Introductory matters

Chapter 1

Brainix Manual To-Do List

So, this manual, like the Brainix Operating System itself, is in need of constant revision. The obvious things to-do are as follow:

- Document the kernel itself.
- Explain how the message system works.
- Explain how the block cache concept works.
- Explain the scheduler.
- Explain the idle process[1]
- Explain the library.
- Have pedagogical material(?) so the manual is more like a self-contained textbook than a real manual (see [2] for example of this).

[1]Perhaps we should get the idle process *working* before explaining it...

Part II

Understanding the Brainix Operating System

Chapter 2

On the Brainix File System

> “ ‘Where shall I begin, please your Majesty?’
>
> ‘Begin at the beginning,’ the King said gravely, ‘and go on till you come to the end: then stop.’ ”[1]

Note that this is released under the GNU Free Documentation License version 1.2. See the file fdl.tex for details of the license. This, like the rest of the Brainix Project, is a work in progress.

2.1 Introduction: How Servers Work (A Quick Gloss over it)

The structure for the file system is simple, it is structured like all servers for the micro-kernel:

```
 /* main.c */
int main(void) {
     init(); //This starts up the server and initializes values
             //registers it with the kernel and file system
             //if necessary, etc.
     msg* m; //this is the message buffer

     //You can tell I didn't program this otherwise
     //SHUT_DOWN would be GO_TO_HELL
     while((&m = msg_receive(ANYONE))->op != SHUT_DOWN)
     {
          switch(m->op) {
               case OP_ONE: /* ... */ break;
               /* other op cases supported by the server */
               default: panic("server", "unrecognized message!");
          }
          //The following deals with the reply
          switch(m->op)
          {
```

[1] *Alice's Adventures in Wonderland* by Lewis Carroll, chapter 12 “Alice's Evidence”

```
            case OP_ONE: /* ... */ break;
            /* other replies that require modifications */
            default: msg_reply(m);
        }
    }
    deinit(); //this is called to de-initialize the server
              //to prepare for shut down
    shut_down(); //I would've named it "buy_the_farm()"
                 //or "go_to_hell()"
    return 0;
}
```

Code fragment 1: typical_server.pseudo_c

With most servers, this is the entirety of the `main.c` file. The actual implementation of the methods (i.e. "the dirty work is carried out through") auxiliary files.

The "op" field of the message refers to the operation; which is a sort of parallel to the monolithic kernel system call. The system call is merely handled in user space.

2.2 How File Systems Traditionally Work

There are probably a number of introductory texts and tutorials on Unix-like file systems. I will mention a few worthy of note [1] [2] [4] [3]. I will **attempt** to briefly explain how the Unix file system works, and explain its implementation in operating systems such as Linux and maybe FreeBSD.

File systems deal with long term information storage. There are three essential requirements for long-term information storage that Tanenbaum and Woodhull recognize [2]:

1. It must be possible to store a very large amount of information.

2. The information must survive the termination of the process using it.

3. Multiple processes must be able to access the information concurrently.

With the exception of the GNU-Hurd solution to these problems, the answer is usually to store information on hard disks in units called **files**. The management of these units is done by a program called the **file system.** (What's so interesting and exciting about Unix and Unix-like operating systems is that it's object oriented: everything "is-a" file!)

Some few notes on the geometry of the structure of hard disks. There are sectors, which consist of 512 bytes. There are blocks, which consist of 2^n sectors (where n is usually 3, but varies between 1 and 5). That is a block is 1024 to 16384 bytes. Typically it is 4096 bytes per block.

2.2.1 The I-Node

The file in Unix[2] is a represented by something called an **inode (index-node)**. This lists the attributes and disk addresses of the file's blocks. The skelix code[3] shall be used (with permission of course) as an example of the simplest inode:

```
01 /* Skelix by Xiaoming Mo (xiaoming.mo@skelix.org)
02  * Licence: GPLv2 */
03 #ifndef FS_H
04 #define FS_H
05
06 #define FT_NML    1
07 #define FT_DIR    2
08
09 struct INODE {
10     unsigned int i_mode;        /* file mode */
11     unsigned int i_size;        /* size in bytes */
12     unsigned int i_block[8];
13 };
```

Code fragment 2: /skelix07/include/fs.h

Note that the different types of inodes there are is defined in lines 06 and 07. The permissions and type of the inode is on line 10. The actual addresses to the blocks that hold the data for the file are stored in the array on line 12. At first you look and think "Huh, only 8 blocks per file? That's only, what, 32768 bytes?!" Since it is incredibly unlikely that all the information you'd ever need could be held in 32 kilobytes, the last two addresses refers to *indirect* addresses. That is the seventh address refers to a sector that contains (512 bytes per sector)(1 address per 4 bytes) = 128 addresses. The seventh entry is called a **indirect block** (although because Skelix is so small, it's an indirect sector). The last entry refers to an indirect block, for this reason it is called a **double indirect block**. The indirect block holds 128 addresses, each address refers to a 512 byte sector (in other operating systems they refer to blocks), so each indirect block refers to $128 \times 512 = 65536$ bytes or 64 kilobytes. The last double indirect block contains 128 single indirect blocks, or $128 \times 64 = 8192$ kilobytes or 8 Megabytes.

In bigger operating systems, there are triple indirect blocks, which if we implemented it in skelix we would get $128 \times 8192 = 1048576$ kilobytes or 1024 megabytes or 1 gigabyte. "Surely there must be quadruple indirect blocks, as I have a file that's several gigabytes on my computer!" Well, the way it is implemented on Linux is that rather than refer to sectors, there are groups of sectors called **block groups**. Instead of accessing *only* 512 byte atoms, we are accessing **4 kilobyte atoms!** Indeed, if I am not mistaken, the Minix 3 file system refers to blocks instead of sectors too.

[2] Out of sheer laziness, "Unix" should be read as "Unix and Unix-like operating systems".
[3] Specifically from here `http://skelix.org/download/07.rar`

2.2.2 The Directory

So what about the directory? Well, in Unix file systems, the general idea is to have a file that contains **directory entries**. Directory entries basically hold at least two things: the file name, and the inode number of the entry. There are other things that are desirable like the name length of the entry, the type of file the entry is, or the offset to be added to the starting address of the directory entry to get the starting address of the next directory entry (the "rectangular length"). Consider the implementation in Skelix:

```
extern struct INODE iroot;

#define MAX_NAME_LEN 11

struct DIR_ENTRY {
     char de_name[MAX_NAME_LEN];
     int de_inode;
};
```

Code fragment 3: /skelix07/include/fs.h

The directory entry is, like the skelix inode, extremely simplistic. It consists of the address to the entry, and the entry's name. Suppose one had the following directory:

inode number	name
1	.
1	..
4	bin
7	dev

One wants to run a program, so one looks up the program `/bin/pwd`. The look-up process then goes to the directory and looks up `/bin/`, it sees the inode number is 4, so the look up process goes to inode 4. It finds:

(I-Node 4 is for /bin/)
Mode
Size
132
...

I-node 4 says that `/bin/` is in block 132. It goes to block 132:

6	.
1	..
19	bash
30	gcc
51	man
26	ls
45	pwd

The look up process goes to the last entry and finds `pwd` - the program we're looking for! The look up process goes to block 45 and finds the inode that refers to the blocks necessary to execute the file. That's how the directory system works in Unix file systems.

Every directory has two directory entries when they are made: 1) `.` which refers to "this" directory, 2) `..` which refers to the parent of "this" directory. In this sense, the directories are a sort of doubly linked lists.

2.3 The File System Details

> "The rabbit-hole went straight on like a tunnel for some time, and then dipped suddenly down, so suddenly that Alice had not a moment to think about stopping herself before she found herself falling down a very deep well."[4]

So if you actually go and look at the file system directory, there are a number of ops that are implemented. Some of them are obvious, like `read()`, `write()`, etc. Others are not really intuitively clear why they're there, like `execve()`. The reason for this is because Brainix attempts to be POSIX-Compliant, and POSIX really wasn't made with Microkernels in mind. So we're stuck having an odd design like this; but the advantage is that we can eventually use a package manager like Portage[5]. The advantages really outweigh the cost of odd design.

So this section will inspect the various operations, and follow the code "down the rabbit hole". Yes we shall inspect the nitty-gritty details and analyze as much as possible. That is my duty as the file system hacker to explain as much as possible, using code snippets where appropriate. So we begin with the initialization of the file system.

2.4 File System Initialization

Looking in the file `/brainix/src/fs/main.c` one finds:

```
void fs_main(void)
{
	/* Initialize the file system. */
	block_init(); /* Initialize the block cache. */
	inode_init(); /* Initialize the inode table. */
	super_init(); /* Initialize the superblock table. */
	dev_init();   /* Initialize the device driver PID table. */
	descr_init(); /* Init the file ptr and proc-specific info tables. */
```

Code fragment 4: /brainix/src/fs/main.c

This is the initialization code that we are interested in. Let's analyze it line by line. First there is a call to the function `block_init();`. So let us inspect this function's code.

[4] *Alice's Adventures in Wonderland* by Lewis Carroll, chapter 1 "Down the Rabbit-Hole"

[5] For those that do not know, Portage is the package manager for the Gentoo distribution of Linux. As far as I know it has been ported to FreeBSD, Open-BSD, Net-BSD, Darwin, and other operating systems because Portage is distributed via its source code. It works by downloading and compiling source code auto-magically and optimizing it as much as possible with the GCC.

2.4.1 block_init()

block_t

There is the matter of the data structure that is involved here extensively that we ought to investigate first: `block_t`.

```
/* A cached block is a copy in RAM of a block on a device: */
typedef struct block
{
     /* The following field resides on the device: */
     char data[BLOCK_SIZE]; /* Block data. */

     /* The following fields do not reside on the device: */
     dev_t dev;             /* Device the block is on.            */
     blkcnt_t blk;          /* Block number on its device.        */
     unsigned char count;   /* Number of times the block is used. */
     bool dirty;            /* Block changed since read.          */
     struct block *prev;    /* Previous block in the list.        */
     struct block *next;    /* Next block in the list.            */
} block_t;
```

Code fragment 5: /brainix/inc/fs/block.h

dev_t

This is all rather straight forward. The `dev_t` field tells us what device we are dealing with, rather what device the file system is dealing with. To be more precise about what exactly `dev_t` is we look to the code:

```
/* Used for device IDs: */
#ifndef _DEV_T
#define _DEV_T
typedef unsigned long dev_t;
#endif
```

Code fragment 6: /brainix/inc/lib/sys/type.h

blkcnt_t

which is pretty self-explanatory that `dev_t` is little more than an unsigned long. The `blkcnt_t blk` field gives more precision with what we are dealing with, which is a rather odd field because I don't know what the `blkcnt_t` type is off hand so I doubt that you would either. Let us shift our attention to this type!

```
/* Used for file block counts: */
#ifndef _BLKCNT_T
#define _BLKCNT_T
typedef long blkcnt_t;
#endif
```

Code fragment 7: /brainix/inc/lib/sys/type.h

So this is a rather straight forward type that needs no explanation it seems. We can continue our analysis of the `block_t` struct. The `unsigned char count;` is little more than a simple counter it seems, and the `bool dirty;` tells us whether the block has changed since last read or not. The last two entries tells us this `block_t` data structure is a doubly linked list. This is common, the use of doubly linked lists that is, because it is common to lose things at such a low level.

block_init()

Now we may proceed to analyze the `block_init()` function defined in the `block.c` file:

```
void block_init(void)
{
```

```
34
35 /* Initialize the block cache. */
36
37      block_t *block_ptr;
38
39      /* Initialize each block in the cache. */
40      for (block_ptr = &block[0]; block_ptr < &block[NUM_BLOCKS]; block_ptr++)
41      {
42           block_ptr->dev = NO_DEV;
43           block_ptr->blk = 0;
44           block_ptr->count = 0;
45           block_ptr->dirty = false;
46           block_ptr->prev = block_ptr - 1;
47           block_ptr->next = block_ptr + 1;
48      }
49
50      /* Make the cache linked list circular. */
51      block[0].prev = &block[NUM_BLOCKS - 1];
52      block[NUM_BLOCKS - 1].next = &block[0];
53
54      /* Initialize the least recently used position in the cache. */
55      lru = &block[0];
56 }
```

Code fragment 8: /brainix/src/fs/block.c

Line 37 simply initializes a block pointer that is used to initialize the blocks. Lines 40 to 48 (the for-loop) uniformly sets all the blocks to be identical with the exact same fields. The fields are self explanatory; the device number is set to no device (line 42), the number of times the block has been used is set to zero (line 43), the block has not changed since it's last been read (line 44), the previous block and next block are rather elementarily defined.

