

CSC 262

Lab Project 3

Due: Oct. 14, 2008

1 The `ext2` filesystem

`ext2` was the official filesystem of Linux when it was still in its infancy (the early to mid nineties). Most Linux systems now use `ext3` or another advanced filesystem. `ext2` was phased out because it had reliability problems (if the power went out) and was inefficient when dealing with massive files or directories. However, `ext2` is perfect for teaching, because it's a production implementation of most of the filesystem concepts we've been discussing in class.

In this lab, you'll be asked to extend or implement functions in some provided C files. These files are available from the class website, under Labs.

2 USB sticks

The first part of this lab will be to create an `ext2` filesystem on the USB stick I gave you. Usually, USB devices are formatted with one of the less-wholesome FAT filesystems preferred by older versions of Windows. So, you're going to have to format the USB drive as an `ext2` device.

To Linux, your USB drive looks like a hard drive. When you insert it, chances are it will automatically be mounted and associated with a SCSI drive device (`/dev/sda1`, usually). So, you can treat your USB drive as a tiny `harddrive`. So, you can assume that the sectors on the drive are ordered and can therefore treat it as a sequence of addressable bytes. The two low-level commands you will be using in this lab are `seek`, which moves the read/write head to a location on the drive, and `read` which reads in a number of bytes from the location on the drive you just seeked to. Because `seek` is interested in moving around the read/write head of a drive, it is concerned with offsets. That is, the number of bytes from the beginning of the drive to the desired location. Some of the questions in this lab are all about translating from block number to byte offset, to get you used to this (sometimes tricky) scheme.

2.1 Creating an `ext2` filesystem

A new memory stick comes formatted for Windows. That means that it uses the `vfat` filesystem, which is simpler and less capable than `ext2`. Our first step will be to replace the `vfat` format with `ext2`. (Note: if for some reason the stick came without formatting, we could create a formatted partition on it using `fdisk`. If we were working with a floppy, we would format it first using `fdformat`.) Insert the memory stick into one of the USB ports on the left side of the monitor screen.

Creating the `ext2` file system is simple using the `mke2fs` command. However, we first have to make sure that the `vfat` drive is not mounted (connected to the file system directory tree). Modern Linux systems automatically detect insertion of a memory stick and mount it for you. We can see

the devices currently mounted using the mount command. The result should look somewhat like the following:

```
/dev/hda5 on / type ext3 (rw,noatime)
none on /proc type proc (rw)
none on /proc/bus/usb type usbfs (rw)
none on /sys type sysfs (rw)
/dev/hda7 on /boot type ext3 (rw,noatime)
/dev/hda1 on /mnt/windows type ntfs (ro,umask=0022,nls=iso8859-1)
```

