

## Assignment 5 Solutions

cpe 453 Fall 2024

*First things first – but not necessarily in that order.*

— The Doctor, “Doctor Who” (as quoted by `/usr/games/fortune`)

Due by 11:59:59pm, Wednesday, December 4th.

This assignment may be done with a partner.

### Programs: minls and minget

This assignment requires you to write two small programs to manipulate MINIX filesystem images. That is, these are for working with MINIX filesystems from *outside* MINIX. (After all, from inside MINIX you could just use the `sytem` to read the filesystem.) The programs are `minls` and `minget`, described below.

`minls` [ `-v` ] [ `-p part` [ `-s subpart` ] ] `imagefile` [ `path` ]  
Minls lists a file or directory on the given filesystem image. If the optional `path` argument is omitted it defaults to the root directory.

`minget` [ `-v` ] [ `-p part` [ `-s subpart` ] ] `imagefile srcpath` [ `dstpath` ]  
Minget copies a regular file from the given source path to the given destination path. If the destination path is omitted, it copies to stdout.

Both programs take the same options:

- `-p <part>` choose a primary partition on the image
- `-s <sub>` choose a subpartition
- `-v` verbose. Print partition table(s), superblock and inode of source file or directory to stderr

**Note:** Your program must support there being a space between `-p` or `-s` and its argument. You may make the space optional, of course.

If no partition or subpartition option is present, both programs default to treating the image as unpartitioned.

Paths that do not include a leading `/'` are processed relative to the root directory.

Each program must:

- check the disk image for valid partition table(s), if partitioning is requested,
- check for a valid MINIX superblock, and
- check that directories being listed really are directories and files being copied really are regular files.

### Useful Readings

§3.7.4

Hard disk driver in MINIX. Helpful, but, mercifully, not terribly important for this task.

§5.3

Filesystem implementation.

§5.6

The minix filesystem. Quite important.

`include/arch/i386/partition.h`

the partition table structure is here

`servers/mfs/super.h`

the superblock structure (Also Figure 5-35 in T&W)

others

many other useful things

## Endianness

Before any discussion of reading low-level data structures, we must discuss the implications of byte-order. With a single byte, the meaning of an address is clear, but with multi-byte data, such as integers, the question arises, “Which end of the data does the address really point to?” The two obvious possibilities are the most significant byte or the least significant byte. Each is quite valid, but unfortunately, they are incompatible.

Consider the number `0xAABBCCDD`. If represented as a little-endian number at address  $A$ , the least significant byte, `DD`, comes at location  $A$ , then the more significant bytes follow at locations  $A + 1$ ,  $A + 2$  and  $A + 3$ . For big-endian, the most significant byte, `AA` comes at address  $A$ , and the less significant bytes follow:

Little Endian					Big Endian				
Address	+0	+1	+2	+3	Address	+0	+1	+2	+3
	<code>DD</code>	<code>CC</code>	<code>BB</code>	<code>AA</code>		<code>AA</code>	<code>BB</code>	<code>CC</code>	<code>DD</code>

One can easily be mapped to the other by reversing the order of the bytes.

For this assignment, you **do not** have to support opposite-ordered filesystems, but it is important to know about them during development. If you try to test a program developed on a big-endian system with a filesystem written on a little-endian machine, it will not work, or, worse, if it does work, it will not work when tested on a machine of the same endianness.

Intel x86 machines are little-endian. SPARCs, like `sparc.csc.calpoly.edu`<sup>1</sup>, and non-Intel Macintosh processors are big-endian.

## Sizes of Data Types

Most C data types do not have fixed sizes. For example, in C, “`int`” is defined as being at least as big as a `short` and no bigger than a `long`. What if you really want to know? The header file `<stdint.h>` defines a set of fixed-width data types which are more portable than `ints` when you really need to know how big something is. These exist in signed and unsigned versions named `intXX_t` and `uintXX_t`, where `XX` is the number of bits. For example, `uint32_t size` makes `size` a 32-bit unsigned integer.

## Disk geometry

A disk drive is divided up into *sectors* which are, in turn, combined to form *blocks*. MINIX builds its filesystem out of multi-block units called *zones*, but we will come to that later. The sector is the minimum addressable unit of the disk drive and a particular sector can be described in one of two ways:

**CHS—Cylinder, Head, Sector** The physical geometry of the drive is used by specifying which cylinder, which head, and which sector of that track is to be read or written. This is precise, but not terribly intuitive. Sector numbering starts at 1. Head and cylinder numbering starts at 0.

**LBA—Linear Block Addressing** Some controllers allow drivers to ignore the real geometry and simply treat the disk as an array of sectors numbered  $0 \dots n$ . For our purposes—reading a linear file from a disk—this is clearly preferable.

---

<sup>1</sup>Now retired, but once upon a time...

If you have a disk drive with  $H$  heads and  $S$  sectors per track, you can convert CHS address  $(c, h, s)$  to LBA as follows:

$$\begin{aligned}\text{LBA} &= c \cdot HS + h \cdot S + s - 1 \\ &= (c \cdot H + h) \cdot S + s - 1\end{aligned}$$

Or, if your controller supports it, you can just use linear block addressing. Luckily, for this assignment, since our “disks” are file images we never need to do anything more than LBA.

The sizes used by MINIX are shown in Figure 1. Because it is critical to the interpretation of integers stored in the filesystem, note that filesystems used in this assignment will be little-endian.

Structure	Size
Sector size:	512B
Block size:	<i>in superblock</i>
Zone size:	$k \times \text{blocksize}$
Endianness:	little

Figure 1: Sizes used in the MINIX filesystem.

First, we have to look at what a filesystem looks like.

The first sector of every disk or disk partition is the *boot sector*. This sector contains the *master boot record (MBR)* and the *partition table*, if there is one.

## Partitions and Subpartitions

Any disk can have up to four *primary partitions*. The information for these partitions is stored in the *partition table*, located at address 0x1BE on the disk. The structure of a partition table entry is given in Figure 2. The fields we will be interested in are: **type**, because it says whether this is a MINIX partition, and **lFirst** and **size**. **lFirst** gives the first absolute<sup>2</sup> sector number of the partition and **lFirst** + **size** - 1 gives last.