At first one would think looking up until line 47 that there would have to be a negative block, and that block would require another, and so on *ad infinitum*. But lines 50 to 52 make the block a circularly doubly linked list. Line 51 makes the zeroeth block's previous block `prev` refer to the last block, and line 52 makes the last block's `next` field refers to the zeroeth block's address.

What's the significance of line 55? Well, I don't know. It does not seem to relevant at the moment, though undoubtedly we shall have to come back to it in the future.

2.4.2 inode_init()

Just as we had the `block_init()` we have a `inode_init()`. If you are new to this whole Unix-like file system idea, it is highly recommended that you read [5] [6] [7] [8] [9] [10]. Perhaps in a future version of this documentation it will be explained in further detail. The original motivation I suspect (yes, this is a baseless conjecture I made up from my own observations that is probably not true at all) was to have something similar to a hybrid of Linux and Minix 3, and this is somewhat reflected by the choice of attempting to support the ext2 file system (the file system from the earlier Linux distributions). The inode data structure is identical to its description in the third edition of *Understanding the Linux Kernel*. However I am making this an independent, stand-alone type of reference...so that means I am going to inspect the data structure, line by line.

```
42 /* An inode represents an object in the file system: */
43 typedef struct
44 {
45      /* The following fields reside on the device: */
46      unsigned short i_mode;       /* File format / access rights.       */
47      unsigned short i_uid;        /* User owning file.                  */
48      unsigned long i_size;        /* File size in bytes.                */
49      unsigned long i_atime;       /* Access time.                       */
50      unsigned long i_ctime;       /* Creation time.                     */
51      unsigned long i_mtime;       /* Modification time.                 */
52      unsigned long i_dtime;       /* Deletion time (0 if file exists).  */
53      unsigned short i_gid;        /* Group owning file.                 */
54      unsigned short i_links_count; /* Links count.                      */
55      unsigned long i_blocks;      /* 512-byte blocks reserved for file. */
56      unsigned long i_flags;       /* How to treat file.                 */
57      unsigned long i_osd1;        /* OS dependent value.                */
58      unsigned long i_block[15];   /* File data blocks.                  */
59      unsigned long i_generation;  /* File version (used by NFS).        */
60      unsigned long i_file_acl;    /* File ACL.                          */
61      unsigned long i_dir_acl;     /* Directory ACL.                     */
62      unsigned long i_faddr;       /* Fragment address.                  */
63      unsigned long i_osd2[3];     /* OS dependent structure.            */
64
65      /* The following fields do not reside on the device: */
66      dev_t dev;                   /* Device the inode is on.            */
67      ino_t ino;                   /* Inode number on its device.        */
68      unsigned char count;         /* Number of times the inode is used. */
69      bool mounted;                /* Inode is mounted on.               */
70      bool dirty;                  /* Inode changed since read.          */
71 } inode_t;
```

Code fragment 9: /brainix/inc/fs/inode.h

A lot of this code is seemingly unused. All that really matters is that the `inode_t` data type is a wrapper for the addresses (line 58), with some constraints for permissions and so forth (lines 46 to 57), and some device specific fields (lines 66 to 70). This data structure is nearly identical to the ext2 file system's `inode` struct. As stated previously, the motivation was to incorporate the ext2 file system into Brainix. This proved too difficult since the ext2 file system is intimately related to the Linux virtual file system. It seems that the most appropriate description for the Brainix file system is a fork of the ext2 one.

inode_init()

Now on to the `inode_init()` code itself:

```
void inode_init(void)
{

/* Initialize the inode table. */

     inode_t *inode_ptr;

     /* Initialize each slot in the table. */
     for (inode_ptr = &inode[0]; inode_ptr < &inode[NUM_INODES]; inode_ptr++)
     {
          inode_ptr->dev = NO_DEV;
          inode_ptr->ino = 0;
          inode_ptr->count = 0;
          inode_ptr->mounted = false;
```

```
46          inode_ptr->dirty = false;
47      }
48 }
```

Code fragment 10: /brainix/src/fs/inode.c

Line 37 tells us there is a dummy inode pointer that is used later on, more specifically it is used in lines 40 to 47 when the inode table is initialized. The for-loop, as stated, initializes the inode-table. Line 42 sets the device that the inode is on to `NO_DEV`, line 43 sets the inode number to zero, the next line (line 44) sets the number of times the inode is used to zero, line 45 sets the boolean checking whether the inode is mounted or not to false (the inode is initialized to be not mounted), and line 46 tells us that the inode has not changed since we last dealt with it.

Now that the inode table has been initialized, we now look to the initialization of the super block.

2.4.3 super_init()

To inspect the inner workings of the `super_init()` method we need to first investigate the `super` struct representing the super block.

```
/* The superblock describes the configuration of the file system: */
typedef struct
{
    /* The following fields reside on the device: */
    unsigned long s_inodes_count;      /* Total number of inodes.         */
    unsigned long s_blocks_count;      /* Total number of blocks.         */
    unsigned long s_r_blocks_count;    /* Number of reserved blocks.      */
    unsigned long s_free_blocks_count; /* Number of free blocks.          */
    unsigned long s_free_inodes_count; /* Number of free inodes.          */
    unsigned long s_first_data_block;  /* Block containing superblock.    */
    unsigned long s_log_block_size;    /* Used to compute block size.     */
    long s_log_frag_size;              /* Used to compute fragment size.  */
    unsigned long s_blocks_per_group;  /* Blocks per group.               */
    unsigned long s_frags_per_group;   /* Fragments per group.            */
    unsigned long s_inodes_per_group;  /* Inodes per group.               */
    unsigned long s_mtime;             /* Time of last mount.             */
    unsigned long s_wtime;             /* Time of last write.             */
    unsigned short s_mnt_count;        /* Mounts since last fsck.         */
    unsigned short s_max_mnt_count;    /* Mounts permitted between fscks. */
    unsigned short s_magic;            /* Identifies as ext2.             */
    unsigned short s_state;            /* Cleanly unmounted?              */
    unsigned short s_errors;           /* What to do on error.            */
    unsigned short s_minor_rev_level;  /* Minor revision level.           */
    unsigned long s_lastcheck;         /* Time of last fsck.              */
    unsigned long s_checkinterval;     /* Time permitted between fscks.   */
    unsigned long s_creator_os;        /* OS that created file system.    */
    unsigned long s_rev_level;         /* Revision level.                 */
    unsigned short s_def_resuid;       /* UID for reserved blocks.        */
    unsigned short s_def_resgid;       /* GID for reserved blocks.        */
    unsigned long s_first_ino;         /* First usable inode.             */
    unsigned short s_inode_size;       /* Size of inode struct.           */
    unsigned short s_block_group_nr;   /* Block group of this superblock. */
    unsigned long s_feature_compat;    /* Compatible features.            */
```

```
68      unsigned long s_feature_incompat;  /* Incompatible features.          */
69      unsigned long s_feature_ro_compat; /* Read-only features.             */
70      char s_uuid[16];                   /* Volume ID.                      */
71      char s_volume_name[16];            /* Volume name.                    */
72      char s_last_mounted[64];           /* Path where last mounted.        */
73      unsigned long s_algo_bitmap;       /* Compression methods.            */
74
75      /* The following fields do not reside on the device: */
76      dev_t dev;                         /* Device containing file system. */
77      blksize_t block_size;              /* Block size.                    */
78      unsigned long frag_size;           /* Fragment size.                 */
79      inode_t *mount_point_inode_ptr;    /* Inode mounted on.              */
80      inode_t *root_dir_inode_ptr;       /* Inode of root directory.       */
81      bool dirty;                        /* Superblock changed since read. */
82 } super_t;
```

Code fragment 11: /brainix/inc/fs/super.h

This is the super block, and - as previously iterated a number of times - this is from the ext2 file system. The superblock should have the magic number `s_magic` which tells us this is indeed the ext2 file system. Line 61 tells us the revision level which allows the mounting code to determine whether or not this file system supports features available to particular revisions. When a new file is created, the values of the `s_free_inodes_count` field in the Ext2 superblock and of the `bg_free_inodes_count` field in the proper group descriptor must be decremented. If the kernel appends some data to an existing file so that the number of data blocks allocated for it increases, the values of the `s_free_blocks_count` field in the Ext2 superblock and of the `bg_free_blocks_count` field in the group descriptor must be modified. Even just rewriting a portion of an existing file involves an update of the `s_wtime` field of the Ext2 superblock. For a more in-depth analysis of the ext2 file system's `super_block` data structure which was forked for the Brainix file system, see Chapter 18 [3] or [7] [11] [12].

super_init()

```
32 void super_init(void)
33{
34
35 /* Initialize the superblock table. */
36
37      super_t *super_ptr;
38
39      /* Initialize each slot in the table. */
40      for (super_ptr = &super[0]; super_ptr < &super[NUM_SUPERS]; super_ptr++)
41      {
42           super_ptr->dev = NO_DEV;
43           super_ptr->block_size = 0;
44           super_ptr->frag_size = 0;
45           super_ptr->mount_point_inode_ptr = NULL;
46           super_ptr->root_dir_inode_ptr = NULL;
47           super_ptr->dirty = false;
48      }
49 }
```

Code fragment 12: /brainix/src/fs/super.c

As previously stated, the brainix file system is perhaps more properly thought of as a fork (rather than an implementation) of the ext2 file system. In the ext2 file system, each block group has a super block (as a sort of back up), and this feature has been inherited in the brainix file system. This `init()` method is pretty much identical to the other ones. There is a pointer struct (line 37) that's used in a for-loop to set all the super blocks to be the same (lines 40 to 48).