The memory stick should be the first (and only) SCSI drive, referred to by Unix as `/dev/sda`. (Additional SCSI drives would be named `/dev/sdb`, `/dev/sdc`, etc. The floppy drive is `/dev/fd0`.) If you see a reference to `/dev/sda1`, that means that the memory stick has already been mounted, probably at `/media/disk`. To unmount it, use `umount`. You may then proceed with the ext2 format, using `mke2fs`. Finally, remount the device and change the permissions on it to be writable by anyone. (The items marked with a * need root privileges, so you should use the `sudo` command.) These steps are summarized below.

```
[lab3]$ sudo /sbin/mke2fs /dev/sdb1
[sudo] password:
mke2fs 1.40.8 (13-Mar-2008)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
31360 inodes, 125296 blocks
6264 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67371008
16 block groups
8192 blocks per group, 8192 fragments per group
1960 inodes per group
Superblock backups stored on blocks:
8193, 24577, 40961, 57345, 73729

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

This filesystem will be automatically checked every 23 mounts or 180 days, whichever comes first. Use `tune2fs -c` or `-i` to override.

`mount` and `umount` require root access, so you'll probably have to use `sudo` to invoke them. Note too that most of the low-level access to the raw device requires root access, so you'll probably have to run your code with `sudo`, too. But now, you have your very own teeny hard drive to mess with!

3 ext2 filesystem structure

The following picture illustrates the basic structure of the `ext2` filesystem:

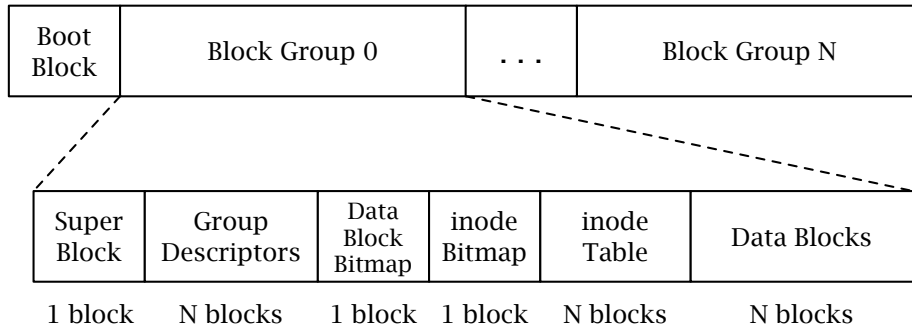


Figure 1: `ext2`, in pictures

The first 1024 bytes of a volume are the boot block. This area is reserved for partition boot sectors and aren't used by `ext2`. The rest of the disk is divided into block groups. Small devices, like floppies, contain only 1 block group. But your USB drive should have several (16 or so). All of the structures we'll be discussing are defined in the `linux/ext2.h` header file, which is available on your linux boxes under `/usr/include/linux/ext2.h`.

3.1 The Superblock

The superblock contains all the pertinent global information for the entire drive. Information such as: the total number of blocks on disk, the size of the blocks, the number of free blocks, and so forth. The structure of the superblock is described by `struct ext2_super_block` (which occurs around line 315 of the `ext2.h` header file). I've extracted some of the more pertinent fields below:

```
struct ext2_super_block {
    __le32  s_inodes_count;           /* Inodes count */
    __le32  s_blocks_count;          /* Blocks count */
    __le32  s_r_blocks_count;        /* Reserved blocks count */
    __le32  s_free_blocks_count;     /* Free blocks count */
    __le32  s_free_inodes_count;     /* Free inodes count */
    __le32  s_first_data_block;      /* First Data Block */
    __le32  s_log_block_size;        /* Block size */
    ...
    __le32  s_blocks_per_group;       /* # Blocks per group */
    ...
    __le32  s_inodes_per_group;       /* # Inodes per group */
```

```

        __le16  s_magic;                /* Magic signature */
...
        __le32  s_first_ino;           /* First non-reserved inode */
        __le16  s_inode_size;         /* size of inode structure */
...

```

The `__le32` and `__le16` datatypes specify little-endian 32-bit and 16-bit integers (unsigned). Because Linux is cross-platform, more expressive type names are used rather than just plain old `int`. But you can still cast these values to an `int` and use them like you'd expect. `s_inodes_count` and `s_blocks_count` contain the total number of inodes and blocks on the disk, respectively. As stated by `mke2fs`, the block size is 1024 bytes. If you multiply the block size by the number of blocks, you get the total disk capacity.

In the superblock, block size is contained in `s_log_block_size`. This value expresses block size as a power of two and using 1024 as the unit. This means that an `s_log_block_size` value of 0 means 1024 byte blocks, a value of 1 means 2048 byte blocks, etc. If you need to calculate the block size in bytes from this value, you should use the following code:

```
unsigned int block_size = 1024 << super.s_log_block_size;
```

The superblock also tells us how many blocks are in a block group. This value is contained in the `s_blocks_per_group` field. Going back to the `mke2fs` output, we see that the block group size is 8192. Now you can see that on a floppy (capacity 1.44 MB), only one block group would fit. But on our memory sticks, there's room for multiple blocks.

Q1: With 8192 blocks per group, how large would a block have to be for one block group to exactly fill your USB drive?

Q2: What would the `s_log_block_size` field be for the block size from Q1?

The superblock is located at offset 1024 on the drive (those first bytes are reserved for the boot block). Here's some C code to read in the superblock:

```

#include <linux/ext2_fs.h>

#define BASE_OFFSET 1024
#define USB_DEVICE "/dev/sda1"

. . .

int open_usb(struct ext2_super_block* super){
    int fd;
    struct ext2_group_desc* group = malloc(sizeof(struct ext2_group_desc));

    /* open USB device */

```

```

fd = open(USB_DEVICE, O_RDONLY); //opening the device for reading
if(fd < 0){ //some kind of error occurred
    perror(USB_DEVICE);
    exit(1); //we give up at this point
}