Type	Name	Meaning
uint8_t	bootind	Boot magic number (0x80 if bootable)
uint8_t	start_head	Start of partition in CHS
uint8_t	start_sec	
uint8_t	start_cyl	
uint8_t	type	Type of partition (0x81 is MINIX)
uint8_t	end_head	End of partition in CHS
uint8_t	end_sec	
uint8_t	end_cyl	
uint32_t	lFirst	First sector (LBA addressing)
uint32_t	size	size of partition (in sectors)

Figure 2: Partition table entry

<sup>2</sup>That is, even for the subpartition table, the sector numbers are relative to the beginning of the disk, not the partition.

**Note**, for the CHS form of the partition description only the bottom 6 bits of the sector field are the sector. The top two bits of the sector field are prepended to the cylinder to form a 10-bit cylinder value.

A valid partition table contains a *signature*: 0x55 in byte 510, and 0xAA in byte 511. You must check the partition table for validity before proceeding.

Each partition is like a complete disk of its own and could include a (sub)partition table of its own, with the same structures at the same positions relative to the beginning of the containing partition. Once you have chased down the right partition, it's necessary to navigate the filesystem.

## Filesystems

As we've already discussed, the first 1K block of a filesystem contains the boot sector. This is followed by another 1K block containing the *superblock*. The superblock determines the geometry of the rest of the filesystem.

The superblock is drawn in Figure 5-35 in T&W. Note that the figure is two bytes wide. A C version of the superblock can be found in Figure 3.

```
struct superblock {           /* Minix Version 3 Superblock
                               * this structure found in fs/super.h
                               * in minix 3.1.1
                               */
    /* on disk. These fields and orientation are non-negotiable */
    uint32_t  ninodes;         /* number of inodes in this filesystem */
    uint16_t  pad1;            /* make things line up properly */
    int16_t   i_blocks;        /* # of blocks used by inode bit map */
    int16_t   z_blocks;        /* # of blocks used by zone bit map */
    uint16_t  firstdata;       /* number of first data zone */
    int16_t   log_zone_size;    /* log2 of blocks per zone */
    uint16_t  pad2;            /* make things line up again */
    uint32_t  max_file;        /* maximum file size */
    uint32_t  zones;           /* number of zones on disk */
    int16_t   magic;           /* magic number */
    int16_t   pad3;            /* make things line up again */
    uint16_t  blocksize;       /* block size in bytes */
    uint8_t   subversion;      /* filesystem sub-version */
}
```

Figure 3: A MINIX superblock as it exists on disk

The superblock contains a magic number that marks it as a minix filesystem.

## The MINIX filesystem

Files in minix are built out of *zones* which are multiples of blocks. The  $\log_2$  of the number of blocks per zone is given in the superblock. The size of a zone, then, can be calculated by a simple bit shift:

$$\text{zonesize} = \text{blocksize} \ll \log_2 \text{zonesize}$$

0x1BE	location of the partition table
0x81	partition type for MINIX
0x55	byte 510 of a boot sector with a valid partition table
0xAA	byte 511 of a boot sector with a valid partition table
0x4D5A	the minix magic number
0x5A4D	minix magic number on a byte-reversed filesystem
64	size of an inode in bytes
64	size of a directory entry in bytes

Table 1: Useful constants

Blocks are, in turn, built out of sectors. In version 3 of the MINIX filesystem, the block size is defined in the superblock.

The filesystem is divided into six regions, shown in Table 2.

## Files and Directories

**Files** Files are a collection of zones indexed by an inode. (See T&W§5.6.4.) The minix inode structure is drawn in Figure 5-36 in T&W. A C version is included here in Figure 4.

```
#define DIRECT_ZONES 7

struct inode {
    uint16_t mode;           /* mode */
    uint16_t links;          /* number or links */
    uint16_t uid;
    uint16_t gid;
    uint32_t size;
    int32_t atime;
    int32_t mtime;
    int32_t ctime;
    uint32_t zone[DIRECT_ZONES];
    uint32_t indirect;
    uint32_t two_indirect;
    uint32_t unused;
};
```

Figure 4: A MINIX inode as it exists on disk

All directories are linked into a tree starting at the root directory at inode 1.

**Directories** Directories are just files consisting of directory entries. A directory entry (Fig. 5) is a uint32\_t holding the inode number followed by a 60 character array holding the filename. If the filename is less than the size of the buffer, it is null-terminated. If it occupies the whole buffer (is 60 characters long), it is not null-terminated.

The total size is given in the inode. A directory entry with an inode of 0 is a file marked as deleted. It is not a valid entry.

Blocks(s)		Contents	Description
Start	Number		
0	1	boot block	First sector contains boot loader and partition table, if any
1*	1	super block	determines the geometry of the other components. *Even though block 1 is reserved for it, the superblock is always found at offset 1024, regardless of the filesystem's block size.
2	$B_{\text{imap}}$	inode bitmap	a bitmap indicating which inodes are free. The number of blocks ( $B_{\text{imap}}$ ) used by the inode bitmap is contained in the superblock.
$2 + B_{\text{imap}}$	$B_{\text{zmap}}$	zone bitmap	a bitmap indicating which data zones are free. The number of blocks used by the zone bitmap ( $B_{\text{zmap}}$ ) is contained in the superblock.
$2 + B_{\text{imap}} + B_{\text{zmap}}$	$B_{\text{inodes}}$	inodes	a series of blocks containing the inodes themselves. inodes are numbered starting at one. There is no inode zero. The number of blocks needed ( $B_{\text{inodes}}$ ) is the number of inodes times 64 divided by the size of a block.
(see note)		data zones	The actual data zones are allocated last. Zones are numbered starting at zero from the beginning of the filesystem. The number of the first data zone is included as part of the superblock. The first block number of a zone can be determined by multiplying the zone number by the number of blocks per zone.

Table 2: Components of a MINIX filesystem

Type	Name	Meaning
uint32_t	inode	inode number
unsigned char	name[60]	filename string

Figure 5: A MINIX directory entry

**File Types** File types can be determined by taking the bitwise and of the inode's mode field with the mask and comparing it with the masks for MINIX given in Table 3.

