On the Brainix File System*

pqnelson - official File System Hacker

June 18, 2007

> " '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]

 This, like the rest of the Brainix Project, is a work in progress.

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)
          {
               case OP_ONE: /* ... */ break;
               /* other replies that require modifications */
```

*This is specifically for revision 68, and the revision number will change as the file system changes

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

```
            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 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.

2.1 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]

[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`

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.

2.2 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:

```
15 extern struct INODE iroot;
16
17 #define MAX_NAME_LEN 11
18
19 struct DIR_ENTRY {
20      char de_name[MAX_NAME_LEN];
21      int de_inode;
22 };
```

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.

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 mo-

ment 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.

4 File System Initialization

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

```
34 void fs_main(void)
35 {
36      /* Initialize the file system. */
37      block_init(); /* Initialize the block cache. */
38      inode_init(); /* Initialize the inode table. */
39      super_init(); /* Initialize the superblock table. */
40      dev_init();   /* Initialize the device driver PID table. */
41      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.

4.1 block_init()

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

```
43 /* A cached block is a copy in RAM of a block on a device: */
44 typedef struct block
45 {
46      /* The following field resides on the device: */
47      char data[BLOCK_SIZE]; /* Block data. */
```

[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.

```
48
49      /* The following fields do not reside on the device: */
50      dev_t dev;              /* Device the block is on.            */
51      blkcnt_t blk;           /* Block number on its device.        */
52      unsigned char count;    /* Number of times the block is used. */
53      bool dirty;             /* Block changed since read.          */
54      struct block *prev;     /* Previous block in the list.        */
55      struct block *next;     /* Next block in the list.            */
56 } block_t;
```

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

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:

```
48 /* Used for device IDs: */
49 #ifndef _DEV_T
50 #define _DEV_T
51 typedef unsigned long dev_t;
52 #endif
```

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

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!

```
30 /* Used for file block counts: */
31 #ifndef _BLKCNT_T
32 #define _BLKCNT_T
33 typedef long blkcnt_t;
34 #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.

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

```
32 void block_init(void)
33 {
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.

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.

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

```
32 void inode_init(void)
33 {
34
35 /* Initialize the inode table. */
36
37      inode_t *inode_ptr;
38
39      /* Initialize each slot in the table. */
40      for (inode_ptr = &inode[0]; inode_ptr < &inode[NUM_INODES]; inode_ptr++)
41      {
42           inode_ptr->dev = NO_DEV;
43           inode_ptr->ino = 0;
44           inode_ptr->count = 0;
45           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.

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.

```
35 /* The superblock describes the configuration of the file system: */
36 typedef struct
37 {
38      /* The following fields reside on the device: */
39      unsigned long s_inodes_count;      /* Total number of inodes.         */
40      unsigned long s_blocks_count;      /* Total number of blocks.         */
41      unsigned long s_r_blocks_count;    /* Number of reserved blocks.      */
42      unsigned long s_free_blocks_count; /* Number of free blocks.          */
43      unsigned long s_free_inodes_count; /* Number of free inodes.          */
44      unsigned long s_first_data_block;  /* Block containing superblock.    */
45      unsigned long s_log_block_size;    /* Used to compute block size.     */
46      long s_log_frag_size;              /* Used to compute fragment size.  */
47      unsigned long s_blocks_per_group;  /* Blocks per group.               */
48      unsigned long s_frags_per_group;   /* Fragments per group.            */
49      unsigned long s_inodes_per_group;  /* Inodes per group.               */
50      unsigned long s_mtime;             /* Time of last mount.             */
51      unsigned long s_wtime;             /* Time of last write.             */
52      unsigned short s_mnt_count;        /* Mounts since last fsck.         */
53      unsigned short s_max_mnt_count;    /* Mounts permitted between fscks. */
54      unsigned short s_magic;            /* Identifies as ext2.             */
55      unsigned short s_state;            /* Cleanly unmounted?              */
56      unsigned short s_errors;           /* What to do on error.            */
57      unsigned short s_minor_rev_level;  /* Minor revision level.           */
58      unsigned long s_lastcheck;         /* Time of last fsck.              */
59      unsigned long s_checkinterval;     /* Time permitted between fscks.   */
60      unsigned long s_creator_os;        /* OS that created file system.    */
61      unsigned long s_rev_level;         /* Revision level.                 */
62      unsigned short s_def_resuid;       /* UID for reserved blocks.        */
63      unsigned short s_def_resgid;       /* GID for reserved blocks.        */
64      unsigned long s_first_ino;         /* First usable inode.             */
65      unsigned short s_inode_size;       /* Size of inode struct.           */
66      unsigned short s_block_group_nr;   /* Block group of this superblock. */
67      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.

4.4 dev_init()

4.4.1 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:

```
112 #ifndef _PID_T
113 #define _PID_T
114 typedef long pid_t;
115 #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.

4.4.2 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.

