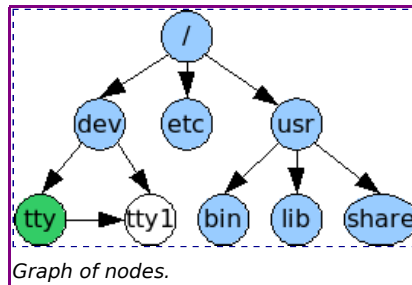


8. The VFS and the initrd

In this chapter we're going to be starting work on our virtual filesystem (VFS). As a baptism of fire, we will also be implementing an initial ramdisk so you can load configuration files or executables to your kernel.

8.1. The virtual filesystem

A VFS is intended to abstract away details of the filesystem and location that files are stored, and to give access to them in a uniform manner. They are usually implemented as a graph of nodes; Each node representing either a file, directory, symbolic link, device, socket or pipe. Each node should know what filesystem it belongs to and have enough information such that the



relavent open/close/etc functions in its driver can be found and executed. A common way to accomplish this is to have the node store function pointers which can be called by the kernel. We'll need a few function pointers:

- *Open* - Called when a node is opened as a file descriptor.
- *Close* - Called when the node is closed.
- *Read* - I should hope this was self explanatory!
- *Write* - Same as above :-)
- *Readdir* - If the current node is a directory, we need a way of enumerating it's contents. Readdir should return the n'th child node of a directory or NULL otherwise. It returns a 'struct dirent', which is compatible with the UNIX readdir function.
- *Finddir* - We also need a way of finding a child node, given a name in string form. This will be used when following absolute pathnames.

So far then our node structure looks something like:

```

typedef struct fs_node
{
    char name[128];    // The filename.
    u32int flags;      // Includes the node type (Directory, file etc).
    read_type_t read;  // These typedefs are just function pointers. We'll define them later!
    write_type_t write;
    open_type_t open;
    close_type_t close;
    readdir_type_t readdir; // Returns the n'th child of a directory.
    finddir_type_t finddir; // Try to find a child in a directory by name.
} fs_node_t;
  
```

Obviously we need to store the filename, and flags contains the type of the node (directory, symlink etc), but we are still missing things. We need to know what permissions the file has, which user/group it belongs to, and possibly also its length.

```

typedef struct fs_node
  
```

1. Environment setup
2. Genesis
3. The Screen
4. The GDT and IDT
5. IRQs and the PIT
6. Paging
7. The Heap
8. The VFS and the initrd
9. Multitasking
10. User Mode

```

{
    char name[128];    // The filename.
    u32int mask;        // The permissions mask.
    u32int uid;         // The owning user.
    u32int gid;         // The owning group.
    u32int flags;       // Includes the node type.
    u32int length;      // Size of the file, in bytes.
    read_type_t read;
    write_type_t write;
    open_type_t open;
    close_type_t close;
    readdir_type_t readdir;
    finddir_type_t finddir;
} fs_node_t;

```

Again though, we are still missing things! We need a way for the filesystem driver to track which node is which. This is commonly known as an *inode*. It is just a number assigned by the driver which uniquely represents this file. Not only that, but we may have *multiple instances of the same filesystem type*, so we must also have a variable that the driver can set to track which filesystem instance it belongs to.

Lastly we also need to account for symbolic links (shortcuts in Windows-speak). These are merely pointers or placeholders for other files, and so need a pointer member variable.

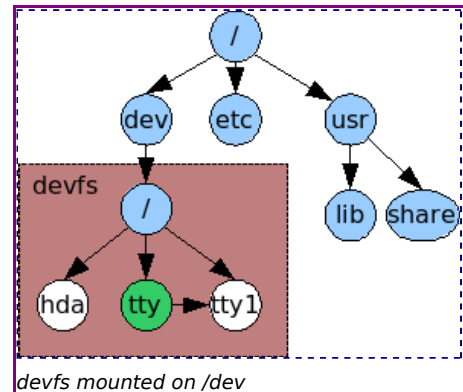
```

typedef struct fs_node
{
    char name[128];    // The filename.
    u32int mask;        // The permissions mask.
    u32int uid;         // The owning user.
    u32int gid;         // The owning group.
    u32int flags;       // Includes the node type. See #defines above.
    u32int inode;       // This is device-specific - provides a way for a filesystem to identify files.
    u32int length;      // Size of the file, in bytes.
    u32int impl;        // An implementation-defined number.
    read_type_t read;
    write_type_t write;
    open_type_t open;
    close_type_t close;
    readdir_type_t readdir;
    finddir_type_t finddir;
    struct fs_node *ptr; // Used by mountpoints and symlinks.
} fs_node_t;

```

8.1.1. Mountpoints

Mountpoints are the UNIX way of accessing different filesystems. A filesystem is mounted on a directory - any subsequent access to that directory will actually access the root directory of the new filesystem. So essentially the directory is told that it is a mountpoint and given a pointer to the root node of the new filesystem. We can actually reuse the *ptr* member of *fs_node_t* for this purpose (as it is currently only used for symlinks and they can never be mountpoints).