Mask	Description
0170000	File type mask
0100000	Regular file
0040000	Directory
0000400	Owner read permission
0000200	Owner write permission
0000100	Owner execute permission
0000040	Group read permission
0000020	Group write permission
0000010	Group execute permission
0000004	Other read permission
0000002	Other write permission
0000001	Other execute permission

Note that these constants are in octal

Table 3: Minix file mode bitmaks

#### Note:

- Zero is an invalid inode number. This marks an entry as having been deleted.
- Zone 0 is also special. Zone 0 can never be part of a file. If 0 appears as a zone of a file, it means that the entire zone referred to is to be treated as all zeros. This is how holes are implemented in files.
- Finally, indirect and double indirect zones only use the first block of that zone. This seems slightly inaesthetic, but nobody asked me and it mostly doesn't limit the filesystem.

#### Output

The output for `miniget` is fairly self-explanatory. Listings generated by `minls` should be in the following format.

- For a file:

`Minls` should print the file's permission string (described below) followed by a space, then the file size, right-justified in a field nine characters wide, followed by the pathname. The three fields are separated by spaces.

For example:

```
% minls -p 0 -s 0 HardDisk /minix/2.0.3
-rw-r--r--      130048 /minix/2.0.3
%
```

The permissions string consists of 10 characters. The first gives the files type: 'd' for a directory, or '-' for any other type of file. The remaining nine characters indicate the presence or absence of read (r), write (w) or execute (x) permission for the file's owner, group, and other respectively. If a permission is not granted, write a dash (-).

- For a directory:

print the path of the directory, followed by a colon, then list all the files in the directory, in the order they are found in the directory, as described above.

Example:

```
% minls -p 0 -s 0 HardDisk /minix
/minix:
drwxr-xr-x      64 .
drwxr-xr-x     320 ..
-rw-r--r--   130048 2.0.3
-rw-r--r--   152064 2.0.3r22
%
```

## Pitfalls

Watch out for:

**endianness** Not an issue if you develop on a PC, but the filesystem you'll be tested on will be little-endian on a little-endian architecture.

**compiler padding** Be *sure* that your data structures line up right if you're overlaying structures on the file.

**numbering** Inode numbers start at one.

Zone and block numbering starts at zero, but zero is *not* a valid zone number to be contained in a file.

**holes** These may exist in a file. If any file zone has the zone number zero, it means that the corresponding zone is to be treated as if it is all zeros.

**definitions** Remember, this is a MINIX filesystem being read, not a Linux, Solaris, or OSX one. File type and permission masks may or may not be the same as those on the system where you are compiling the program.

## Tricks and Tools

This has pretty much been covered above. Some (potentially) useful functions are listed in Table 4. Also, you don't have to worry about it much on an x86 system, but the compiler is allowed to add padding to structs. If you do not want it to add padding you can use the gcc **attribute** modifier to forbid that (in return for potentially slower code) like so:

```
struct __attribute__((__packed__)) thing {
    uint8_t byte;    /* a byte */
    uint32_t word;   /* a 4-byte int */
};
```

## Coding Standards and Make

See the pages on coding standards and make on the cpe 453 class web page.



<code>void *memcpy(3)</code>	copies $n$ bytes from memory area <code>src</code> to memory area <code>dest</code> . The memory areas may not overlap.
<code>void *memmove(3)</code>	copies $n$ bytes from memory area <code>src</code> to memory area <code>dest</code> . The memory areas may overlap.
<code>void *memset(3)</code>	Sets a region of memory to a given value.
<code>int fseek(3)</code> <code>long ftell(3)</code> <code>void rewind(3)</code> <code>int fgetpos(3)</code> <code>int fsetpos(3)</code>	File pointer positioning functions
<code>fread(3)</code> <code>fwrite(3)</code>	Stdio analogues of <code>read(2)</code> and <code>write(2)</code>
<code>strncpy(3)</code> <code>strncmp(3)</code>	Functions for manipulating limited-length strings
<code>ctime(3)</code>	parse a time into a string
<code>getopt(3)</code>	very helpful for parsing options

Table 4: Some potentially useful system calls and library functions

## What to turn in

Submit via `handin` in the CSL to the `asgn5` directory of the `pn-cs453` account:

- your well-documented source files.
- A makefile (called `Makefile`) that will build both programs with “`make all`”.
- A README file that contains:
  - Your name.
  - Any special instructions for running your program.
  - Any other thing you want me to know while I am grading it.

The README file should be **plain text**, i.e, **not a Word document**, and should be named “README”, all capitals with no extension.

## Sample runs

Below are some sample runs of `minls` and `minget`. I will also place some sample filesystems on the CSL in `~pn-cs453/Given/Asgn5` for your testing pleasure. Executable versions are in `~pn-cs453/demos`.

```
% minls
usage: minls  [ -v ] [ -p num [ -s num ] ] imagefile [ path ]
Options:
-p part    --- select partition for filesystem (default: none)
-s sub     --- select subpartition for filesystem (default: none)
-h help    --- print usage information and exit
-v verbose --- increase verbosity level
```

```
% minls TestImage
/:
drwxrwxrwx      384 .
drwxrwxrwx      384 ..
-rw-r--r--    73991 Other
drwxr-xr-x      3200 src
-rw-r--r--       11 Hello
% minls -v TestImage
```

#### Superblock Contents:

##### Stored Fields:

```
ninodes      768
i_blocks      1
z_blocks      1
firstdata     16
log_zone_size 0 (zone size: 4096)
max_file    4294967295
magic        0x4d5a
zones        360
blocksize    4096
subversion    0
```

##### File inode:

```
uint16_t mode      0x41ff (drwxrwxrwx)
uint16_t links      3
uint16_t uid        2
uint16_t gid        2
uint32_t size       384
uint32_t atime      1141098157 --- Mon Feb 27 19:42:37 2006
uint32_t mtime      1141098157 --- Mon Feb 27 19:42:37 2006
uint32_t ctime      1141098157 --- Mon Feb 27 19:42:37 2006
```