More specifically, in more detail, line 42 sets each super block's device to `NO_DEV`. The block size for the super block is initialized to be zero as well, with no fragments either (lines 43 and 44). The inode holding the mount point information is set to be `NULL` as is the root directory inode pointer. Since we just initialized the super blocks, they haven't changed since we last used them, so we tell that to the super blocks with line 47.

2.4.4 dev_init()

The pid_t data structure

We should first inspect the vital data structure relevant to discussion here: `pid_t` which is defined in /brainix/inc/lib/unistd.h:

```
#ifndef _PID_T
#define _PID_T
typedef long pid_t;
#endif
```

Code fragment 13: /brainix/inc/lib/unistd.h

That's pretty much the only new data structure (or type, rather) that's relevant for discussion here.

Back to the dev_init() method

The next function called in the `init()` section of the file system server is the `dev_init()`. This is defined in the `/brainix/src/fs/device.c` file:

dev_init()

```
55 void dev_init(void)
56 {
57
58 /* Initialize the device driver PID table. */
59
60      unsigned char maj;
61
62      for (maj = 0; maj < NUM_DRIVERS; maj++)
63           driver_pid[BLOCK][maj] =
64           driver_pid[CHAR][maj] = NO_PID;
65 }
```

Code fragment 14: /brainix/src/fs/device.c

This is the `dev_init()` code, that basically initializes the device driver part of the PID[6] table. At first looking at the for-loop, one says "This won't work!" But upon further inspection, the line 63 doesn't have a semicolon, so the compiler continues to the next line (line 64). It sets the `driver_pid[BLOCK][maj]` to be `NO_PID`. It does this for every major device (more precisely, for the number of drivers `NUM_DRIVERS`). Note that the first index of the matrix that represents the device driver PID table is capable of having values 0 and 1, represented by `BLOCK` and `CHAR` respectively.

[6] "PID" stands for "Process identification".

2.4.5 descr_init()

For this method, there are global variables defined in the headers:

```
/* Global variables: */
file_ptr_t file_ptr[NUM_FILE_PTRS]; /* File pointer table.                   */
fs_proc_t fs_proc[NUM_PROCS];       /* Process-specific information table. */
```

Code fragment 15: /brainix/inc/fs/fildes.h

file_ptr_t

This allows us to introduce the data structures `fs_proc_t` and `file_ptr_t`.

```
35 /* A file pointer is an intermediary between a file descriptor and an inode: */
36 typedef struct
37 {
38      inode_t *inode_ptr;  /* Inode pointer.        */
39      unsigned char count; /* Number of references. */
40      off_t offset;        /* File position.        */
41      int status;          /* File status.          */
42      mode_t mode;         /* File mode.            */
43 } file_ptr_t;
```

Code fragment 16: /brainix/inc/fs/fildes.h

This is self explanatory thanks to the comments. The `file_ptr_t` is an intermediary between a file descriptor and an i-node. It consists of the inode it intermediates for (line 38), the number of references made to the inode in the file descriptor (line 39), the file position's offset (line 40), the status of the file (41), and the mode of the file (42). The other important data structure is:

fs_proc_t

```
/* Process-specific file system information: */
typedef struct
{
     inode_t *root_dir;                /* Root directory.            */
     inode_t *work_dir;                /* Current working directory. */
     mode_t cmask;                     /* File mode creation mask.   */
     file_ptr_t *open_descr[OPEN_MAX]; /* File descriptor table.     */
} fs_proc_t;
```

Code fragment 17: /brainix/inc/fs/fildes.h

Which gives us information about the file system which is process-specific, as the comment suggests. More to the point, the root directory inode, the current directory inode, the "file mode creation mask" which is little more than telling the file what you **DON'T** want ("Setting a mask is the opposite of setting the permissions themselves; when you set a mask, you are telling the computer the permissions you do not want, rather than the permissions you do" [13]), and more importantly the file descriptor table.

This is the last step in the file system initialization. It essentially initializes a few other tables that we are going to use.

descr_init()

```
void descr_init(void)
{

/* Initialize the file pointer table and the process-specific file system
 * information table. */

     int ptr_index;
     pid_t pid;
```

```
    int descr_index;

    /* Initialize the file pointer table. */
    for (ptr_index = 0; ptr_index < NUM_FILE_PTRS; ptr_index++)
    {
        file_ptr[ptr_index].inode_ptr = NULL;
        file_ptr[ptr_index].count = 0;
        file_ptr[ptr_index].offset = 0;
        file_ptr[ptr_index].status = 0;
        file_ptr[ptr_index].mode = 0;
    }

    /* Initialize the process-specific file system information table. */
    for (pid = 0; pid < NUM_PROCS; pid++)
    {
        fs_proc[pid].root_dir = NULL;
        fs_proc[pid].work_dir = NULL;
        fs_proc[pid].cmask = 0;
        for (descr_index = 0; descr_index < OPEN_MAX; descr_index++)
            fs_proc[pid].open_descr[descr_index] = NULL;
    }
}
```

Code fragment 18: /brainix/src/fs/fildes.c

There is nothing new here, only two for-loops rather than one to initialize two (rather than one) tables. But where are these tables defined? They seem to fall from thin air into our laps!

2.5 File System Operations

This section, unlike the previous, is in a seemingly random order. It does not logically follow the structure of the code as it would appear to a new comer to the Brainix kernel. Instead, it inspects the more important operations first, discussing them at length. We shall begin with the most recently inspected method: `REGISTER`.

2.5.1 REGISTER

This method was long thought to be a problem child, until some clever debugging proved it to be little more than a nuisance. It is a function in the `device.c` file:

fs_register()

```
void fs_register(bool block, unsigned char maj, pid_t pid)
{

/* Register a device driver with the file system - map a device's major number
 * to its driver's PID.  If the driver for the device containing the root file
 * system is being registered, mount the root file system and initialize the
 * root and current working directories. */

    dev_t dev;

    /* Register the device driver with the file system. */
```

```
81      driver_pid[block][maj] = pid;
82
83      if (block && maj == ROOT_MAJ)
84      {
85          /* The driver for the device containing the root file system is
86           * being registered. */
87          mount_root();
88          dev = maj_min_to_dev(ROOT_MAJ, ROOT_MIN);
89          fs_proc[FS_PID].root_dir = inode_get(dev, EXT2_ROOT_INO);
90          fs_proc[FS_PID].work_dir = inode_get(dev, EXT2_ROOT_INO);
91      }
92 }
```

Code fragment 19: /brainix/src/fs/device.c

This register method is rather straightforward: it adds the device driver to the device driver PID table, then it checks to see if this is a root device we are mounting. If it is, then it calls some additional functions to mount the root file system on the device (line 87), it creates the `dev_t` from the major and minor numbers of the device, assigns the inodes to the root directory and working directory. We shall investigate each of these components of the function in turn.

mount_root()

First, the `mount_root()` method which unsurprisingly mounts the root file system.

```
void mount_root(void)
{

/* Mount the root file system. */

    super_t *super_ptr;

    /* Open the device. */
    dev_open_close(ROOT_DEV, BLOCK, OPEN);
    if (err_code)
        /* The device could not be opened. */
        panic("mount_root", strerror(err_code));
```

Code fragment 20: /brainix/src/fs/mount.c

dev_open_close()

So let us explore this far and say to ourselves "Aha! So, it calls this function '`dev_open_close()`', let's see what that does exactly!" We look for this method and find it:

```
int dev_open_close(dev_t dev, bool block, bool open)
{

/* If open is true, open a device.  Otherwise, close a device. */

    unsigned char maj, min;
    pid_t pid;
    msg_t *m;
    int ret_val;
```

Code fragment 21: /brainix/src/fs/device.c

So far several variables are initialized as dummy variables, that is "local variables" which are not used permanently. They are used only temporarily, like a counter.

```
156     /* Find the device driver's PID. */
157     dev_to_maj_min(dev, &maj, &min);
158     pid = driver_pid[block][maj];
159     if (pid == NO_PID)
160         return -(err_code = ENXIO);
```

Code fragment 22: /brainix/src/fs/device.c

We have all ready seen the driver PID table before, but line 157 is completely foreign. We have yet to see exactly what `dev_to_maj_min` does. We can easily locate it however: dev_to_maj_min()

```
32 void dev_to_maj_min(dev_t dev, unsigned char *maj, unsigned char *min)
33 {
34
35 /* Extract the minor number and the minor number from a device number. */
36
37     *maj = (dev & 0xFF00) >> 8;
38     *min = (dev & 0x00FF) >> 0;
39 }
```

Code fragment 23: /brainix/src/fs/device.c

Which is pretty self explanatory code. Line 37 uses bitwise operators to set the Major number to be a modification of the last 8 bits of the `dev`, and line 38 uses bitwise operators to set the Minor number to be a modification of the first 8 bits of the `dev` variable. This code is solid and has been tested, it probably shouldn't need to be changed. At any rate, back to the `dev_open_close()` method:

```
162     /* Send a message to the device driver. */
163     m = msg_alloc(pid, open ? SYS_OPEN : SYS_CLOSE);
164     m->args.open_close.min = min;
165     msg_send(m);
```

Code fragment 24: /brainix/src/fs/device.c

The code explains itself quite readily. Line 163 allocates a message, line 164 sets the minor number, and line 165 sends the message to the device.

```
167     /* Await the device driver's reply. */
168     m = msg_receive(pid);
169     ret_val = m->args.open_close.ret_val;
170     msg_free(m);
171     if (ret_val < 0)
172         err_code = -ret_val;
173     return ret_val;
174 }
```

Code fragment 25: /brainix/src/fs/device.c

This code is also self-explanatory. Line 168 waits for the message from the driver, presumably in reply to the message sent from line 165 though this may or may not be the case; line 169 analyzes the message's return value. Now that the return value has been extracted, we can delete the message to free up space (line 170) assuming this wasn't a different message sent by the device driver asking to do some other method, the code - as you can tell - does not check. It does return the return value from the message however (line 173) and catches any possible errors (lines 171-2).

We can assume that the device driver coder knows what he's doing, so we won't investigate the interactions of this message with regards to the device driver. The interested reader can look up the appropriate code in the driver documentation (or supposing that it has yet to be written, which implies the Brainix operating system is still early in development, look at the /brainix/src/driver/floppy.c).

