On the Brainix File System*

pqnelson - official File System Hacker

June 17, 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]

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.

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 microkernel:

```
 /* 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") auxilliary 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 processemust 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 quadrouple 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 lookup 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 tothink 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, OpenBSD, NetBSD, Darwin, and other operating systems because Portage is distributed via its source code. It works by downloading and compiling source code automagically 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:

```
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:

```
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.

```
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()

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:

```
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 13: /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`).

4.5 descr_init()

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

```
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 14: /brainix/src/fs/fildes.c

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

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