##### Direct zones:

```
zone[0] = 16
zone[1] = 0
zone[2] = 0
zone[3] = 0
zone[4] = 0
zone[5] = 0
zone[6] = 0
uint32_t indirect 0
uint32_t double    0
```

```
/:
drwxrwxrwx      384 .
drwxrwxrwx      384 ..
-rw-r--r--    73991 Other
drwxr-xr-x      3200 src
-rw-r--r--       11 Hello
```

```

% minls HardDisk
Bad magic number. (0x0000)
This doesn't look like a MINIX filesystem.
% minls -p 0 -s 2 HardDisk
/:
drwxrwxrwx      1280 .
drwxrwxrwx      1280 ..
drwxr-xr-x       512 adm
drwxr-xr-x       512 ast
drwxr-xr-x     20800 bin
drwxr-xr-x       384 etc
drwxr-xr-x       640 gnu
drwxr-xr-x     3392 include
drwxr-xr-x     2112 lib
drwxr-xr-x       704 log
drwxr-xr-x       896 man
drwxr-xr-x       384 mdec
drwx-----      128 preserve
drwxr-xr-x       192 run
drwxr-xr-x     1088/sbin
drwxr-xr-x       384 spool
drwxrwxrwx       128 tmp
drwxr-xr-x       896 src
drwxr-xr-x       192 home
% minls -p 0 -s 2 HardDisk /home/pnico
/home/pnico:
drwxr-xr-x       576 .
drwxr-xr-x       192 ..
-rw-r--r--       577 .ashrc
-rw-r--r--       300 .ellepro.b1
-rw-r--r--     5979 .ellepro.e
-rw-r--r--        44 .exrc
-rw-r--r--       304 .profile
-rw-r--r--     2654 .vimrc
-rw-r--r--        72 Message
% minls -p 0 -s 2 HardDisk /home/pnico/Message
-rw-r--r--        72 /home/pnico/Message
% minget -p 0 -s 2 HardDisk /home/pnico/Message
Hello.

```

If you can read this, you're getting somewhere.

Happy hacking.

%

**Solution:**

File	Where
Makefile	p.13
diskstuff.h	p.14
diskstuff.c	p.15
file.h	p.19
file.c	p.20
filesystem.h	p.21
filesystem.c	p.23
main.c	p.29
minls.c	p.32
options.h	p.35
util.h	p.36
util.c	p.37

## Makefile

---

```
CC      = gcc
CFLAGS  = -Wall -g -ansi -pedantic
LD       = gcc
LIBS     =
LDFLAGS = $(LIBS) -g
PROGS    = minget minls
COMMON   = diskstuff.o filesystem.o util.o file.o
LSOBSJS  = minls.o $(COMMON)
GETOBSJS = main.o $(COMMON)
HDRS     =
EXTRACLEAN = minls minget
.phony:
all:     $(PROGS)
minget:  $(GETOBSJS)
         $(LD) $(LDFLAGS) -o $@ $^
minls:   $(LSOBSJS)
         $(LD) $(LDFLAGS) -o $@ $^
allclean: clean
         @rm -f $(EXTRACLEAN)
clean:
         rm -f $(LSOBSJS) $(GETOBSJS) *~ TAGS
tags :
         etags *c *h
depend:
         makedepend -Y *c
test:    minls
         #      minls TestImage two three
         minls -v -p0 -s2 HardDisk two

# DO NOT DELETE

diskstuff.o: diskstuff.h options.h
file.o: file.h filesystem.h diskstuff.h options.h util.h
filesystem.o: filesystem.h diskstuff.h options.h util.h file.h
main.o: diskstuff.h options.h filesystem.h
minls.o: diskstuff.h options.h filesystem.h
```

---

```

#ifndef DISKSTUFF
#define DISKSTUFF

#include <stdio.h>
#include <stdint.h>
#include "options.h"

/* structures */
struct disk {
    long fsbase;           /* offset of beginning of filesystem */
    long fssize;           /* length of filesystem (in bytes) */
    FILE *fp;             /* file pointer for disk image */
};

struct __attribute__((packed)) partition {
    /* see include/ibm/partition.h */
    uint8_t bootind;
    uint8_t start_head;    /* start head */
    uint8_t start_sec;     /* start sector */
    uint8_t start_cyl;     /* start cylinder */
    uint8_t type;
    uint8_t end_head;      /* end head */
    uint8_t end_sec;       /* end sector */
    uint8_t end_cyl;       /* end cylinder */
    uint32_t lfirst;       /* logical first sector */
    uint32_t size;        /* size of partition (in sectors) */
};

/* Constants */

#define SECTORSIZE 512
#define ONEK 1024

#define PTABLE_OFFSET 0x1BE
#define PMAGIC510 0x55
#define PMAGIC511 0xAA
#define MINIXPART 0x81

#define NUMPARTS 4

#define NOTSET -1

/* prototypes */

void *read_sector(struct disk *disk, int sector, void *buffer);
void *write_sector(struct disk *disk, int sector, void *buffer);
void *read_block(struct disk *disk, int block, int blocksize, void *buffer);
void *write_block(struct disk *disk, int block, int blocksize, void *buffer);
struct disk *open_disk(char *filename, char *how, struct options *opts,
                      struct disk *dp);
void close_disk(struct disk *dp);
#endif