8.1.2. Implementation

8.1.2.1. fs.h

We first need to define the prototypes for our read/write/etc functions. The first four can be gained by looking at the [POSIX specification](#). The other two can just be made up :-)

```

typedef u32int (*read_type_t)(struct fs_node*, u32int, u32int, u8int*);

```

```
typedef u32int (*write_type_t)(struct fs_node*, u32int, u32int, u8int*);
typedef void (*open_type_t)(struct fs_node*);
typedef void (*close_type_t)(struct fs_node*);
typedef struct dirent * (*readdir_type_t)(struct fs_node*, u32int);
typedef struct fs_node * (*finddir_type_t)(struct fs_node*, char *name);
```

```
struct dirent // One of these is returned by the readdir call, according to POSIX.
{
    char name[128]; // Filename.
    u32int ino;      // Inode number. Required by POSIX.
};
```

We also need to define what the values in the `fs_node_t::flags` field mean:

```
#define FS_FILE      0x01
#define FS_DIRECTORY 0x02
#define FS_CHARDEVICE 0x03
#define FS_BLOCKDEVICE 0x04
#define FS_PIPE      0x05
#define FS_SYMLINK    0x06
#define FS_MOUNTPOINT 0x08 // Is the file an active mountpoint?
```

Notice that `FS_MOUNTPOINT` is given the value 8, not 7. This is so that it can be bitwise-OR'd in with `FS_DIRECTORY`. The other flags are given sequential values as they are mutually exclusive.

Lastly we need to define the root node of the filesystem and our read/write/etc functions.

```
extern fs_node_t *fs_root; // The root of the filesystem.

// Standard read/write/open/close functions. Note that these are all suffixed with
// _fs to distinguish them from the read/write/open/close which deal with file descriptors
// not file nodes.
u32int read_fs(fs_node_t *node, u32int offset, u32int size, u8int *buffer);
u32int write_fs(fs_node_t *node, u32int offset, u32int size, u8int *buffer);
void open_fs(fs_node_t *node, u8int read, u8int write);
void close_fs(fs_node_t *node);
struct dirent *readdir_fs(fs_node_t *node, u32int index);
fs_node_t *finddir_fs(fs_node_t *node, char *name);
```

8.1.2.2. fs.c

```
// fs.c -- Defines the interface for and structures relating to the virtual file system.
// Written for JamesM's kernel development tutorials.

#include "fs.h"

fs_node_t *fs_root = 0; // The root of the filesystem.

u32int read_fs(fs_node_t *node, u32int offset, u32int size, u8int *buffer)
{
    // Has the node got a read callback?
    if (node->read != 0)
        return node->read(node, offset, size, buffer);
    else
        return 0;
}
```

The above code should really be self-explanatory. If the node doesn't have a callback set, just return an error value. You should replicate the above code for `open()`, `close()` and `write()`. The same is true of `readdir()` and `finddir()`, although in those there should be an extra check: If the node is actually a directory!

```
if ((node->flags&0x7) == FS_DIRECTORY && node->readdir != 0 )
```

Believe it or not, that is all the code that is needed to make a simple virtual filesystem! With this code as a base we can make our initial ramdisk and maybe later more complex filesystems like FAT or ext2.

8.2. The initial ramdisk

An initial ramdisk is just a filesystem that is loaded into memory when the kernel boots. It is useful for storing drivers and configuration files that are needed before the kernel can access the root filesystem (indeed, it usually contains the driver to access that root filesystem!).

An *initrd*, as they are known, usually uses a proprietary filesystem format. The reason for this is that the most complex thing a filesystem has to handle, *deletion of files and reclamation of space*, isn't necessary. The kernel should try to get the root filesystem up and running as quick as possible - why would it want to delete files from the *initrd*??

As such you can just make a filesystem format up! I've made one for you as well, if you're not feeling very creative ;)

8.3. My own solution

My format does not support subdirectories. It stores the number of files in the system as the first 4 bytes of the *initrd* file. That is followed by a set number (64) of header structures, giving the names, offsets and sizes of the files contained. The actual file data follows. I have written a small C program to make this for me: it takes two arguments for each file to add: The path to the file from the current directory and the name to give the file in the generated filesystem.