We continue our investigation of the `mount_root()` method, after finding out quite a bit about the `dev_open_close()` method.

```
143      /* Read the superblock. */
144      super_ptr = super_read(ROOT_DEV);
145      if (err_code)
146           /* The superblock could not be read. */
147           panic("mount_root", strerror(err_code));
```

Code fragment 26: /brainix/src/fs/mount.c

The `mount_root()` reads in the super block from the root directory on the device (line 144). If it could not have been read in, there is a kernel panic (line 147).

super_read()

We shall now shift our focus onto the `super_read()` method:

```
75 super_t *super_read(dev_t dev)
76 {
77
78 /* Read a superblock from its block into the superblock table, and return a
79  * pointer to it. */
80
81      super_t *super_ptr;
82      block_t *block_ptr;
```

Code fragment 27: /brainix/src/fs/super.c

Again, as is the style of Brainix, the dummy variables are defined first (lines 81-2) and a brief comment description of the method is given (lines 78-9).

```
84      /* Find a free slot in the table. */
85      if ((super_ptr = super_get(NO_DEV)) == NULL)
86           /* There are no free slots in the table --- too many mounted
87            * file systems. */
88           return NULL;
```

Code fragment 28: /brainix/src/fs/super.c

This segment of code from the `super_read()` calls the `super_get()` method. Indeed it is a bit convoluted, but it's the easiest way to program it. Let us now analyze the `super_get()` method:

super_get()

```
54 super_t *super_get(dev_t dev)
55 {
56
57 /* Search the superblock table for a superblock.  If it is found, return a
58  * pointer to it.  Otherwise, return NULL. */
59
60      super_t *super_ptr;
61
62      /* Search the table for the superblock. */
63      for (super_ptr = &super[0]; super_ptr < &super[NUM_SUPERS]; super_ptr++)
```

```
64          if (super_ptr->dev == dev)
65              /* Found the superblock.  Return a pointer to it. */
66              return super_ptr;
67
68      /* The superblock is not in the table. */
69      return NULL;
70 }
```

Code fragment 29: /brainix/src/fs/super.c

Line 60 initializes the dummy variable, lines 63-66 is a simple, linear for-loop search through the superblock table that is looking for a superblock on the device `dev`. If it is found (line 64), then it returns a pointer to that super block struct (line 66). Supposing that the for loop has run out of places to look on the table, it returns `NULL` indicating that the superblock is not in the table.

Returning our focus to the `super_read()` method:

```
90      /* Copy the superblock from its block into the free slot. */
91      block_ptr = block_get(dev, SUPER_BLOCK);
92      memcpy(super_ptr, block_ptr->data, offsetof(super_t, dev));
```

Code fragment 30: /brainix/src/fs/super.c

Lines 91 and 92 introduce two new functions that we will need to investigate: `block_get()` and `memcpy()`. Essentially, line 91 looks on the device `dev` for the block `SUPER_BLOCK`. Get some caffeine in your system, because the `block_get()` method requires more focus and attention: block_get()

```
block_t *block_get(dev_t dev, blkcnt_t blk)
{

/* Search the cache for a block.  If it is found, return a pointer to it.
 * Otherwise, evict a free block, cache the block, and return a pointer to
 * it. */

     block_t *block_ptr;

     /* Search the cache for the block. */
     for (block_ptr = lru->prev; ; )
          if (block_ptr->dev == dev && block_ptr->blk == blk)
          {
               /* Found the block.  Increment the number of times it is
                * used, mark it recently used, and return a pointer to
                * it. */
               block_ptr->count++;
               recently_used(block_ptr, MOST);
               return block_ptr;
          }
          else if ((block_ptr = block_ptr->prev) == lru->prev)
               /* Oops - we've searched the entire cache already. */
               break;
```

Code fragment 31: /brainix/src/fs/block.c

The file system keeps a block cache. Whenever you change anything, it changes the file system's block cache. `Block_put()` writes these changes to the disk, that's the whole point of `block_put()`. Right now, however, we are interested in looking through this cache for a specific block. It may be a little inelegant by most standards, but we are absolved by virtue of this being an operating system ("breaks" do not exist in the Queen's C!).

The cache is searched through until either the specific block in question is found (lines 176-183) or we've run out of cache (we're baroque, that is out of Monet, by line 187). If we do run out of cache, that means the requested block is not cached. Which means there is more to this method than meets the eye:

```
189     /* The requested block is not cached.  Search the cache for the least
190      * recently used free block. */
191     for (block_ptr = lru; ; )
192          if (block_ptr->count == 0)
193          {
194               /* Found the least recently used free block.  Evict it.
195                * Cache the requested block, mark it recently used, and
196                * return a pointer to it. */
197               block_rw(block_ptr, WRITE);
198               block_ptr->dev = dev;
199               block_ptr->blk = blk;
200               block_ptr->count = 1;
201               block_ptr->dirty = true;
202               block_rw(block_ptr, READ);
203               recently_used(block_ptr, MOST);
204               return block_ptr;
205          }
206          else if ((block_ptr = block_ptr->next) == lru)
207               /* Oops - we've searched the entire cache already. */
208               break;
```

Code fragment 32: /brainix/src/fs/block.c

What happens is that the cache is searched through again (this may be a source of inefficiency to search the cache twice, just an aside) for a block that has a `count` of 0. Upon finding it we write the `block_ptr` to that block location (line 197), and set some new values for our `block_ptr` (lines 198 to 201). We indicate that we have changed the block since it has last been read (that is what the dirty flag indicates...and how long it's been since the block had a bath). This seems intuitively circular to change this only to have it completely ignored by line 202 when `block_rw()` essentially sets the fields of `block_ptr` to whatever the `block_ptr` is. Just as before, there is a method to break out of the for-loop using pragmatic C coding.

block_rw()

Let us now shift our attention to the method `block_rw()`:

```
86 void block_rw(block_t *block_ptr, bool read)
87 {
88
89 /* If read is true, read a block from its device into the cache.  Otherwise,
90  * write a block from the cache to its device. */
91
92      dev_t dev = block_ptr->dev;
93      off_t off = block_ptr->blk * BLOCK_SIZE;
94      void *buf = block_ptr->data;
95      super_t *super_ptr;
```

Code fragment 33: /brainix/src/fs/block.c

Lines 92-95 are the dummy variables that are used throughout the method, as is usual in the Brainix coding style. Note the comment that tells us what exactly this method does.

```
    if (!block_ptr->dirty)
        /* The cached block is already synchronized with the block on
         * its device.  No reason to read or write anything. */
        return;
```

Code fragment 34: /brainix/src/fs/block.c

If the block pointer is dirty, that means that it has changed since last inspected, then `block_ptr->dirty=TRUE`. We are hoping that the `block_ptr` is dirty, that's the entire point of this method. If it's not dirty, we leave the method right here and now.

```
    /* Read the block from its device into the cache, or write the block
     * from the cache to its device. */
    dev_rw(dev, BLOCK, read, off, BLOCK_SIZE, buf);
```

Code fragment 35: /brainix/src/fs/block.c

Unfortunately, we have not yet had the good fortune to investigate the `dev_rw()` method, so let us do so now! dev_rw()

```
179 ssize_t dev_rw(dev_t dev, bool block, bool read, off_t off, size_t size,
180     void *buf)
181 {
182
183 /* If read is true, read from a device.  Otherwise, write to a device. */
184
185     unsigned char maj, min;
186     pid_t pid;
187     msg_t *m;
188     ssize_t ret_val;
189
190     /* Find the device driver's PID. */
191     dev_to_maj_min(dev, &maj, &min);
192     pid = driver_pid[block][maj];
193     if (pid == NO_PID)
194         return -(err_code = ENXIO);
```

Code fragment 36: /brainix/src/fs/device.c

Lines 185-188 initialize the dummy variables that hold the values for this method. The really interesting part begins at line 190, wherein the device driver's PID is found. If the PID is `NO_PID`, then an error is returned (line 194).

```
196     /* Send a message to the device driver. */
197     m = msg_alloc(pid, read ? SYS_READ : SYS_WRITE);
198     m->args.read_write.min = min;
199     m->args.read_write.off = off;
200     m->args.read_write.size = size;
201     m->args.read_write.buf = buf;
202     msg_send(m);
```

Code fragment 37: /brainix/src/fs/device.c

We hope that the driver can read or write for us, so assuming the driver coder did his or her homework, then we have no worries. We simply allocate a message (line 197), give the message the minor number of the device (198), the offset to read/write (199), the size of the buffer (200), and the buffer to read to or write from (201). The message is then sent.

```
204      /* Await the device driver's reply. */
205      m = msg_receive(pid);
206      ret_val = m->args.read_write.ret_val;
207      msg_free(m);
208      if (ret_val < 0)
209           err_code = -ret_val;
210      return ret_val;
211 }
```

Code fragment 38: /brainix/src/fs/device.c

We wait for a reply. We assume, and I can't stress this enough, assume that the message from the process with PID `pid` is in response to the message sent. The return value is extracted from the reply (line 206), and we free up the message (207). We check to see if there is an error, and then we return the `ret_val`. It's pretty simple.

Back to our discussion on `block_rw()`, recall the code we left off at was:

```
101      /* Read the block from its device into the cache, or write the block
102       * from the cache to its device. */
103      dev_rw(dev, BLOCK, read, off, BLOCK_SIZE, buf);
```

Code fragment 39: /brainix/src/fs/block.c

So now we know what exactly the `dev_rw()` method is, we can understand that line 103 is really asking to read the device `dev`, this is indeed a `BLOCK` device that we are reading from rather than a character device, we are indeed `read`-ing from it with an offset of `off`, we are reading exactly 1 `BLOCK_SIZE` into the buffer `buf` by the method we just inspected above.

```
105      /* The cached block is now synchronized with the block on its device. */
106      block_ptr->dirty = false;
107      if (!read)
108      {
109           super_ptr = super_get(block_ptr->dev);
110           super_ptr->s_wtime = do_time(NULL);
111           super_ptr->dirty = true;
112      }
113 }
```

Code fragment 40: /brainix/src/fs/block.c

We have no updated the inspection with the `block_ptr` so we may set its `dirty` flag to be `false` (clean as a whistle).

Now, we go on to investigate if we are writing to the file (that is checking that the boolean `read` is false, if it is that means we are of course writing to the block, and we simply follow out lines 109 to 111; however, we are not really interested in that at the moment so we will not really inspect those lines of code here).

Back on track to our analysis of `block_get()`:

```
     /* There are no free blocks in the cache.  Vomit. */
     panic("block_get", "no free blocks");
     return NULL;
}
```

Code fragment 41: /brainix/src/fs/block.c

Which are the final lines of `block_get()`. It basically calls a kernel panic (line 211) and returns `NULL`, there's nothing elegant needing explanation here.

Back to the `super_read()` method:

```
    block_put(block_ptr, IMPORTANT);

    return super_ptr;
}
```

Code fragment 42: /brainix/src/fs/super.c

We have not seen `block_put()` although we have seen `block_get()`. There is a difference between the two, and now we shall investigate `block_put()`: block_put()

```
218 void block_put(block_t *block_ptr, bool important)
219 {
220
221 /* Decrement the number of times a block is used.  If no one is using it, write
222  * it to its device (if necessary). */
223
224     if (block_ptr == NULL || --block_ptr->count > 0)
225         return;
```

Code fragment 43: /brainix/src/fs/block.c

If the `block_ptr` is null, or if the `block_ptr`'s `count` is greater than 1, then we exit this method. I think this code needs to be modified, as I think line 224 is supposed to be `--block_ptr->count < 0)`. This code works however, so I wouldn't touch it just yet (although don't feel discouraged or intimidated when meddling with code!).

```
226     switch (ROBUST)
227     {
228         case PARANOID:
229             block_rw(block_ptr, WRITE);
230             return;
231         case SANE:
232             if (important)
233                 block_rw(block_ptr, WRITE);
234             return;
235         case SLOPPY:
236             return;
237     }
238 }
```

Code fragment 44: /brainix/src/fs/block.c

Basically, this tells us *when* `block_rw()` should be called. As previously mentioned, the file system has its own block cache, and this method (`block_put()`) essentially writes the changes in this cache. How often it happens depend on the `ROBUST`-ness of the configuration of the Brainix kernel. Basically, `SLOPPY` optimizes performance, `PARANOID` always writes blocks when the cache changes slightly, and `SANE` is a center between the two where the cache is written if and only if it is `IMPORTANT`.