```

```
#include <stdio.h>
#include <string.h>
#include "diskstuff.h"

void *read_sector(struct disk *disk, int sector, void *buffer){
    /* read the given sector from the given disk image into the
     * given buffer.
     * returns the pointer on success, NULL on failure
     */
    void *rval;
    long offset;

    if ( !disk->fp || !buffer ) {
        rval=NULL;
    } else {
        offset = disk->fsbase + sector*SECTOR_SIZE;
        if ( -1 == fseek(disk->fp,offset,SEEK_SET) ) {
            rval=NULL;
        } else {
            if ( 1 != fread(buffer,SECTOR_SIZE,1,disk->fp) ) {
                rval=NULL;
            } else {
                rval=buffer;
            }
        }
    }
    return rval;
}

void *write_sector(struct disk *disk, int sector, void *buffer){
    /* write the given buffer to the given sector from the given disk image
     * returns the pointer on success, NULL on failure
     */
    void *rval;
    long offset;

    if ( !disk->fp || !buffer ) {
        rval=NULL;
    } else {
        offset = disk->fsbase + sector*SECTOR_SIZE;
        if ( -1 == fseek(disk->fp,offset,SEEK_SET) ) {
            rval=NULL;
        } else {
            if ( 1 != fwrite(buffer,SECTOR_SIZE,1,disk->fp) ) {
                rval=NULL;
            } else {
                rval=buffer;
            }
        }
    }
    return rval;
}

void *read_block(struct disk *disk, int block, int blocksize, void *buffer){
    /* read the given block from the given disk image into the
     * given buffer.
     * returns the pointer on success, NULL on failure
     */
    void *rval;
    long offset;
    int res;

    if ( !disk->fp || !buffer ) {
        rval=NULL;
    } else {
        offset = disk->fsbase + block*blocksize;
        if ( -1 == fseek(disk->fp,offset,SEEK_SET) ) {
            rval=NULL;
        } else {
            res = fread(buffer,blocksize,1,disk->fp);
            if ( 1 != res ) {
```

```

    rval=NULL;
  } else {
    rval=buffer;
  }
}
return rval;
}
80

void *write_block(struct disk *disk, int block, int blocksize, void *buffer){
  /* write the given buffer to the given block from the given disk image
   * returns the pointer on success, NULL on failure
   */
  void *rval;
  long offset;

  if ( !disk->fp || !buffer ) {
    rval=NULL;
  } else {
    offset = disk->fsbase + block*blocksize;
    if ( -1 == fseek(disk->fp,offset,SEEK_SET) ) {
      rval=NULL;
    } else {
      if ( 1 != fwrite(buffer,blocksize,1,disk->fp) ) {
        rval=NULL;
      } else {
        rval=buffer;
      }
    }
  }
  return rval;
}
90
100

/* translate cylinder and sector fields into real numbers */
#define cyl(c,s) ( (((s)&0xc0)<<2) | (c))
#define sec(c,s) ( (s) & 0x3f )

110

void print_ptable(FILE *where, struct partition ptable[],
                  struct options *opts){
  int i;

  fprintf(stderr, "      ----Start----      -----End-----\n");
  fprintf(stderr,
    "      Boot head sec cyl Type head sec cyl      First      Size\n");
  for(i=0;i<NUMPARTS;i++){
    fprintf(stderr,
      "      0x%02x %4u %4u %4u 0x%02x %4u %4u %4u %10u %10u\n",
      ptable[i].bootind,
      ptable[i].start_head,

      sec( ptable[i].start_cyl, ptable[i].start_sec),
      cyl( ptable[i].start_cyl, ptable[i].start_sec),

      ptable[i].type,

      ptable[i].end_head,
      sec( ptable[i].end_cyl, ptable[i].end_sec),
      cyl( ptable[i].end_cyl, ptable[i].end_sec),

      ptable[i].lFirst,
      ptable[i].size
    );
  }
}
120
130
140

struct partition *read_ptable(struct disk *dp, struct options *opts,
                             struct partition *ptable) {
  /* the partition table exists in the first 1-k block
   * of the disk (or partition, if we're doing a subpartition)

```



```

    * returns the table on success, NULL on failure
    */
    unsigned char buffer[ONEK];

    if ( !read_sector(dp, 0, buffer) )
        return NULL;
    if ( (buffer[510] != PMAGIC510) || (buffer[511] != PMAGIC511) ){
        fprintf(stderr,"Invalid partition table.\n");
        return NULL;
    }
    return memcpy(ptable,buffer+PTABLE_OFFSET,
        NUMPARTS*sizeof(struct partition));
}

static long filesize(char *filename) {
    FILE *fp;
    long res;
    fp=fopen(filename,"r");
    if ( !fp ) {
        res = 0;
    } else {
        if ( -1 == fseek(fp,0,SEEK_END) )
            res = 0;
        else {
            res = ftell(fp);
            if ( res == -1 )
                res = 0;
            fclose(fp);
        }
    }
    return res;
}

struct disk *open_disk(char *filename, char *how, struct options *opts,
    struct disk *dp){
    /* open the given file as a disk image for use by the rest of the
    program.
    * open_disk reads the partition table and, if successful
    * returns an open FILE* to the disk image and sets the
    * offset of its zeroth block.
    * Returns the pointer dp on success, NULL on failure.
    */
    struct partition ptable[NUMPARTS];

    if ( NULL == (dp->fp = fopen(filename,how)))
        return NULL;

    dp->fsbase = 0;
    dp->fssize = filesize(filename);

    /* the file's open. Let's find our (sub)partition */
    if ( opts->part != NOTSET ) {
        if ( !read_ptable(dp,opts,ptable) ) {
            close_disk(dp);
            return NULL;
        }

        if (verbose) {
            fprintf(stderr,"\nPartition table:\n");
            print_ptable(stderr,ptable,opts);
        }

        if ( ptable[opts->part].type == MINIXPART ) {
            dp->fsbase = ptable[opts->part].lFirst * SECTORSIZE;
            dp->fssize = ptable[opts->part].size * SECTORSIZE;
        } else {
            fprintf(stderr,"Not a Minix partition.\n");
            close_disk(dp);
            return NULL;
        }
    }
}

```

```
/* is there a subpartition?  Get it */
if ( opts->subpart != NOTSET ) {
    if ( !read_ptable(dp,opts,ptable) ) {
        close_disk(dp);
        return NULL;
    }

    if (verbose) {
        fprintf(stderr, "\nSubpartition table:\n");
        print_ptable(stderr,ptable,opts);
    }

    if ( ptable[opts->subpart].type == MINIXPART ) {
        dp->fsbase = ptable[opts->subpart].lFirst * SECTORSIZE;
        dp->fssize = ptable[opts->subpart].size * SECTORSIZE;
    } else {
        fprintf(stderr, "Not a Minix subpartition.\n");
        close_disk(dp);
        return NULL;
    }
}
}
return dp;
}

void close_disk(struct disk *dp){
    /* close anything that's open surrounding this disk
    */
    if ( dp && dp->fp )
        fclose(dp->fp);
}
```