8.3.1. Filesystem generator

```
#include <stdio.h>

struct initrd_header
{
    unsigned char magic; // The magic number is there to check for consistency.
    char name[64];
    unsigned int offset; // Offset in the initrd the file starts.
    unsigned int length; // Length of the file.
};

int main(char argc, char **argv)
{
    int nheaders = (argc-1)/2;
    struct initrd_header headers[64];
    printf("size of header: %d\n", sizeof(struct initrd_header));
    unsigned int off = sizeof(struct initrd_header) * 64 + sizeof(int);
    int i;
    for(i = 0; i < nheaders; i++)
    {
        printf("writing file %s->%s at 0x%x\n", argv[i*2+1], argv[i*2+2], off);
        strcpy(headers[i].name, argv[i*2+2]);
        headers[i].offset = off;
        FILE *stream = fopen(argv[i*2+1], "r");
        if(stream == 0)
        {
            printf("Error: file not found: %s\n", argv[i*2+1]);
            return 1;
        }
        fseek(stream, 0, SEEK_END);
        headers[i].length = ftell(stream);
        off += headers[i].length;
    }
}
```

```

        fclose(stream);
        headers[i].magic = 0xBF;
    }

    FILE *wstream = fopen("./initrd.img", "w");
    unsigned char *data = (unsigned char *)malloc(off);
    fwrite(&headers, sizeof(int), 1, wstream);
    fwrite(headers, sizeof(struct initrd_header), 64, wstream);

    for(i = 0; i < nheaders; i++)
    {
        FILE *stream = fopen(argv[i*2+1], "r");
        unsigned char *buf = (unsigned char *)malloc(headers[i].length);
        fread(buf, 1, headers[i].length, stream);
        fwrite(buf, 1, headers[i].length, wstream);
        fclose(stream);
        free(buf);
    }

    fclose(wstream);
    free(data);

    return 0;
}

```

I'm not going to explain the contents of this file: It is auxiliary and not important. Besides, you should be making your own anyway! ;)

8.3.2. Integrating it in to your own OS

Even if you are using a different file format to mine, this section may be useful in helping you integrate it into the kernel.

8.3.2.1. initrd.h

This file just defines the header structure types and gives a function prototype for the *initialise_initrd* function so the kernel can call it.

```

// initrd.h -- Defines the interface for and structures relating to the initial ramdisk.
// Written for JamesM's kernel development tutorials.

#ifndef INITRD_H
#define INITRD_H

#include "common.h"
#include "fs.h"

typedef struct
{
    u32int nfiles; // The number of files in the ramdisk.
} initrd_header_t;

typedef struct
{
    u8int magic; // Magic number, for error checking.
    s8int name[64]; // Filename.
    u32int offset; // Offset in the initrd that the file starts.
    u32int length; // Length of the file.
} initrd_file_header_t;

// Initialises the initial ramdisk. It gets passed the address of the multiboot module,
// and returns a completed filesystem node.
fs_node_t *initialise_initrd(u32int location);

```

```
#endif
```

8.3.2.2. initrd.c

The first thing we need is some static declarations:

```
// initrd.c -- Defines the interface for and structures relating to the initial ramdisk.
// Written for JamesM's kernel development tutorials.

#include "initrd.h"

initrd_header_t *initrd_header;    // The header.
initrd_file_header_t *file_headers; // The list of file headers.
fs_node_t *initrd_root;           // Our root directory node.
fs_node_t *initrd_dev;            // We also add a directory node for /dev, so we can mount devfs
later on.
fs_node_t *root_nodes;            // List of file nodes.
int nroot_nodes;                  // Number of file nodes.

struct dirent dirent;
```

The next thing we need is a function to read from a file in our initrd.

```
static u32int initrd_read(fs_node_t *node, u32int offset, u32int size, u8int *buffer)
{
    initrd_file_header_t header = file_headers[node->inode];
    if (offset > header.length)
        return 0;
    if (offset+size > header.length)
        size = header.length-offset;
    memcpy(buffer, (u8int*) (header.offset+offset), size);
    return size;
}
```

That function demonstrates one very annoying thing about writing low level code: 80% of it is error-checking. Unfortunately you can't get away from it - if you leave it out you will spend literally days trying to work out why your code doesn't work.