Back to the `mount_root()` method which invoked this long aside:

```
    /* Perform the mount.  Fill in the superblock's fields. */
    super_ptr->s_mtime = do_time(NULL);
```

```
151      super_ptr->s_mnt_count++;
152      super_ptr->s_state = EXT2_ERROR_FS;
```

Code fragment 45: /brainix/src/fs/mount.c

Now that we are mounting the root file system, we have to fill in the super block's fields. We start by setting the time of last mount to be `do_time(NULL)` which is, as far as we know so far, an unknown function. We continue our pattern of looking functions up and find `do_time()`. However, it is a messy system call rather than some ordinary function, so we will not exactly look at the code line by line. It simply gets the number of seconds since January 1, 1970. That is the mount time for the super block (line 150).

Recall that `s_mnt_count` is the number of Mounts since last `fsck`. Line 151 simply tells the super block that it is getting mounted one more time.

Note that in the Brainix /brainix/inc/fs/super.h header, we define:

```
89 #define EXT2_ERROR_FS 2 /* Mounted or uncleanly unmounted. */
```

Code fragment 46: /brainix/inc/fs/super.h

so really line 152 of the `mount_root()` method is telling the super block that it's mounted, and nothing more.

```
153      memcpy(super_ptr->s_last_mounted, "/\0", 2);
```

Code fragment 47: /brainix/src/fs/mount.c

memcpy()

We do not know the `memcpy()` method, so allow us to look it up as usual:

```
75 void *memcpy(void *s1, const void *s2, size_t n)
76 {
77
78 /* Copy bytes in memory. */
79
80      char *p1 = s1;
81      const char *p2 = s2;
82
83      for (; n; n--, p1++, p2++)
84           *p1 = *p2;
85      return s1;
86 }
```

Code fragment 48: /brainix/src/lib/string.c

This function is pretty straightforward. There is a pair of dummy variables declared (lines 80-81), and then the addresses are switched in a for-loop (lines 83-84). The buffer `void *s1` is returned after the copying (rather, switching of addresses) has occurred.

Back to the line we left off at in the `mount_root()` method:

```
153      memcpy(super_ptr->s_last_mounted, "/\0", 2);
```

Code fragment 49: /brainix/src/fs/mount.c

This basically tells us that we copy to the `s_last_mounted` component of the super block the null character `\0`, we only copy 2 bytes.

```
154      super_ptr->dev = ROOT_DEV;
155      super_ptr->block_size = 1024 << super_ptr->s_log_block_size;
156      super_ptr->frag_size = super_ptr->s_log_frag_size >= 0 ?
157           1024 << super_ptr->s_log_frag_size :
158           1024 >> -super_ptr->s_log_frag_size;
159      super_ptr->mount_point_inode_ptr = NULL;
```

Code fragment 50: /brainix/src/fs/mount.c

We set the `dev` device for the super block to be the `ROOT_DEV` device. The block size is set to be 2^{10} shifted to the right by `s_log_block_size`, and the fragment is variable depending on whether `s_log_frag_size` is less than zero or not. If it is not less than zero, then you shift 2^{10} to the left by `s_log_frag_size`, otherwise you shift to the right by negative one times `s_log_frag_size`.

```
160      super_ptr->root_dir_inode_ptr = inode_get(ROOT_DEV, EXT2_ROOT_INO);
161      super_ptr->dirty = true;
162 }
```

Code fragment 51: /brainix/src/fs/mount.c

Well, we have yet to cover what exactly the `inode_get()` method is, but we know line 161 is telling us that the super block has changed since we last inspected it, which makes sense since we just obtained it and set its values. Let us now have a more intelligible investigation of the `inode_get()` method (besides simply guessing "Well, it has the words 'get inode' so I'm guessing it gets an inode..."):

inode_get()

```
095 inode_t *inode_get(dev_t dev, ino_t ino)
096 {
097
098 /* Search the inode table for an inode.  If it is found, return a pointer to it.
099  * Otherwise, read the inode into the table, and return a pointer to it. */
100
101      inode_t *inode_ptr;
102
103      /* Search the table for the inode. */
104      for (inode_ptr = &inode[0]; inode_ptr < &inode[NUM_INODES]; inode_ptr++)
105           if (inode_ptr->dev == dev && inode_ptr->ino == ino)
106           {
107                /* Found the inode.  Increment the number of times it is
108                 * used, and return a pointer to it. */
109                inode_ptr->count++;
110                return inode_ptr;
111           }
```

Code fragment 52: /brainix/src/fs/inode.c

The inode table which was initialized previously in section (4.2), we now search the very same table for an inode with the device field equal to `dev` and inode number equal to `ino`. If it is found, the field indicating how many times its been used `count` is incremented (line 109), and the inode pointer to it is returned.

Then there is the case where the inode is not in the table:

```
113      /* The inode is not in the table.  Find a free slot. */
114      for (inode_ptr = &inode[0]; inode_ptr < &inode[NUM_INODES]; inode_ptr++)
115           if (inode_ptr->count == 0)
116           {
```

```
117             /* Found a free slot.  Read the inode into it, and
118              * return a pointer into it. */
119             inode_ptr->dev = dev;
120             inode_ptr->ino = ino;
121             inode_ptr->count = 1;
122             inode_ptr->mounted = false;
123             inode_ptr->dirty = true;
124             inode_rw(inode_ptr, READ);
125             return inode_ptr;
126         }
```

Code fragment 53: /brainix/src/fs/inode.c

We look for the first inode that has absolutely no references whatsoever (line 115). If a free slot is found, then we simply write the inode into the slot, and return a pointer to it.

inode_rw()

However, we also have not seen the method `inode_rw()` before either (line 124). We shall investigate that method now:

```
void inode_rw(inode_t *inode_ptr, bool read)
{

/* If read is true, read an inode from its block into the inode table.
 * Otherwise, write an inode from the table to its block. */

    blkcnt_t blk;
    size_t offset;
    block_t *block_ptr;
    super_t *super_ptr = super_get(inode_ptr->dev);

    if (!inode_ptr->dirty)
        /* The inode in the table is already synchronized with the inode
         * on its block.  No reason to read or write anything. */
        return;
```

Code fragment 54: /brainix/src/fs/inode.c

The basic approach is the same as always, check to see if the inode is not dirty (if it isn't, there's no point in writing what's all ready there).

```
    /* Get the block on which the inode resides. */
    group_find(inode_ptr, &blk, &offset);
    block_ptr = block_get(inode_ptr->dev, blk);
```

Code fragment 55: /brainix/src/fs/inode.c

The `group_find()` method is completely foreign to us! So, you know the drill, let's look its data structure up first:

```
typedef struct
{
    unsigned long bg_block_bitmap;       /* First block of block bitmap. */
    unsigned long bg_inode_bitmap;       /* First block of inode bitmap. */
    unsigned long bg_inode_table;        /* First block of inode table.  */
    unsigned short bg_free_blocks_count; /* Number of free blocks.       */
    unsigned short bg_free_inodes_count; /* Number of free inodes.       */
    unsigned short bg_used_dirs_count;   /* Number of directories.       */
    unsigned short bg_pad;               /* Padding.                     */
    unsigned long bg_reserved[3];        /* Reserved.                    */
} group_t;
```

Code fragment 56: /brainix/inc/fs/group.h

I reiterate a main point: this is exactly identical to the ext2 block group data structure. I won't really go in depth into any analysis of it, but merely presented it to point out what it is and where you can find it.

Meanwhile, the `group_find()` method we still need to analyze: group_find()

```
32 void group_find(inode_t *inode_ptr, blkcnt_t *blk_ptr, size_t *offset_ptr)
33 {
34
35 /* Find where an inode resides on its device - its block number and offset
36  * within that block. */
37
38      super_t *super_ptr;
39      unsigned long group;
40      unsigned long index;
41      block_t *block_ptr;
42      group_t *group_ptr;
43      unsigned long inodes_per_block;
44
45      /* From the superblock, calculate the inode's block group and index
46       * within that block group. */
47      super_ptr = super_get(inode_ptr->dev);
48      group = (inode_ptr->ino - 1) / super_ptr->s_inodes_per_group;
49      index = (inode_ptr->ino - 1) \% super_ptr->s_inodes_per_group;
```

Code fragment 57: /brainix/src/fs/group.c

Again, the brainix style has the dummy variables defined first (lines 38-43). We then find the superblock in order to calculate out the inode's block group and index therein. The super block extraction occurs on line 47, recall from Code fragment 29 the `super_get()` method.

Lines 48-49 uses bitwise operator black magic to actually come up with the group and index therein, but it is valid black magic.

Continuing our analysis of `group_find()`:

```
51      /* From the group descriptor, find the first block of the inode
52       * table. */
53      block_ptr = block_get(inode_ptr->dev, GROUP_BLOCK);
54      group_ptr = (group_t *) block_ptr->data;
55      *blk_ptr = group_ptr[group].bg_inode_table;
56      block_put(block_ptr, IMPORTANT);
```

Code fragment 58: /brainix/src/fs/group.c

We see that line 53 uses `block_get()` which was explored in code fragment 31, it gets the block with the inode's `dev` component, and the `GROUP_BLOCK blkcnt_t`. If you remember we explored briefly the `blkcnt_t` data structure back in code fragment 7...it's basically a (signed) long.

Line 54 casts the data of the `block_ptr` as a `group_t`, which is used later on...the next line as a matter of fact, to find out what the `*blk_ptr` is (or more precisely, assign an address to the `*blk_ptr`).

Then, regardless of the `ROBUST`-ness of the operating system (except for `SLOPPY` of course), line 56 writes the block cache to the disk.

```
58      /* Finally, calculate the block on which an inode resides (it may or may
59       * not be the first block of the inode table) and its offset within that
```

```
60      * block. */
61     inodes_per_block = super_ptr->block_size / super_ptr->s_inode_size;
62     *blk_ptr += index / inodes_per_block;
63     *offset_ptr = (index \% inodes_per_block) * super_ptr->s_inode_size;
64 }
```

Code fragment 59: /brainix/src/fs/group.c

The last thing that occurs is the calculation of the block which an inode resides, and its offset therein. This occurs using the same old bitwise black magic that was seen previously.

Back to the `inode_rw()` method that we left off at:

```
69     /* Get the block on which the inode resides. */
70     group_find(inode_ptr, &blk, &offset);
71     block_ptr = block_get(inode_ptr->dev, blk);
```

Code fragment 60: /brainix/src/fs/inode.c

We assign addresses to `blk` and `offset` by use of `group_find()` on line 70. We then invoke `block_get()` (recall from code fragment 31 what exactly `block_get()` is) to assign an address to `block_ptr`.

```
73     if (read)
74          /* Read the inode from its block into the table. */
75          memcpy(inode_ptr, &block_ptr->data[offset],
76               super_ptr->s_inode_size);
```

Code fragment 61: /brainix/src/fs/inode.c

This basically reads the inode from the block into the table, by means of copying the address using `memcpy()`.

```
77     else
78     {
79          /* Write the inode from the table to its block, and mark the
80           * block dirty. */
81          memcpy(&block_ptr->data[offset], inode_ptr,
82               super_ptr->s_inode_size);
83          block_ptr->dirty = true;
84     }
```

Code fragment 62: /brainix/src/fs/inode.c

This is the alternative case where we are writing to the block from the inode table, and mark the block as dirty. Note which fields of `memcpy()` have changed compared to lines 75-76.