220

230

240

```
#ifndef FILEH
#define FILEH
#include "filesystem.h"

typedef struct file {
    superblock *sb;    /* the filesystem */
    struct inode *ino; /* the inode */
} minfile;

minfile *new_minfile(superblock *sb, struct inode *ino);
void free_minfile(minfile *mf);
int file_zone_to_zone(minfile *file, int znum);

#endif
```

```

#include <stdio.h>
#include <stdlib.h>
#include "file.h"
#include "util.h"

minfile *new_minfile(superblock *sb, struct inode *ino) {
    /* allocate and return a new minfile structure */
    minfile *mf;
    mf = safe_malloc(sizeof (struct file));
    mf->sb = sb;
    mf->ino = ino;
    return mf;
}

void free_minfile(minfile *mf) {
    /* free the given minfile structure */
    free(mf);
}

int file_zone_to_zone(minfile *file, int znum) {
    /* translate the nth zone of a file to an absolute zone number */
    /*
     * static for some efficiency boost */
    static uint32_t *indzone; /* for reading out indirect zones */
    static uint32_t *dindzone; /* for reading double indirect zones */
    int i_idx, b_idx;
    int res = -1;

    if ( !indzone || !dindzone ) {
        indzone = safe_malloc(file->sb->sb.zonesize);
        dindzone = safe_malloc(file->sb->sb.zonesize);
    }

    if ( znum < DIRECT_ZONES ) { /* its a direct zone */
        res = file->ino->zone[znum];
    } else {
        znum -= DIRECT_ZONES;
        if ( znum < file->sb->sb.ptrs_per_zone ) { /* in the indirect block */
            read_zone(file->sb, file->ino->indirect, (void*)indzone);
            res = indzone[znum];
        } else { /* it's double indirect */
            znum -= file->sb->sb.ptrs_per_zone;
            /* first get the double block, then the real one */
            read_zone(file->sb, file->ino->two_indirect, (void*)dindzone);
            /* now, where are we in this thing? */
            i_idx = znum / file->sb->sb.ptrs_per_zone; /* index of indirect block */
            b_idx = znum % file->sb->sb.ptrs_per_zone; /* index of block */
            read_zone(file->sb, dindzone[i_idx], (void*)indzone);
            res = indzone[b_idx];
        }
    }
    return res;
}

```

```

#ifndef FILESYSTEMH
#define FILESYSTEMH

#include <stdio.h>
#include <stdint.h>
#include "diskstuff.h"

/* types */
struct __attribute__((packed)) superblock {
    /* this structure found in fs/super.h
    * in minix 2.0.3
    */
    /* on disk. These fields and orientation are non-negotiable */
    uint32_t ninodes; /* number of inodes in this filesystem */
    uint16_t pad1; /* make things line up properly */
    int16_t i_blocks; /* # of blocks used by inode bit map */
    int16_t z_blocks; /* # of blocks used by zone bit map */
    uint16_t firstdata; /* number of first data zone */
    int16_t log_zone_size; /* log2 of blocks per zone */
    int16_t pad2; /* make things line up again */
    uint32_t max_file; /* maximum file size */
    uint32_t zones; /* number of zones on disk */
    int16_t magic; /* magic number */
    int16_t pad3; /* make things line up again */
    uint16_t blocksize; /* block size in bytes */
    unsigned char subversion; /* filesystem sub-version */

    /* computed. These can be anything we want. */
    struct disk *diskinfo; /* a pointer to the disk structure */
    uint16_t firstlblock; /* number of first block with inodes */
    unsigned int zonesize; /* used all over the place */
    uint32_t ptrs_per_zone; /* number of zone pointers storable in
    * a zone */
    uint32_t ino_per_block; /* number of inodes storable in a block */
    int wrongended; /* set to true if we're dealing with
    * a backwards-ended filesystem.
    */
};

typedef union {
    unsigned char buffer[ONEK];
    struct superblock sb;
} superblock;

#define DIRECT_ZONES 7

struct __attribute__((packed)) inode {
    uint16_t mode; /* mode */
    uint16_t links; /* number or links */
    uint16_t uid;
    uint16_t gid;
    uint32_t size;
    int32_t atime;
    int32_t mtime;
    int32_t ctime;
    uint32_t zone[DIRECT_ZONES];
    uint32_t indirect;
    uint32_t two_indirect;
    uint32_t unused;
};

#ifndef DIRSIZ
#define DIRSIZ 60
#endif

struct __attribute__((packed)) fileent {
    uint32_t ino;
    char name[DIRSIZ];
};

/* constants (for v3) */

```

```
#define MIN_MAGIC    0x4d5a    /* the minix magic number */
#define MIN_MAGIC_REV 0x5a4d    /* the minix magic number reversed
                                * we have an endian problem */
```

```
/* prototypes */
int  read_superblock(struct disk *disk, superblock *sb);
void print_superblock(FILE *where, superblock *sb);
void *read_zone(superblock *sb, int zone, char *buffer);
void *static_read_zone(superblock *sb, int zone);
void *write_zone(superblock *sb, int zone, char *buffer);
struct inode *read_inode(superblock *sb, int inumber, struct inode *inode);
void print_inode(FILE *where, struct inode *inode);
int copy_file(superblock *sb, int inumber, FILE *where);
int path_to_inode(superblock *sb, char *path);
void listdir(superblock *sb, int inumber, FILE *where);
void listfile(superblock *sb, int inum, char *name, int limit, FILE *where);
```

80

```
/* macros */
#define MIN_ISREG(m) (((m)&0170000)==0100000)
#define MIN_ISDIR(m) (((m)&0170000)==0040000)
#define NOLIMIT 0
```

90

```
#endif
```

```

#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include "filesystem.h"
#include "util.h"
#include "file.h"

#define min(a,b) (((a)<(b))?(a):(b))

int read_superblock(struct disk *disk, superblock *sb) {
    /* read the superblock of the filesystem and verify that
     * this looks like a superblock for a minix filesystem.
     * returns true on success, false on failure.
     */
    int res;

    res=1;
    sb->sb.wrongended=1;      /* hope for the best */

    if ( NULL==read_block(disk,1,ONEK,sb) ) {
        res = 0;
    } else if ( ( sb->sb.magic != MIN_MAGIC ) &&
                ( sb->sb.magic != MIN_MAGIC_REV ) ) {
        /* check the magic number */
        fprintf(stderr,"Bad magic number.  (0x%04x)\n",sb->sb.magic);
        res = 0;
    }

    if ( res ) {
        /* success.  Do computed fields */
        /* first sort out endianness */
        if ( sb->sb.magic == MIN_MAGIC ) {
            sb->sb.wrongended=0;  /* everything is in the right order */
        } else {
            /* this is a backwards filesystem, turn everything around */
            sb->sb.wrongended=1;
            fprintf(stderr,"Other-endian filesystem.\n");
            return 0;
        }

        /* now that we're oriented, compute the computed fields */
        sb->sb.diskinfo = disk;
        sb->sb.firstIblock = 2 + /* two for boot record and superblock */
                               sb->sb.i_blocks + /* inode map blocks */
                               sb->sb.z_blocks; /* data zone blocks */
        sb->sb.zonesize = sb->sb.blocksize << sb->sb.log_zone_size; /*zone size*/
        sb->sb.ptrs_per_zone = (sb->sb.zonesize/sizeof(uint32_t));
        sb->sb.ino_per_block = (sb->sb.blocksize/sizeof(struct inode));
    }

    return res;
}

void print_superblock(FILE *where, superblock *sb) {
    /* print out the contents of a superblock for debugging
     * purposes
     */
    fprintf(stderr,"Stored Fields:\n");
    fprintf(stderr,"  ninodes      %6u\n", sb->sb.ninodes );
    fprintf(stderr,"  i_blocks     %6u\n", sb->sb.i_blocks );
    fprintf(stderr,"  z_blocks     %6u\n", sb->sb.z_blocks );
    fprintf(stderr,"  firstdata    %6u\n", sb->sb.firstdata );
    fprintf(stderr,"  log_zone_size %6u (zone size: %0u)\n",
            sb->sb.log_zone_size,
            sb->sb.blocksize << sb->sb.log_zone_size );
    fprintf(stderr,"  max_file     %10u\n", sb->sb.max_file );
    fprintf(stderr,"  magic        0x%04x\n",sb->sb.magic );
    fprintf(stderr,"  zones        %6u\n", sb->sb.zones );
    fprintf(stderr,"  blocksize    %6u\n", sb->sb.blocksize );
    fprintf(stderr,"  subversion    %6u\n", sb->sb.subversion );
    fprintf(stderr,"Computed Fields:\n");
    fprintf(stderr,"  firstIblock  %6u\n", sb->sb.firstIblock );