It would also be quite useful to have some working readdir and finddir functions:

```
static struct dirent *initrd_readdir(fs_node_t *node, u32int index)
{
    if (node == initrd_root && index == 0)
    {
        strcpy(dirent.name, "dev");
        dirent.name[3] = 0; // Make sure the string is NULL-terminated.
        dirent.ino = 0;
        return &dirent;
    }

    if (index-1 >= nroot_nodes)
        return 0;
    strcpy(dirent.name, root_nodes[index-1].name);
    dirent.name[strlen(root_nodes[index-1].name)] = 0; // Make sure the string is NULL-terminated.
    dirent.ino = root_nodes[index-1].inode;
    return &dirent;
}

static fs_node_t *initrd_finddir(fs_node_t *node, char *name)
{
    if (node == initrd_root &&
        !strcmp(name, "dev") )
```

```

    return initrd_dev;

int i;
for (i = 0; i < nroot_nodes; i++)
    if (!strcmp(name, root_nodes[i].name))
        return &root_nodes[i];
return 0;
}

```

Last but not least we need to initialise the filesystem:

```

fs_node_t *initialise_initrd(u32int location)
{
    // Initialise the main and file header pointers and populate the root directory.
    initrd_header = (initrd_header_t *)location;
    file_headers = (initrd_file_header_t *) (location+sizeof(initrd_header_t));
}

```

We assume that the kernel knows where our initrd starts and can convey that location to the initialise function.

```

// Initialise the root directory.
initrd_root = (fs_node_t*)kmalloc(sizeof(fs_node_t));
strcpy(initrd_root->name, "initrd");
initrd_root->mask = initrd_root->uid = initrd_root->gid = initrd_root->inode = initrd_root->length = 0;
initrd_root->flags = FS_DIRECTORY;
initrd_root->read = 0;
initrd_root->write = 0;
initrd_root->open = 0;
initrd_root->close = 0;
initrd_root->readdir = &initrd_readdir;
initrd_root->finddir = &initrd_finddir;
initrd_root->ptr = 0;
initrd_root->impl = 0;

```

Here we make the root directory node. We get some memory from the kernel heap and give the node a name. We really don't need to name this node as the root is never referenced by name, just '/'.

Most of the code initialises pointers to NULL (0), but you'll notice that the node is told it is a directory (flags = FS_DIRECTORY) and that it has both readdir and finddir functions.

The same is done for the /dev node:

```

// Initialise the /dev directory (required!)
initrd_dev = (fs_node_t*)kmalloc(sizeof(fs_node_t));
strcpy(initrd_dev->name, "dev");
initrd_dev->mask = initrd_dev->uid = initrd_dev->gid = initrd_dev->inode = initrd_dev->length = 0;
initrd_dev->flags = FS_DIRECTORY;
initrd_dev->read = 0;
initrd_dev->write = 0;
initrd_dev->open = 0;
initrd_dev->close = 0;
initrd_dev->readdir = &initrd_readdir;
initrd_dev->finddir = &initrd_finddir;
initrd_dev->ptr = 0;
initrd_dev->impl = 0;

```

Now that they're done we can start actually adding the files in the ramdisk. First we allocate space for them:

```

root_nodes = (fs_node_t*)kmalloc(sizeof(fs_node_t) * initrd_header->nfiles);
nroot_nodes = initrd_header->nfiles;

```

Then we make them:

```
// For every file...
int i;
for (i = 0; i < initrd_header->nfiles; i++)
{
    // Edit the file's header - currently it holds the file offset
    // relative to the start of the ramdisk. We want it relative to the start
    // of memory.
    file_headers[i].offset += location;
    // Create a new file node.
    strcpy(root_nodes[i].name, &file_headers[i].name);
    root_nodes[i].mask = root_nodes[i].uid = root_nodes[i].gid = 0;
    root_nodes[i].length = file_headers[i].length;
    root_nodes[i].inode = i;
    root_nodes[i].flags = FS_FILE;
    root_nodes[i].read = &initrd_read;
    root_nodes[i].write = 0;
    root_nodes[i].readdir = 0;
    root_nodes[i].finddir = 0;
    root_nodes[i].open = 0;
    root_nodes[i].close = 0;
    root_nodes[i].impl = 0;
}
```

And finally return the root node so the kernel can access us:

```
return initrd_root;
}
```

8.4. Loading the initrd as a multiboot module

Now we need to work out how to get our initrd loaded into memory in the first place. Luckily, the multiboot specification allows for 'modules' to be loaded. We can tell GRUB to load our initrd as a module. You can do this by mounting the floppy.img file as a loopback device, finding the /boot/grub/menu.lst file and adding a 'module (fd0)/initrd' line just below the 'kernel' line.