4.5 descr_init()

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

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

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

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

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:

```
45 /* Process-specific file system information: */
46 typedef struct
47 {
48      inode_t *root_dir;                /* Root directory.            */
49      inode_t *work_dir;                /* Current working directory. */
50      mode_t cmask;                     /* File mode creation mask.   */
51      file_ptr_t *open_descr[OPEN_MAX]; /* File descriptor table.     */
52 } 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()

```
32 void descr_init(void)
33 {
34
35 /* Initialize the file pointer table and the process-specific file system
36  * information table. */
37
38      int ptr_index;
39      pid_t pid;
40      int descr_index;
41
42      /* Initialize the file pointer table. */
43      for (ptr_index = 0; ptr_index < NUM_FILE_PTRS; ptr_index++)
```

```
44      {
45           file_ptr[ptr_index].inode_ptr = NULL;
46           file_ptr[ptr_index].count = 0;
47           file_ptr[ptr_index].offset = 0;
48           file_ptr[ptr_index].status = 0;
49           file_ptr[ptr_index].mode = 0;
50      }
51
52      /* Initialize the process-specific file system information table. */
53      for (pid = 0; pid < NUM_PROCS; pid++)
54      {
55           fs_proc[pid].root_dir = NULL;
56           fs_proc[pid].work_dir = NULL;
57           fs_proc[pid].cmask = 0;
58           for (descr_index = 0; descr_index < OPEN_MAX; descr_index++)
59                fs_proc[pid].open_descr[descr_index] = NULL;
60      }
61 }
```

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!

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`.

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()

```
70 void fs_register(bool block, unsigned char maj, pid_t pid)
71 {
72
73 /* Register a device driver with the file system - map a device's major number
74  * to its driver's PID.  If the driver for the device containing the root file
75  * system is being registered, mount the root file system and initialize the
76  * root and current working directories. */
77
78      dev_t dev;
79
80      /* 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.

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

```
130 void mount_root(void)
131 {
132
133 /* Mount the root file system. */
134
135      super_t *super_ptr;
136
137      /* Open the device. */
138      dev_open_close(ROOT_DEV, BLOCK, OPEN);
139      if (err_code)
140           /* The device could not be opened. */
141           panic("mount_root", strerror(err_code));
```

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

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: dev_open_close()

```
146 int dev_open_close(dev_t dev, bool block, bool open)
147 {
148
149 /* If open is true, open a device.  Otherwise, close a device. */
150
151      unsigned char maj, min;
152      pid_t pid;
153      msg_t *m;
154      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).

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

```
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()

```
165 block_t *block_get(dev_t dev, blkcnt_t blk)
166 {
167
168 /* Search the cache for a block.  If it is found, return a pointer to it.
169  * Otherwise, evict a free block, cache the block, and return a pointer to
170  * it. */
171
172     block_t *block_ptr;
173
174     /* Search the cache for the block. */
175     for (block_ptr = lru->prev; ; )
176          if (block_ptr->dev == dev && block_ptr->blk == blk)
177          {
178               /* Found the block.  Increment the number of times it is
179                * used, mark it recently used, and return a pointer to
180                * it. */
181               block_ptr->count++;
182               recently_used(block_ptr, MOST);
183               return block_ptr;
184          }
185          else if ((block_ptr = block_ptr->prev) == lru->prev)
186               /* Oops - we've searched the entire cache already. */
187               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.

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

```
97      if (!block_ptr->dirty)
98           /* The cached block is already synchronized with the block on
```

```
99            * its device.  No reason to read or write anything. */
99           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.

```
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 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()`:

```
210      /* There are no free blocks in the cache.  Vomit. */
211      panic("block_get", "no free blocks");
212      return NULL;
213 }
```

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:

```
93      block_put(block_ptr, IMPORTANT);
94
95      return super_ptr;
96 }
```

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:

```
149      /* Perform the mount.  Fill in the superblock's fields. */
150      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

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

```
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.

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

```
53 void inode_rw(inode_t *inode_ptr, bool read)
54 {
55
56 /* If read is true, read an inode from its block into the inode table.
57  * Otherwise, write an inode from the table to its block. */
58
59      blkcnt_t blk;
60      size_t offset;
61      block_t *block_ptr;
62      super_t *super_ptr = super_get(inode_ptr->dev);
63
64      if (!inode_ptr->dirty)
65           /* The inode in the table is already synchronized with the inode
66            * on its block.  No reason to read or write anything. */
67           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).

```
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 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:

```
34 typedef struct
35 {
36      unsigned long bg_block_bitmap;       /* First block of block bitmap. */
37      unsigned long bg_inode_bitmap;       /* First block of inode bitmap. */
38      unsigned long bg_inode_table;        /* First block of inode table.  */
39      unsigned short bg_free_blocks_count; /* Number of free blocks.       */
40      unsigned short bg_free_inodes_count; /* Number of free inodes.       */
41      unsigned short bg_used_dirs_count;   /* Number of directories.       */
42      unsigned short bg_pad;               /* Padding.                     */
43      unsigned long bg_reserved[3];        /* Reserved.                    */
44 } 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()`:

```
128      /* There are no free slots in the table.  Vomit. */
129      panic("inode_get", "no free inodes");
130      return NULL;
131 }
```

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:

```
44 dev_t maj_min_to_dev(unsigned char maj, unsigned char min)
45 {
46
47 /* Build the device number from a major number and a minor number. */
48
49      return ((maj & 0xFF) << 8) | ((min & 0xFF) << 0);
50 }
```

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.

References

[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`