```

```

fprintf(stderr," zonesize      %6u\n", sb->sb.zonesize      );
fprintf(stderr," ptrs_per_zone %6u\n", sb->sb.ptrs_per_zone );
fprintf(stderr," ino_per_block %6u\n", sb->sb.ino_per_block );
fprintf(stderr," wrongended   %6d\n", sb->sb.wrongended );
}

void *read_zone(superblock *sb, int zone, char *buffer){
    /* read the zone-th zone from the disk
     * zone 0 is special (used for holes) and returns a blanked buffer.
     * returns buffer on success, NULL on failure
     */
    int i,base;
    void *res;

    res = buffer;

    if ( zone ) {
        /* This is a real zone on the disk */
        base = zone << sb->sb.log_zone_size;
        for(i=0; i < (1 << sb->sb.log_zone_size); i++) {
            if (NULL == read_block(sb->sb.diskinfo,base+i,sb->sb.blocksize,
                buffer+(i*sb->sb.blocksize))) {
                res=NULL;
                break;
            }
        }
    } else {
        /* we're reading a hole. Just blank out the buffer */
        memset(buffer,0,sb->sb.zonesize);
    }
    return res;
}

void *static_read_zone(superblock *sb, int znum){
    /* reads the given zone into a statically allocated buffer
     * defined here. This saves a little allocation overhead
     * for zones that'll only be used once. Returns a pointer
     * on success, NULL on failure
     */
    static char *zone=NULL; /* static for some efficiency boost */

    if ( !zone ) {
        zone=safe_malloc(sb->sb.zonesize);
    }

    return read_zone(sb,znum,zone);
}

void *write_zone(superblock *sb, int zone, char *buffer){
    /* write the zone-th zone to the disk
     * returns buffer on success, NULL on failure
     */
    int i,base;
    void *res;

    res = buffer;
    base = zone << sb->sb.log_zone_size;
    for(i=0; i < (1 << sb->sb.log_zone_size); i++) {
        if ( NULL == write_block(sb->sb.diskinfo, base+i,sb->sb.blocksize,
            buffer+(i*sb->sb.blocksize)) ) {
            res=NULL;
            break;
        }
    }
    return res;
}

static char *permstring(unsigned int mode) {
    /* returns a pointer to a statically allocated buffer containing expanded
     * permissions for this mode string in the form drwxrwxrwx
     * minix mode values can be found in include/sys/stat.h
     */
    static char res[10];

```



```

sprintf(res,"%c%c%c%c%c%c%c%c%c",
        MIN_ISDIR(mode)?'d':'-',
        (mode&0400)?'r':'-',
        (mode&0200)?'w':'-',
        (mode&0100)?'x':'-',
        (mode&0040)?'r':'-',
        (mode&0020)?'w':'-',
        (mode&0010)?'x':'-',
        (mode&0004)?'r':'-',
        (mode&0002)?'w':'-',
        (mode&0001)?'x':'-');
return res;
}

struct inode *read_inode(superblock *sb, int inumber, struct inode *inode){
    /* read the given inode from the given inode table
     * returns the pointer on success, NULL on failure
     * inode numbers start at 1
     */
    struct inode *ntable;
    struct inode *res;
    int blockno,nodeno;

    /* allocate space to hold one disk block worth of inodes */
    ntable = safe_malloc(sb->sb.ino_per_block * sizeof(struct inode));

    res = NULL;

    blockno = (inumber-1) / sb->sb.ino_per_block;
    nodeno = (inumber-1) % sb->sb.ino_per_block;

    if ( read_block(sb->sb.diskinfo,
        sb->sb.firstlblock+blockno,
        sb->sb.blocksize,
        ntable ) ) {
        *inode = ntable[nodeno]; /* ANSI C does a field-wise copy */
        res = inode;
    }

    free(ntable);

    return res;
}

void print_inode(FILE *where, struct inode *inode){
    /* print out the fields of an inode for debugging purposes */
    int i;

    fprintf(stderr," unsigned short mode      0x%04x\t(%s)\n",
        inode->mode, permstring(inode->mode));
    fprintf(stderr," unsigned short links      %6u\n",inode->links);
    fprintf(stderr," unsigned short uid      %6u\n",inode->uid );
    fprintf(stderr," unsigned short gid      %6u\n",inode->gid );

    fprintf(stderr," uint32_t size      %10u\n",inode->size );
    fprintf(stderr," uint32_t atime      %10u\n",inode->atime );
    fprintf(stderr," \t--- %s", int_ctime(inode->atime));
    fprintf(stderr," uint32_t mtime      %10u\n",inode->mtime );
    fprintf(stderr," \t--- %s", int_ctime(inode->mtime));
    fprintf(stderr," uint32_t ctime      %10u\n",inode->ctime );
    fprintf(stderr," \t--- %s\n", int_ctime(inode->ctime));
    fprintf(stderr," Direct zones:\n");
    for(i=0;i<DIRECT_ZONES;i++)
        fprintf(stderr,"          zone[%d]      = %10u\n",i,inode->zone[i]);

    fprintf(stderr," uint32_t indirect      = %10u\n",inode->indirect);
    fprintf(stderr," uint32_t double        = %10u\n",inode->two_indirect);
}

int copy_file(superblock *sb, int inumber, FILE *where){