Regardless of which path was taken, we conclude:

```
86     /* Put the block on which the inode resides, and mark the inode in the
87      * table as no longer dirty. */
88     block_put(block_ptr, IMPORTANT);
89     inode_ptr->dirty = false;
90 }
```

Code fragment 63: /brainix/src/fs/inode.c

By putting the block wherein the inode resides, and mark the inode in the inode table as clean as a whistle. Recall, again, code fragment 31 for `block_put()`.

We continue on with our analysis of `inode_get()`:

```
    /* There are no free slots in the table.  Vomit. */
    panic("inode_get", "no free inodes");
    return NULL;
}
```

Code fragment 64: /brainix/src/fs/inode.c

Basically, the file system cries if there are no free slots. It calls a kernel panic, and returns empty handed. Such is life I guess.

We have concluded our investigation of the nitty-gritty details of `mount_root()` but we left off at line 88 of `fs_register()` in fragment 19. We shall return to it here:

```
83      if (block && maj == ROOT_MAJ)
84      {
85           /* The driver for the device containing the root file system is
86            * being registered. */
87           mount_root();
88           dev = maj_min_to_dev(ROOT_MAJ, ROOT_MIN);
89           fs_proc[FS_PID].root_dir = inode_get(dev, EXT2_ROOT_INO);
90           fs_proc[FS_PID].work_dir = inode_get(dev, EXT2_ROOT_INO);
91      }
92 }
```

Code fragment 65: /brainix/src/fs/device.c

Line 87 we just investigated thoroughly, so let us investigate starting with line 88. The `dev` device is assigned based on the device Major and Minor numbers. We have seen `dev_to_maj_min()` but we have not seen `maj_min_to_dev()`, let us try investigating it here:

```
dev_t maj_min_to_dev(unsigned char maj, unsigned char min)
{

/* Build the device number from a major number and a minor number. */

     return ((maj & 0xFF) << 8) | ((min & 0xFF) << 0);
}
```

Code fragment 66: /brainix/src/fs/device.c

This is basically bitwise black magic that undoes the `dev_to_maj_min()`. Note that if we were to look at the `dev` variable as bits (that is, a string of 1s and 0s) and compare it to the `maj` and `min` as bits, we should in theory see that about half way through the `dev` one half looks similar to the `maj` and the other half resembles `min`. That is no coincidence, that is how the `dev` is assigned a value.

Back to the `fs_register()` code:

```
83      if (block && maj == ROOT_MAJ)
84      {
85           /* The driver for the device containing the root file system is
86            * being registered. */
87           mount_root();
88           dev = maj_min_to_dev(ROOT_MAJ, ROOT_MIN);
89           fs_proc[FS_PID].root_dir = inode_get(dev, EXT2_ROOT_INO);
90           fs_proc[FS_PID].work_dir = inode_get(dev, EXT2_ROOT_INO);
91      }
92 }
```

Code fragment 67: /brainix/src/fs/device.c

So line 88 basically constructs a device number out of the `ROOT_MAJ` and `ROOT_MIN` numbers. Lines 89-90 assign the `fs_proc` entries of `FS_PID` (the file system process ID) to have a root directory (line 89). Recall `inode_get()` from code fragment 52 takes in as arguments the device number `dev` and the inode to search the inode table for (`EXT2_ROOT_INO` in our case).

Upon completion of those functions, the `fs_register()` method is complete.

Chapter 3

On The Implementation of Virtual File Systems in Linux and FreeBSD

This paper/chapter/document aims to investigate the implementations of the Virtual File System concept in both Linux and FreeBSD. We shall investigate each implementation in turn. I suppose the best place to start therefore is an explanation of why even bother with a virtual file system. If you are interested in the design and implementation of the Minix 3 virtual file system, there is a thesis out with that very title [16].

So the road map is: we shall observe the history and implementation of the Virtual File System in SunOS, then we shall observe - since SunOS was basically an attempt to hack BSD 4.2 - that the implementation in FreeBSD would (*a priori*) resemble SunOS' virtual file system so FreeBSD would be the next logical system to analyze, and wrap up with the implementation in Linux. More than likely we will need to visit the ext2 file system prior to exploring the implementation of the virtual file system in Linux because the virtual file system in Linux was based off of the ext2 file system data structures.

Note however the overall moral of the story: **virtual file systems are essentially derivatives of the file system that is used locally.**

3.1 History and Motivation for the Virtual File System

In technical jargon, the virtual file system is the abstraction layer over the file system implementations in the operating system. In other words it handles all system calls related to the file system and allow for client programs to access different file systems in a uniform manner; one may think of it therefore as a sort of black box. Sun Microsystems first introduced this virtual file system in

1986 in its SunOS operating system [17][1]. We shall proceed to investigate the structure of the initial SunOS implementation of the virtual file system concept (from the 1986 implementation).

A priori one could say that since it is essentially a black box wherein one makes certain calls and it performs the same actions regardless of the file system, it is intuitively appealing to assert that it is object oriented. Indeed, in most implementations (at least the two that we are interested in) it is object oriented.

As is standard for investigating code, we need to investigate some foundational data structures. We shall follow the pattern set forth by the foundational paper from Sun [17]. It should be noted that the source cited is far more in depth than we hope to get in here. First a brief outline of the relationships of these data structures is in order it seems. The virtual system is modeled after the Unix file system, so the virtual file system's inode is aptly named the **vnode**. All file manipulations are done with this object. The file systems are manipulated through a similar virtual file system object **vfs**. The file system independent layer is usually referred to (originally) as the **vnode layer**.
struct vfs

`struct vfs *vfs`	the next `vfs` in the list
`struct vfsops *vfs_ops`	operations on the vfs struct
`struct vnode *vfs_vnodecovered`	vnode we cover
`int vfs_flags`	generic flags (e.g. read only)
`int vgs_bsize`	the native block size
`caddr_t vfs_data`	private data for the virtual file system

This `struct vfs` data structure is supposed to represent the analog to ye olde mount table entry. I suspect that it is a circularly linked list, but all the raw code shows thus far is that it is a linked list.

The next data structure which is of sequential interest is the `struct vfsops` structure.
struct vfsops

[1]Arguably, for the sake of discussion, it was in SunOS version 2.0 back in 1985 when the first proto-virtual file system was introduced. It allowed UNIX system calls to access local (BSD based) UFS file systems and remote "network file system" file systems transparently. Under SunOS 4.0 the *Stackable* virtual file system was introduced.

`int (*vfs_mount)()`	The mount operation of the file system implementation
`int (*vfs_unmount)()`	The unmount operation of the file system implementation
`int (*vfs_root)()`	This gets the root vnode of the file system implementation
`int (*vfs_statfs)()`	This gets the file stats (e.g. number of links to file) from the file system implementation
`int (*vfs_sync)()`	This synchronizes the file system
`int (*vfs_fid)()`	This gets the vfs' id(?)
`int (*vfs_vget)()`	This gets the vfs struct

These operations are pretty self explanatory. All file systems (well, all the file systems that Sun was interested in supporting at the time) had these methods; they were the system calls made by the kernel that is responsible for the file systems' operations. Since these are the "file system implementation independent" operations, they would constitute the virtual file system operations. That is the sole *raison d'être* of this structure. When a mount system call is made, the vnode for the mount point is looked up and then the `vfs_mount` operation for the file system type is called. If this succeeds, the file system is linked into the linked list of mounted file system, and the `vfs_vnodecovered` field is set to point to the vnode mount point. Please note that this field may be null in the root vfs. The root vfs is **always** first in the list of mounted file systems.

The SunOS implementation had the virtual file systems named according to the path names of their mount points. The implementation didn't support the special device names because the remote file systems (Sun's NFS - network file system) do not necessarily have a unique local device associated with them. The `umount(2)` operation was changed ("upgraded" some might have argued) to `unmount(2)` which takes a path name for a file system mount point rather than of a device.

The interesting reasoning behind the `vfs_root()` function is that by having the root vnode for a mounted file system be obtained by the method rather than always referencing to the object, one can deallocate the root vnode if the file system is not being used. The example given is remote mount points can exist in "embryonic" form, which contains just enough information to actually contact the server and complete the remote mount when the file system is referenced. These mount points can exist with minimal allocated resources when they are not being used.

We shift our focus now onto the `vnode` struct:
struct vnode

u_short v_flag	vnode flags
u_short v_count	reference count
u_short v_shlockc	number of share locks
u_short v_exlockc	number of exclusive locks
struct vfs *v_vfsmountedhere	the covering vfs struct
struct vnodeops *v_ops	the vnode operations
union { struct socket *v_Socket	unix ipc
struct stdata *v_Stream }	stream
struct vfs *v_vfsp	the vfs struct that we are in
enum vtype v_type	the vnode type
caddr_t v_data	the private data

It should probably be noted that there is an enumeration type defined called `vtype`, it has several entries: `VNON`, `VREG`, `VDIR`, `VBLK`, `VCHR`, `VLNK`, `VSOCK`, and `VBAD`. These represent "none of the above", "regular file", "directory", "block device", "character device", "link", "socket", and "unrecognized file format" respectively (or more specifically, the virtual file system counterparts of these). The reerence count in the vnode struct (`v_count`) is maintained by the generic vnode macros `VN_HOLD` and `VN_RELEASE`. The vnode layer and file systems call these macros when the vnode pointers are copied or destroyed. When the last reference to a vnode is destroyed, the `vn_inactive` operation is called to tell the vnode's file system there are no more references to it. The file system either destroys the vnode or caches it for later use. The `v_vfsp` field points to the `vfs` that the vnode belongs to. If the vnode is a mount point, the `v_vfsmountedhere` field points to the vfs for another file system. There is private data in this vnode struct. Well, it is a private data pointer - `v_data` - in the vnode that points to the data that is dependent on the file system.

It should also be noted that vnodes are not licked by the vnode layer. All non-user-advisory locks are done within the file system dependent layer. The SunOS hackers note that locking could have been done in the vnode layer for synchronization purposes without violating the design goals they had intended. But they found it to be unnecessary for their purposes; a microkernel implementation, on the other hand, might find it very necessary.

Now we shall briefly go over the SunOS's `vnodeops`, continue on to discuss the vfs operations, then finish up with kernel interfaces.

struct vnodeops

`int (*vn_open)()`	Performs any open protocol on a vnode, if the open is a "clone" open the operation may return a new vnode
`int (*vn_close)()`	Perform any close protocol on a vnode (e.g. devices); called on the closing of the last reference to the vnode from the file table, if vnode is a device. Called on the last user close of a file descriptor otherwise.
`int (*vn_rdwr)()`	Read or write vnode. It reads or writes a number of bytes to a specified offset of a specified file.
`int (*vn_ioctl)()`	Performs an ioctl on a given vnode
`int (*vn_select)()`	Performs a select on a given vonde
`int (*vn_getattr)()`	Gets the attributes for a given inode, and allocates them in a given vnode pointer.