Alternatively you can download a new and improved image from [here](#).

GRUB communicates the location of this file to us via the multiboot information structure that we declared but never defined in the first tutorial. We have to define it now: This definition is lifted directly from the [Multiboot spec](#).

multiboot.h

```
#include "common.h"

#define MULTIBOOT_FLAG_MEM      0x001
#define MULTIBOOT_FLAG_DEVICE  0x002
#define MULTIBOOT_FLAG_CMDLINE 0x004
#define MULTIBOOT_FLAG_MODS    0x008
#define MULTIBOOT_FLAG_AOUT    0x010
#define MULTIBOOT_FLAG_ELF     0x020
#define MULTIBOOT_FLAG_MMAP    0x040
#define MULTIBOOT_FLAG_CONFIG  0x080
#define MULTIBOOT_FLAG_LOADER  0x100
#define MULTIBOOT_FLAG_APM     0x200
#define MULTIBOOT_FLAG_VBE     0x400

struct multiboot
{
    u32int flags;
```



```

    u32int mem_lower;
    u32int mem_upper;
    u32int boot_device;
    u32int cmdline;
    u32int mods_count;
    u32int mods_addr;
    u32int num;
    u32int size;
    u32int addr;
    u32int shndx;
    u32int mmap_length;
    u32int mmap_addr;
    u32int drives_length;
    u32int drives_addr;
    u32int config_table;
    u32int boot_loader_name;
    u32int apm_table;
    u32int vbe_control_info;
    u32int vbe_mode_info;
    u32int vbe_mode;
    u32int vbe_interface_seg;
    u32int vbe_interface_off;
    u32int vbe_interface_len;
} __attribute__((packed));

typedef struct multiboot_header multiboot_header_t;

```

The interesting fields are the *mods_addr* and *mods_count* fields. The *mods_count* field contains the number of modules loaded. We should check that this is > 0. The *mods_addr* field is an array of addresses: Each 'entry' consists of the starting address of the module and its end, each being 4 bytes.

As we are only expecting one module we can just treat the *mods_addr* field as a pointer and find whatever value lies there. That will be the location of our initrd. The value of the address 4 bytes on from that will be the end address. We can use this to change the memory management placement address so that memory allocations don't accidentally overwrite our ramdisk!

main.c

```

int main(struct multiboot *mboot_ptr)
{
    // Initialise all the ISRs and segmentation
    init_descriptor_tables();
    // Initialise the screen (by clearing it)
    monitor_clear();

    // Find the location of our initial ramdisk.
    ASSERT(mboot_ptr->mods_count > 0);
    u32int initrd_location = *((u32int*)mboot_ptr->mods_addr);
    u32int initrd_end = *((u32int*)(mboot_ptr->mods_addr+4));
    // Don't trample our module with placement accesses, please!
    placement_address = initrd_end;

    // Start paging.
    initialise_paging();

    // Initialise the initial ramdisk, and set it as the filesystem root.
    fs_root = initialise_initrd(initrd_location);
}

```


Success! That's one VFS and initrd cooked up in no time. Let's test it out.

8.5. Testing it out

Firstly let's
add some test
code to find
all files in '/'
and print their
contents:

main.c

```
// list the
contents of /
int i = 0;
```



```
Found file dev
      (directory)
Found file test.txt
      contents: "Hello, VFS world!"
Found file test2.txt
      contents: "My filename is test2.txt!"
_

CTRL + 3rd button enables mouse  A: NUM CAPS SCRL
```

Success!

```
struct dirent *node = 0;
while ( (node = readdir_fs(fs_root, i)) != 0)
{
    monitor_write("Found file ");
    monitor_write(node->name);
    fs_node_t *fsnode = finddir_fs(fs_root, node->name);

    if ((fsnode->flags&0x7) == FS_DIRECTORY)
        monitor_write("\n\t(directory)\n");
    else
```

```
{
    monitor_write("\n\t contents: \n");
    char buf[256];
    u32int sz = read_fs(fsnode, 0, 256, buf);
    int j;
    for (j = 0; j < sz; j++)
        monitor_put(buf[j]);

    monitor_write("\n\n");
}
i++;
}
```

Make a couple of test files, and build!

```
./make_initrd test.txt test.txt test2.txt test2.txt
cd src
make clean
make
cd ..
./update_image.sh
./run_bochs.sh
```

The code for this tutorial can be found [here](#).

Copyright James Molloy 2008 - james<at>jamesmolloy.co.uk