/* Now we read in Mr. Superblock */
/* seeking across the 'disk' to the superblock location */
lseek(fd, BASE_OFFSET, SEEK_SET);
/*actually reading in the bytes */
read(fd, super, sizeof(struct ext2_super_block));

/* Some sanity checks */
/* Make sure we're reading an EXT2 filesystem */
if(super->s_magic != EXT2_SUPER_MAGIC){
    fprintf(stderr, "Not an Ext2 filesystem!\n");
    exit(1);
}

block_size = 1024 << super->s_log_block_size;

return fd;
}

int main(void){
    struct ext2_super_block usb_block;
    int file_descriptor;

    file_descriptor = open_usb(&usb_block);

    return 0;
}

```

This code reads in the superblock to the struct pointed at by `super` and returns the file descriptor for the USB device.

Q3: Extend the code in the file `open_usb_super.c` to print out the superblock. Implement the `print_super_block` function to print out: the inodes count, blocks count, free blocks count, free inodes count, the first data block, block size, blocks per group, inodes per group, and the size of the inode structure. The output should be "`<field>\t:<value>\n`".

3.2 Group Descriptors

The blocks immediately after the superblock contain a list of group descriptors (essentially an array of group descriptors). This list contains a descriptor for each block group. For a disk with multiple block groups, we have to calculate the size of this list from information in the superblock, specifically `s_blocks_count` and `s_blocks_per_group`. We calculate it like so:

```
/* calculate number of block groups on the disk */
unsigned int group_count =
1 + (super.s_blocks_count-1) / super.s_blocks_per_group;

/* calculate size of the group descriptor list in bytes */
unsigned int descr_list_size =
group_count * sizeof(struct ext2_group_descr);
```

The group descriptor structure is called `ext2_group_descr` and is located around line 106 of the `ext2` header file. I've reproduced it below:

```
struct ext2_group_descr
{
    __le32  bg_block_bitmap;          /* Blocks bitmap block */
    __le32  bg_inode_bitmap;         /* Inodes bitmap block */
    __le32  bg_inode_table;          /* Inodes table block */
    __le16  bg_free_blocks_count;    /* Free blocks count */
    __le16  bg_free_inodes_count;    /* Free inodes count */
    __le16  bg_used_dirs_count;      /* Directories count */
    __le16  bg_pad;
    __le32  bg_reserved[3];
};
```

To read in the group descriptors, you first have to calculate the offset from the beginning of the disk. The first 1024 bytes are reserved and the first block is occupied by the superblock, so the code to read the first group descriptor off the disk looks like:

```
struct ext2_group_descr group_descr;

/* position head above the group descriptor block */
lseek(sd, 1024 + block_size, SEEK_SET);
read(sd, &group_descr, sizeof(group_descr));
```

The group descriptor tells us the location of the block/inode bitmaps and of the inode table (described later) through the `bg_block_bitmap`, `bg_inode_bitmap` and `bg_inode_table` fields. These values indicate the blocks where the bitmaps and the table are located. It is handy to have a function to convert a block number to an offset on disk, which can be easily done by knowing that all blocks on disk have the same size of `block_size` bytes (calculated earlier from the super-block):

```

/* location of the super-block in the first group */
#define BASE_OFFSET 1024

#define BLOCK_OFFSET(block) (BASE_OFFSET + (block-1)*block_size)

```

As you can see from the definition of `BLOCK_OFFSET`, blocks are numbered starting with 1. Block 1 is the superblock of the first group, block 2 are the group descriptors.

- Q4:** Extend the code contained in `open_usb_group.c` by implementing the `print_group_desc`. Print out: blocks bitmap block, inodes bitmap block, inodes table block, free blocks count, free inodes count, and the directories count. Your output should be formatted similarly to Q3.

3.3 Block and Inode Bitmaps

`ext2` keeps track of free space using bitmaps. Two bitmaps are used, actually. The blocks bitmap tracks free blocks (raw space), and the inode bitmap tracks free inodes. These bitmaps are fixed size and are exactly 1 block large. This puts an upper bound on the number of blocks per block group. With a block size of 1024, the block bitmap contains $1024 * 8 = 8192$ bits, which means that the block group can have no more than 8K blocks (which nicely matches the output from `mke2fs`). The following code reads the block bitmap from the disk:

```

struct ext2_super_block super; /* the super block */
struct ext2_group_desc group; /* the group descriptor */
unsigned char *bitmap;

/* ... [read superblock and group descriptor] ... */

bitmap = malloc(block_size); /* allocate memory for the bitmap */
lseek(sd, BLOCK_OFFSET(group->bg_block_bitmap), SEEK_SET);
read(sd, bitmap, block_size); /* read bitmap from disk */

...

free(bitmap);

```

- Q5:** Calculate the number of blocks in a block group given a block size of 4096.

- Q6:** Assume that the block bitmap has been corrupted, either due to a power failure, or a rogue program. Is it possible to reconstruct the bitmap from other metadata on disk? If so, how so? If not, why not?

3.4 The Inode Table

The inode table consists of a set of consecutive blocks, each containing a predefined number of inodes. These blocks basically serve as an array (or list) of all the inodes in the block group (in inode order). The block number of the first block of the inode table is stored in the `bg_inode_table` field in the group descriptor. Like other useful numbers, the number of blocks used by the inode table isn't directly available. It must be calculated by dividing the total number of inodes in the group by the number of inodes that you can fit into a block:

```
/* number of inodes per block */
unsigned int inodes_per_block = block_size / sizeof(struct ext2_inode);

/* size in blocks of the inode table */
unsigned int itable_blocks = super.s_inodes_per_group / inodes_per_block;
```

In the case of a floppy disk, there are 184 inodes per group and a block size of 1024 bytes. The size of an inode is 128 bytes, therefore the inode table will take $184 / (1024/128) = 23$ blocks.

Q7: How many blocks will the inode table take up on each block group of your 128MB memory stick? What percentage of the total storage is being used for just inodes?

The inode table contains everything the operating system needs to know about a file, including the type of file, permissions, owner, and, most important, where its data blocks are located on disk. It is no surprise therefore that this table needs to be accessed very frequently and its read access time should be minimized as much as possible. Reading an inode from disk every time it is needed is usually a very bad idea. However, in this context we will adopt this method to keep the example code as simple as possible. We provide a general function to read an inode from the inode table:

```
static void read_inode(
    int sd, /* the floppy disk file descriptor */
    const struct ext2_super_block *super, /* the super block for the disk */
    /* the block group to which the inode belongs */
    const struct ext2_group_desc *group,
    int inode_no, /* the inode number to read */
    struct ext2_inode *inode) /* where to put the inode */
{
    int group_no = inode_no/super->s_inodes_per_group;
    int inode_offs = inode_no - (group_no * num_inodes_per_group) - 1;
    lseek(sd,
        BLOCK_OFFSET(group[group_no].bg_inode_table)+
        inode_offs*sizeof(struct ext2_inode),
        SEEK_SET);
    read(sd, inode, sizeof(struct ext2_inode));
}
```

This code calculates the offset by adding the absolute offset of the inode table (the `BLOCK_OFFSET` part) with the distance of the desired inode from the start of the table (`inode_offs * sizeof(struct ext2_inode)`).

Q8: Assuming 1024 byte block size, and that the inode table starts at block 5, calculate the offset of inode 71 (in bytes). What block is that? (Assume an inode structure is 128 bytes large)

3.5 The Inode

Inodes, like blocks are numbered starting with 1. The structure of an inode is defined as `struct ext2_inode` at line 184 of the `ext2` header file. I've reproduced the salient fields below:

```
struct ext2_inode {
    __le16  i_mode;           /* File mode */
    __le16  i_uid;           /* Low 16 bits of Owner Uid */
    __le32  i_size;         /* Size in bytes */
    __le32  i_atime;        /* Access time */
    __le32  i_ctime;        /* Creation time */
    __le32  i_mtime;        /* Modification time */
    __le32  i_dtime;        /* Deletion Time */
    __le16  i_gid;           /* Low 16 bits of Group Id */
    __le16  i_links_count;  /* Links count */
    __le32  i_blocks;       /* Blocks count */
    __le32  i_flags;        /* File flags */
    ...
    __le32  i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    ...
}
```

`i_mode` is a 16-bit value that encodes the basic UNIX file permissions and file typing. For each file type, there is a macro defined in `sys/stat.h` to test the mode field. Macros can basically be used like functions, particularly if you're only passing in variables (rather than expressions).

Type	Macro
Regular File	<code>S_ISREG(m)</code>
Directory	<code>S_ISDIR(m)</code>
Character Device	<code>S_ISCHR(m)</code>
Block Device	<code>S_ISBLK(m)</code>
Fifo (Named Pipe)	<code>S_ISIFO(m)</code>
Socket	<code>S_ISSOCK(m)</code>
Symbolic Link	<code>S_ISLNK(m)</code>

The file permissions are also encoded in `i_mode`. Rather than have macros to test for permissions, a set of flags (also defined in `sys/stat.h`) can be bitwise ANDed with `i_mode` to test for permissions:

Domain	Read	Write	Exec	All
User	S_IRUSR	S_IWUSR	S_IXUSR	S_IRWXU
Group	S_IRGRP	S_IWGRP	S_IXGRP	S_IRWXG
All	S_IROTH	S_IWOTH	S_IXOTH	S_IRWXO

The **All** column is useful for testing if any of user, group, or other has permission. So, for instance if I wanted to check if I had permission to execute a file I owned I would write some C code like:

```
if(inode.i_mode & S_IXUSR) {
    //I can execute
}else{
    printf("Error: Insufficient permission\n");
}
```

Q9: Write some C code to check for the following permission string "rwxr-x—".

The `i_blocks` field is where the inode actually stores references to the data blocks of the file. `i_blocks` is actually an array of several references. For reasons described in lecture, simply having direct references severely limits file size. So, only the first 12 entries (0 through 11) in `i_blocks` directly reference data blocks. Entry 12 contains a reference to a single indirect block (a block of block pointers). Entry 13 contains a reference to a doubly indirect block (a block of pointers to singly indirect blocks). Entry 14 contains a reference to a triply indirect block (a block of pointers to doubly indirect blocks).

Q10: Given a block size of 2048 bytes, what is the maximum file size (in blocks and bytes) without using indirection? Using only single indirection? Double indirection? Triple indirection?

The following code prints the blocks used by the root directory of the disk (always located at inode 2):

```
struct ext2_inode inode;
struct ext2_group_desc group;

/* ... [ read superblock and group descriptor ] ... */

/* read root inode (#2) from the USB disk */
read_inode(sd, &super, &group, 2, &inode);

for (i=0; i<EXT2_N_BLOCKS; i++)
if (i < EXT2_NDIR_BLOCKS) /* direct blocks */
printf("Block %2u : %u\n", i, inode.i_block[i]);
else if (i == EXT2_IND_BLOCK) /* single indirect block */
printf("Single : %u\n", inode.i_block[i]);
```

```

else if (i == EXT2_DIND_BLOCK) /* double indirect block */
printf("Double   : %u\n", inode.i_block[i]);
else if (i == EXT2_TIND_BLOCK) /* triple indirect block */
printf("Triple   : %u\n", inode.i_block[i]);

```

Q11: Extend the code contained in `open_usb_inode.c` to print out the contents of an inode (input by the user). You should print out: `i_mode`, the file size, the creation time (as an integer), the blocks count, and the file flags (also as an integer).

3.6 Directories

Directories are a special case. They're more or less like normal files, except that they have the directory flag set in `i_mode` and the file contents are a list of files contained in the directory. Each entry in the directory is specified by `struct ext2_dir_entry_2` (line 497 in `ext2.h`), which I have reproduced below:

```

struct ext2_dir_entry_2 {
__le32 inode; /* Inode number */
__le16 rec_len; /* Directory entry length */
__u8 name_len; /* Name length */
__u8 file_type;
charname[EXT2_NAME_LEN]; /* File name */
};

```

`__u8` just means an unsigned byte (`unsigned char` in C). So each record contains the inode number of the file it references. Then the length of the record (because file names may have variable length, the record length needs to be specified). The length of the name, flags to specify the file type, and lastly the filename. Why is there both a record length and a name length? For efficiency reasons, it may be faster to have directory entries be a multiple of 4 bytes. Therefore, the filesystem may pad the filename with extra `\0` characters to get it up to a full multiple of 4. Additionally, if the filename is exactly the right length, the filesystem won't terminate it with a `\0` character. So a separate name length field is warranted. The `file_type` field indicates what kind of file the entry is pointing to. Possible values are:

file_type	Description
0	Unknown
1	Regular File
2	Directory
3	Character Device
4	Block Device
5	Named pipe
6	Socket
7	Symbolic Link

Here's a graphical representation of the directory entries for a directory that contains the sub-directories `Music` and `src`, and the file `test.txt`.

inode	rec_len	name_len		file-type				name			
13	12	1	2	.	\0	\0	\0				
2	12	2	2	.	.	\0	\0				
18	16	5	2	M	u	s	i	c	\0	\0	\0
15	16	8	1	t	e	s	t	.	t	x	t
19	12	3	2	s	r	c	\0				

Figure 2: A Sample Directory

When processing directories, one would walk a pointer over each record and processing each in turn (using `rec_len` to advance the pointer). For example, the following code reads the entries in a directory:

```

struct ext2_dir_entry_2 *entry;
unsigned int size;
unsigned char block[BLOCK_SIZE];

...

lseek(sd, BLOCK_OFFSET(inode->i_block[0]), SEEK_SET);
read(sd, block, block_size); /* read block from disk*/

size = 0; /* keep track of the bytes read */
/* first entry in the directory */
entry = (struct ext2_dir_entry_2 *) block;

while(size < inode->i_size) {
    char file_name[EXT2_NAME_LEN+1];
    /* copying the characters from the entry to a local string */
    memcpy(file_name, entry->name, entry->name_len);
    /* append null char to the file name */
    file_name[entry->name_len] = '\0';
    printf("%d %s\n", entry->inode, file_name);
}

```

```

    entry = (void*) entry + entry->rec_len; /* move to the next entry */
    size += entry->rec_len;
}

```

In this code, `block` is an in-memory copy of a disk block. Then we read the first directory data block into `block`. And then, for each entry the `name` field is copied (using the `memcpy` function) into a local string. Because we can't be sure that the filesystem is appending a null character, we have to add one ourselves. Then, we advance the `entry` pointer to the next record (which is what `entry = (void*) entry + entry->rec_len` does).

- Q12:** When a file is deleted, the directory entry's inode number is set to 0, and the `rec_len` field of the previous entry is incremented to 'jump over' the deleted field. Redraw the figure above after the user deletes the `src` directory.
- Q13:** Implement the `print_entry` function in the `open_usb_list.c` file. This function is handed a directory entry and needs to print it out (one per line). You should output "`<perm string>\t<file name>\n`" where `<perm string>` is the standard permission string from `ls -l`. e.g. something like "`dr-xrwx---`". Make sure that you remember to set the first character based upon the file type!

3.7 File names and locating a file

The directory tree creates a namespace for a file. Directories allow you to have two files named `foo`, but in separate directories. To locate a file given its absolute path, we must descend through the path, starting at the root directory, until we reach the file's parent directory. For example, to locate the file `/home/trekp/src/foo.c`, first you'd have to locate the root directory's inode, then open the data blocks to locate the entry for the directory named `home`. Using that entry, you would then open the inode for `/home`. From that inode you'd read in the data blocks to locate the directory named `trekp`, and so on until you'd located the inode for `/home/trekp/src` and read in the data to locate the entry for the file named `foo.c`. Because all such searches start at the root inode and may routinely traverse (and re-traverse) other directory inodes, it is common practice to cache directory inodes in memory (so that lookups really only go to disk if the directory has never been accessed before).

- Q14:** Extend the code from Question 13 to print out the contents of the whole disk (similarly to a recursive `ls`). Each file (and directory) should be printed with its permission string as in Question 12, except that directory names should be followed with the `'/'` character and the contents of a directory should be printed immediately after the entry for the directory itself.
- Q15:** Using inode and block numbers you come up with yourself, illustrate the inode, data block, and directory entry structure for a file with the absolute path `/usr/include/linux/ext2.h`. (You needn't fill out the data blocks (except for the directory entries), just draw diagrams similar to the pictures in this lab handout). How many `lseek` and `read` calls would be required to fetch this file?

Q16: (*Extra Credit*) Use the code from question 13 to implement a program that takes an absolute file name (a file name specified from the root directory) and locates it on disk and outputs its contents (byte for byte) to the console.