`int (*vn_setattr)()`	This sets attributes for a given vnode, note that proceeding the analysis of the vnodeops data structure we shall take a peak at the code excerpt of `struct vattr` and not give an indepth explanation but leave the comments to be the explanation
`int (*vn_access)()`	This checks the access permissions for a given vnode
`int (*vn_lookup)()`	This looks up a given component in a given directory
`int (*vn_create)()`	This creates a new file with a given name in a given directory
`int (*vn_remove)()`	This removes a given file in a given directory
`int (*vn_link)()`	This links a given inode to a target name in a given target directory
`int (*vn_rename)()`	This renames a given file in a given directory to a given new name
`int (*vn_mkdir)()`	This creates a new directory in a given directory
`int (*vn_rmdir)()`	This removes a given directory
`int (*vn_readdir)()`	This reads entries from a given directory
`int (*vn_symlink)()`	Symbolically links the given path to a new given name in a destination directory
`int (*vn_readlink)()`	Read a given symbolic link.
`int (*vn_fsync)()`	Writes out all cached information for a given file
`int (*vn_inactive)()`	A given vnode is no longer referenced and may therefore be deallocated.

`int (*vn_bmap)()`	Maps a given logical block number in a given file to a physical block number and physical device
`int (*vn_strategy)()`	Block oriented interface to read or write a logical block from a file into or out of a given buffer.
`int (*vn_bread)()`	Reads a given logical block from a given file and returns a pointer to a (given) header buffer
`int (*vn_brelse)()`	The buffer returned by `vn_bread` can be released

Now, as promised, the code pseudo-explanation of the `struct vattr`:

```
struct vattr {
     enum vtype va_type; /* vnode type */
     u_short va_mode; /* acc mode */
     short va_uid; /* owner uid */
     short va_gid; /* owner gid */
     long va_fsid; /* fs id */
     long va_nodeid; /* node number */
     short va_nlink; /* number of links */
     u_long va_size; /* file size */
     long va_blocksize; /* block size */
     struct timeval vs_atime; /* last accessed */
     struct timeval vs_mtime; /* last modified */
     struct timeval va_ctime; /* last chg */
     dev_t va_rdev; /* device */
     long va_blocks; /* space used */
};
```

Code fragment 68:

I'm sorry it's so cryptic, but that's the best I could present it.

Now, we shift our focus to the kernel interfaces; that is a veneer layer that is provided over the generic vnode interface to make it easier for kernel subsystems to manipulate files.

Kernel Interfaces

`vn_open`	Perform permission checks and then open a vnode given by a path name
`vn_close`	close a vnode
`vn_rdwr`	build a *uio* structure and read/write a vnode
`vn_create`	perform permission checks and then create a vnode given by a path name
`vn_remove`	remove a node given by a path name

`vn_link`	link a node given by a source path name to a target given by a target path
`vn_rename`	rename a node given by a source path name to a target given by a target path name
`VN_HOLD`	increment the vnode reference count
`VN_RELE`	decrement the vnode reference count and call `vn_inactive` if this is the last reference

This basically completes the crude scaffolding of the virtual file system that Sun Microsystems begat.

3.2 FreeBSD's Implementation of the Virtual File System

It is strongly recommended to consult the FreeBSD book chapter 6.5 [4]. It is most unfortunate that, unlike the Linux project, the FreeBSD project does not contain documentation of all most every aspect of the code of the operating system. Indeed, it appears to be the only documentation available on the FreeBSD virtual file system except the raw code itself. Unfortunately, we are forced to go code diving as well, into the 6.2 FreeBSD sourcecode (available from at least one cross-reference [23]).

However, we are told [4] that the vnode is/has the following:

- an extensible object-oriented interface.
- Flags that are used for identifying generic attributed.
- Various reference counts including the number of file entries that are open for reading and/or writing that reference the vnode, the number of file entries that are open for writing that reference the vnode, etc.
- A pointer to the mount structure that describes the file system which contains the vnode struct.
- Various information to do file read-ahead.
- Reference to the `vm_object` associated with the vnode.
- A mutex to protect the flags and counters within the vnode.
- A lock manager to protect pairs of the vnode that may change while some input/output operations in progress.
- Fields used by the name cache to track the names associated with the vnode.

- A pointer to private information needed for identifying the underlying object.
- The typpe of the underlying object.
- Clean and dirty buffers associated with the vnode.
- A count of the number of buffer write operations in progress.

From this survey, it appears that the structure of the SunOS virtual file system implementation has been preserved and expanded upon. There is a serious difference noted by [4]: "Unlike the original Sun Microsystems vnode implementation, the one in FreeBSD allows dynamic addition of vnode operations either at system boot time or when a new file system is dynamically loaded into the kernel." Unfortunately it appears that an insufficient amount has changed to note the differences, as far as I can tell[2]. Please do note that the data structures are defined in the /FreeBSD/sys/sys/ directory (available here [24]). Naturally the data structures are more complex than the ones outlined in the SunOS implementation above, but they retain the same basic character.

3.3 The EXT2 File System

The ext2 file system is a rather interesting file system for one that is forked from the Minix file system. A Hand wavy explanation of the data structures will be given, although in future revisions I hope to go into greater detail (in the spare time on will have to make due with the book *Understanding the Linux Kernel* - it's a good book). Also the design and implementation of the Ext2 file system is very good [25], it's written by the creators of the file system. They should explain it best.

Note that the file system is notionally written for little endian, so `__le32` refers to a 4 byte little endian int, `__le16` refers to a little endian short, `__le6` refers to a little endian char; also `__u32`, `__u16`, `__u8` refer to unsigned longs, shorts, and chars (respectively) and `__s32`, `__s16`, `__s8` refer to signed longs, shorts, and chars (respectively).

Disks are divided into 512 byte atoms called **"sectors"**. There are "disk molocules" (that is a collection of these atoms) called **"blocks"**. A block consists of 2, 4, 8, or sometimes (rarely) 16 sectors. Smaller block sizes means less space is wasted but there is more accounting overhead, on the other hand large block sizes means more space is wasted but less accounting overhead. There are also other limits imposed on the file system (for a detailed account, see [26]).

So we further organize the hard disk into **block groups** which is a collection of blocks. The first block is the super block, the second and third blocks are the block usage bitmap and the inode usage bitmap. If you do a little math, you realize that the maximum size of a block group is 8 times the size of a block.

[2]Disclaimer: I did a really lazy job checking!

Each block group has a portion of the blocks dedicated to the inodes which track the rest of the group.

Let us inspect the ext2_group_desc data structure:
ext2_group_desc

`__le32 bg_block_bitmap`:	Block number of the block bitmap.
`__le32 bg_inode_bitmap`:	Block number of the inode bitmap.
`__le32 bg_inode_table`:	Block number of the first inode.
`__le16 bg_free_blocks_count`:	Number of free blocks in the group
`__le16 bg_free_inodes_count`:	Number of free inodes in the group
`__le16 bg_used_dirs_count`:	Number of directories in the group
`__le16 bg_pad`:	Alignment to word.
`__le32 bg_reserved[3]`:	Nulls to pad out to 24 bytes.

The **superblock** is the most important block because it contains all the information about the configuration of the file system. The primary copy is stored at an offset of 1024 bytes from the start of the device, and it is essential for booting the device. Originally while debugging the file system, back ups for the superblock were placed in every block group. Now they are placed in the first two block groups, as well as all powers of 3, 5, and 7.

Let us examine the superblock in the same manner we examined the data structures of the SunOS implementation of the virtual file system.
struct ext2_super_block

`__le32 s_inodes_count`:	The total number of inodes
`__le32 s_blocks_count`:	File system size in blocks
`__le32 s_r_blocks_count`:	Number of reserved blocks
`__le32 s_free_blocks_count`:	The Free blocks Counter
`__le32 s_free_inodes_count`:	Free inodes counter
`__le32 first_data_block`:	Number of the first data block we'll be using (it's notionally set to 1 always)
`__le32 s_log_block_size`:	The logarithm of the block size
`__le32 s_log_frag_size`:	The logarithm of the fragment size
`__le32 s_blocks_per_group`:	Number of blocks per group.
`__le32 s_frags_per_group`:	Number of fragments per group.
`__le32 s_inodes_per_group`:	Number of inodes per group
`__le32 s_mtime`:	Time of the last mount operations

`__le32 s_wtime`:	Time of the last write operations
`__le16 s_mnt_count`:	Mount operations counter
`__le16 s_max_mnt_count`:	The number of mount operations before we need to check the hard drive.
`__le16 s_magic`:	The magic signature for the ext2 file system
`__le16 s_state`:	The status flag.
`__le16 s_errors`:	Behavior when detecting errors
`__le16 s_minor_rev_level`:	Minor Revision Level
`__le32 s_lastcheck`:	Time of last check
`__le32 s_checkinterval`:	Time between checks
`__le32 s_creator_os`:	OS where filesystem was created
`__le32 s_rev_level`:	Revision level of the file system
`__le16 s_def_resuid`:	Default UID for reserved blocks
`__le16 s_def_resgid`:	Default user group ID for reserved blocks
`__le32 s_first_ino`:	Number of first nonreserved inode
`__le16 s_inode_size`:	Size of on-disk inode structure
`__le16 s_block_group_size`:	Block group number of this superblock
`__le16 s_block_group_nr`:	Compatible features bitmap.
`__le32 s_feature_compat`:	Incompatible features bitmap
`__le32 s_feature_ro_compat`:	Read-only compatible features bitmap
`__u8[16] s_uuid`:	128-bit file system identifier
`char[16] s_volume_name`:	The volume name
`char[64] s_last_mounted`:	Pathname of last mount point
`__le32 s_algorithm_usage_bitmap`:	This is used for compression
`__u8 s_prealloc_blocks`:	Number of blocks to preallocate
`__u8 s_prealloc_dir_blocks`:	Number of blocks to preallocate for directories
`__u16 s_padding1`:	Alignment to word
`__u32[204] s_reserved`:	Nulls out to 1024 bytes

A few minor notes regarding the structure of this structure. The `s_inodes_count` field holds the number of inodes, while `s_blocks_count` holds the number of blocks in the Ext2 file system.

The `s_log_block_size` field expresses the logarithm (base 2) block size of the file system, using 1024 as the base unit. That is, when the block size is 1024 bytes `s_log_block_size` is 0, when the block size is 2048 bytes `s_log_block_size` is 1, when the block size is 4096 bytes `s_log_block_size` is 2, etc. In the Ext2 file system, block fragmentation is not implemented so `s_log_frag_size` is equal to `s_log_block_size`.

The `s_blocks_per_group`, `s_frags_per_group`, and `s_inodes_per_group` fields store the number of blocks, fragments, and inodes (respectively) in each block group.

The fields `s_mnt_count`, `s_max_mnt_count`, `s_lastcheck`, and `s_checkinterval` fields are used so the Ext2 file system can check for consistency automatically at boot time.

So if we take, say, the block group 0 and look at it, we find that the it has the following organization:

- 1 Block is dedicated to the super block (block 0)
- n blocks are dedicated to be the group descriptors (blocks 1 through $1+n$)
- 1 block is dedicated to be the data block bitmap (block $n+1$ to $n+2$)
- 1 block is dedicated to be the inode block bitmap (block $n+2$ to $n+3$)
- m blocks are dedicated to the inodes (blocks $n+3$ to $n+m+3$)
- what is remaining is dedicated to the the actual data blocks (blocks $n+m+3$ to the end).

Typically n is small (24 bytes), and m is (ideally) small too.

Let us now inspect the inodes and the inode table. The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes. The block number of the first block of the inode table is stored in the `bg_inode_table` field of the group descriptor struct we just inspected. All inodes in the Ext2 file system have the uniform size of 128 bytes. Thus a 1024 byte block contains 8 inodes, whereas a 4096 byte block contains 32 inodes. Now using simple algebra, one can deduce the inode table occupies N blocks where N is the total number of inodes in a group (`s_inodes_per_group`) by the number of inodes per block. The `ext2_inode` is rather interesting:
ext2_inode

`__le16 i_mode`:	File type and access rights
`__le16 i_uid`:	Owner Identification
`__le32 i_size`:	File size in bytes
`__le32 i_atime`:	Time of last access
`__le32 i_ctime`:	Time that inode last changed
`__le32 i_mtime`:	Time that file contents last changed
`__le32 i_dtime`:	Time of file deletion
`__le16 i_gid`:	User group identifier

`__le16 i_links_count`:	Hard links counter
`__le32 i_blocks`:	Number of data blocks of the file
`__le32 i_flags`:	File flags
`union osdi`:	Specific operating system information (1)
`__le32 i_block[EXT2_N_BLOCKS]`:	The pointers to the data blocks
`__le32 i_generation`:	File version (used when the file is accessed by a network file system)
`__le32 i_file_acl`:	File Access control list
`__le32 i_dir_acl`:	The directory access control list
`__le32 i_faddr`:	The fragment address
`union osd2`:	Specific operating system information (2)

Part III

Tutorials on Using the Brainix Operating System

Chapter 4

How to run Brainix on Unix-like Operating Systems and Bochs

OK, you have downloaded your copy of brainix by first installing subversion (you need to install subversion!), then type in:

`$ svn checkout http://brainix.googlecode.com/svn/trunk/ brainix`

It should start downloading. You wait until its downloading is complete. I assume that the working directory is `/home/your_user_name_here/`. You type into the command line:

`$ mkdir mnt`

Then go into the Brainix directory. Create several scripts:

```
#!/usr/bin/bash

make clean
make
```

Code fragment 69: brainix/compile.sh

The `compile.sh` script.

```
#!/usr/bin/bash

sudo mount -o loop Bootdisk.img ../mnt/
sudo cp ./bin/Brainix ../mnt
sudo rm ../mnt/Brainix.gz
sudo gzip ../mnt/Brainix
sudo umount ../mnt/
```

Code fragment 70: brainix/update_bootdisk.sh

This is the `update_bootdisk` script that updates the boot disk with the Brainix binary image (the Brainix binary image is made after compilation and put in /brainix/bin). Now, to make a bochs `bochsrc` file. I have included a `dot-bochsrc` file in this documentation (its external to this file), I use it to run Brainix in bochs.

Now to put it all together:

```
#!/usr/bin/bash

sh compile.sh
sudo sh update-bootdisk.sh
bochs -f dot-bochsrc
```

Code fragment 71: brainix/run.sh

This is the script you invoke in order to compile and run the compiled image in bochs.

I have included sample scripts **THAT YOU NEED TO MOVE TO THE BRAINIX FOLDER**, i.e. type into the command line:

```
$ mv compile.sh ../
$ mv update-bootdisk.sh ../
$ mv dot-bochsrc ../
$ mv run.sh ../
```

Chapter 5

How to hack Brainix: Some Things to Bear In Mind

So you're an eager young (or not so young) programmer who wants to start fiddling around with Brainix's code and you'd like to submit what you've done. There are some things that you've go to bear in mind with regard to the documentation and the lisence.

First and foremost, if you modify pre-existing code, logically it changes how the code works. So you need to modify the documentation. It's important that other people know what the hell is going on, and no one knows that better than the person who changed it![1] It helps no one if the explanation is in your noodle!

To add a chapter to the Brainix manual, be sure to make your new file a .tex file and make it begin:

```
\chapter{My Chapter Name}
```

If you would like to add code, simply type:

```
\begin{code}{path/to/code/file} ... \end{code}
```

Then you are golden...provided you add the code and have a unix-like file path.

Also, don't forget that you **should** comment at the top the changelog. That is, the date, your name, and a one line explanation of what you did. Take for example:

```
/* 20 June 2007 A. R. Hacker, modified the main()
 * function to include a better idle() implementation. */
```

[1] A funny anecdote, one day Brainix and Pqnelson were hacking on the kernel and Pqnelson randomly tinkered with some of the code hoping to solve a problem at the time. He succeeded but had no clue what he did. In rare events like this, make sure that the problem is solved and upload your modifications; then you can see what changes were made by the SVN interface on the command line. You can then go and change the documentation

There is enough information to know exactly where to look for the changed code. Be sure not to have too much information, e.g. "modified the main() function to include a modified idle() process fork such that an infinite for-loop is used rather than an infinite for-loop so the operating system won't go to hell after the file system registers the floppy disk." This is like listening to the crazy Aunt that rambles on about everything at the family reunion. On the other hand, don't include too little information, e.g. "modified main()". This tells us nothing about what you did! It's too cryptic and short, what happened with the `main()` function? Was it completely revamped or did you add a semicolon somewhere?

Don't forget, if you add a completely new file, to include the GPL copyright information. This is in addition to writing the documentation for it, of course.

Chapter 6

On the Implementation of the debugging for Brainix

It seems all too evident that a system requiring programming will require debugging, especially for something as important as an operating system! So it seemed all too obvious that, by virtue of Murphy's Law (if something can go wrong, it will), we need to implement debugging features quickly. The impromptu solution was to present a debug function:

```
void debug(unsigned int priority, char *message, ...);
void dbug(char *message, ...);
```

Code fragment 72: /brainix/src/drivers/video.c

Where the priority is ranked from 1 to some large number, 1 being the most important, and compared to `DUM_DBUG` a constant defined in `kernel.h`. If you want to turn off debugging features, set it to -1. The higher the value for `priority`, the more esoteric the debugging results. Note that there is a shorter function `dbug` which essentially calls `debug` with a priority of `1`.

6.1 FAQ

6.1.1 What is the format for debug?

The standard format should be

```
void debug(unsigned int priority-SYS_ESTERIC, "file_name.method(): your message here", ...);
```

Code fragment 73: typical debug

The `SYS_ESOTERIC` indicates the system (file system, driver, kernel, where-ever the debug is working in) debug emphasis. So if you have debug *ONLY* the kernel, or *ONLY* a part of the operating system, you go to its header (unless its a driver, then its defined in the driver's .c file) and you change the `SYS_ESOTERIC` to 10 or something bigger. This will cause the debugger to print out only the messages from this system.

6.1.2 I'm making a new driver, and I don't know what to do about this debugger...

Well, suppose your driver file is `X.c`. Near the top, insert the following code:

```
#define X_ESOTERIC     0
```

Code fragment 74: /brainix/src/drivers/X.c

and simply put debugging information that you might find useful as

```
    debug(1-X_ESOTERIC, "X.method(): debugging information...");
```

Code fragment 75: /brainix/src/drivers/X.c

If you are debugging your dear code, then set `X_ESOTERIC` to be some large number like 10. And you don't have to set the value to `1-X_ESOTERIC` it could be any positive number minus `X_ESOTERIC`.

Bibliography

[1] The Skelix Operating System Introduction to writing (an operating system including) a file system from scratch `http://en.skelix.org/skelixos/tutorial07.php`

[2] Andrew Tanenbaum and Albert Woodhull, *Operating Systems: Design and Implementation* Third Edition, Upper Saddle River, New Jersey: Pearson Prentice Hall (2006).

[3] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel* Sebastopol: O'Reilly Media Inc. (2006).

[4] Marshall Kirk McKusick and George V. Neville-Neil *The Design and Implementation of the FreeBSD Operating System: FreeBSD Version 5.2* Upper Saddle River: Pearson Education, Inc. (2005).

[5] Dr. Matloff's "File Systems in Unix" `http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Unix/FileSyst.html`

[6] Unix File System `http://unixhelp.ed.ac.uk/concepts/fsystem.html`

[7] The Linux File System Explained, by Mayank Sarup `http://www.freeos.com/articles/3102/`

[8] Chapter 9 of *The Linux Kernel* Available `http://www.tldp.org/LDP/tlk/fs/filesystem.html`

[9] Inode Definition by the Linux Information Project `http://www.bellevuelinux.org/inode.html`

[10] Understanding the Filesystem Inodes `http://www.onlamp.com/pub/a/bsd/2001/03/07/FreeBSD_Basics.html`

[11] David A Rusling, The Second Extended File system (EXT2) `http://www.science.unitn.it/~fiorella/guidelinux/tlk/node97.html`

[12] Understanding the Linux Kernel Second Edition Excerpt `www.oreilly.com/catalog/linuxkernel2/chapter/ch17.pdf`

[13] "Setting your file creation mask" http://osr507doc.sco.com/en/OSTut/Setting_your_file_creation_mask.html

[14] Kevin T. Van Maren *The Fluke Device Driver Framework* (1999). http://www.cs.utah.edu/flux/papers/vanmaren-thesis.pdf

[15] David A Rusling *The Linux Kernel* Chapter 9 http://www.tldp.org/LDP/tlk/fs/filesystem.html

[16] Balazs Gerofi *MINIX VFS: Design and implementation of the MINIX Virtual File system* http://www.minix3.org/doc/gerofi_thesis.pdf

[17] S.R. Kleiman *Vnodes: An Architecture for Multiple File System Types in Sun UNIX* In Proc. of the Summer 1986 USENIX Technical Conference, June 1991, pp 238-247 http://www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf

[18] Neil Brown et al. *The Linux Virtual File-system Layer* (1999) http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html

[19] Richard Gooch et al. *Overview of the Linux Virtual File System* (2005) http://www.gelato.unsw.edu.au/lxr/source/Documentation/kernel-docs.txt

[20] Max Bruning *A Comparison of Solaris, Linux, and FreeBSD Kernels* (2005) http://www.opensolaris.org/os/article/2005-10-14_a_comparison_of_solaris__linux__and_freebsd_kernels/

[21] Vans Information *Virtual file system Part 1* (2001) http://freeos.com/articles/3838/

[22] Vans Information *Virtual file system Part 2* (2001) http://freeos.com/articles/3851/

[23] http://fxr.watson.org/fxr/source/?v=RELENG62

[24] http://fxr.watson.org/fxr/source/sys/?v=RELENG62

[25] Remy Card, Theodore Ts'o, and Stephen Tweedie, *The Design and Implementation of the Ext2 File System* (1994) http://web.mit.edu/tytso/www/linux/ext2intro.html

[26] Anon. *The Ext2 Linux Documentation* (for kernel 2.6.20.1) http://lxr.linux.no/source/Documentation/filesystems/ext2.txt