```

```

/* returns zero on success, nonzero on failure
 * pretty straightforward: read a zone, then copy it.  Tries
 * to preserve holes if it can.
 */
220 struct inode ino;
int i,num;
int togo;
minfile *file;
char *zone;
int znum;

if ( read_inode(sb,inum,&ino) ) { /* grab the file's inode */
if ( !MIN_ISREG(ino.mode) ) {
230 fprintf(stderr,"Inode %d:  not a regular file.\n",inum);
return -1;
} else { /* do the copy */
file = new_minfile(sb,&ino); /* set up file structure */
togo = ino.size; /* how many bytes */
for(i=0;togo;i++) {
znum = file_zone_to_zone(file,i);
num = min(sb->sb.zonesize,togo);
if ( znum ) { /* its real.  Read it and copy it */
zone=static_read_zone(sb,znum);
240 fwrite(zone,1,num ,where);
} else { /* it's a hole.  seek if "where" is seekable*/
if ( -1==fseek(where,num,SEEK_CUR)) {
zone=static_read_zone(sb,znum); /* seek failed.  Do the copy */
fwrite(zone,1,num ,where);
}
}
togo -= num;
}
free_minfile(file);
250 }
} else {
fprintf(stderr,"unable to read inode %d\n",inum);
return -1;
}
return 0;
}

void listfile(superblock *sb,int inum, char *name, int limit,
FILE *where){
/* list the given file.  If limit is nonzero, limit the
260 * length of that string to limit.  If it is zero,
* expect the string to be null-terminated.
*/
struct inode ino;

read_inode(sb,inum,&ino);

if ( limit ) {
fprintf(where,"%s %9u %.*s\n",
permstring(ino.mode),
270 ino.size,DIRSIZ,
name);
} else {
fprintf(where,"%s %9u %s\n",
permstring(ino.mode),
ino.size,name);
}
}

void listdir(superblock *sb, int inumber, FILE *where){
280 /* given an inode corresponding to a directory, list
* the directory
*/
int ent_per_zone;
struct fileent *entry;
int togo;
int z;
int e;

```

```

struct inode ino;
minfile *file;
int znum;

ent_per_zone = sb->sb.zonesize / sizeof(struct fileent);

if ( read_inode(sb,inumber,&ino) ) { /* grab the file's inode */

    if ( ! MIN_ISDIR(ino.mode) ) { /* make sure this is a directory */
        fprintf(stderr,"%s: Not a directory\n",_FUNCTION_);
        return;
    }

    file = new_minfile(sb,&ino); /* set up file structure */

    /* now run through the zones printing out each directory entry */
    z = 0;
    znum = file_zone_to_zone(file,z);
    entry = static_read_zone(sb,znum);
    for (togo=ino.size;togo>0; ) {
        for(e=0; togo && e<ent_per_zone ; e++ ) {
            if (entry[e].ino) /* print it if it's real */
                listfile(sb,entry[e].ino,entry[e].name,DIRSIZ,where);
            else if ( verbose > 1 ) /* if verbose, even if it's not real */
                fprintf(where,"----- Deleted- %s\n",DIRSIZ,
                    entry[e].name);

            togo-=sizeof(struct fileent);
        }
        if ( togo ) { /* get the next zone, if necessary */
            znum = file_zone_to_zone(file,++z);
            entry = static_read_zone(sb,znum);
        }
    }
    free_minfile(file);
} else {
    fprintf(stderr,"unable to read inode %d\n",inumber);
}
}

static int lookupdir(superblock *sb, int inumber, char *name){
    /* given an inode corresponding to a directory and
    * a filename, see if the filename is in the directory.
    * if so, return the inode.
    * Only check valid entries, of course.
    * If valid, returns the i-number. Otherwise, 0.
    * valid returns return from the middle of the function
    */
    int ent_per_zone;
    struct fileent *entry;
    int togo;
    int z;
    int e;
    struct inode ino;
    minfile *file;
    int znum;

    /* first check to see if htis is even possible */
    if ( strlen(name) > DIRSIZ ) {
        return 0;
    }

    ent_per_zone = sb->sb.zonesize / sizeof(struct fileent);

    if ( read_inode(sb,inumber,&ino) ) { /* grab the file's inode */
        if ( ! MIN_ISDIR(ino.mode) ) { /* make sure this is a directory */
            fprintf(stderr,"%s: Not a directory\n",_FUNCTION_);
            return 0;
        }

        file = new_minfile(sb,&ino); /* set up file structure */

```

```

/* now run through the zones looking for the right directory entry */
z = 0;
znum = file_zone_to_zone(file,z);
entry = static_read_zone(sb,znum);
for (togo=ino.size;togo>0; ) {
    for(e=0; togo && e<ent_per_zone ; e++ ) {
        if (entry[e].ino){ /* only check if real */
            /* this block is the only difference between this function
             * and the previous one. Some tuning might be in order
             */
            if ( !strcmp(name,entry[e].name,DIRSIZ) )
                return entry[e].ino;
        }
        togo-=sizeof(struct fileent);
    }
    if ( togo ) { /* get the next zone, if necessary */
        znum = file_zone_to_zone(file,++z);
        entry = static_read_zone(sb,znum);
    }
}

free_minfile(file); /* clean up */
} else {
    fprintf(stderr,"unable to read inode %d\n",inumber);
}
return 0; /* if we got here, we didn't find it. */
}

int path_to_inode(superblock *sb, char *path){
/* given a superblock and a path name, return the inode
 * corresponding to the given file, or 0 if it does not exist
 */
int inum;
char *lpath,*name,*s;

lpath = safe_malloc(strlen(path)+1);
strcpy(lpath,path);

for(s=lpath;*s=='/';s++)
    ; /* trim leading '/' */

name = strtok(s,"/");
inum = 1; /* start at the root */
while(inum && name && strlen(name) ) {
    inum = lookupdir(sb,inum,name);
    name = strtok(NULL,"/"); /* get next token */
}
free(lpath);
return inum;
}

```

```

#include <stdlib.h>
#include <stdio.h>
#include <getopt.h>

#ifdef _sun_
extern int getopt(int, char *const *, const char *);
#else
#include <getopt.h>
#endif

#include "diskstuff.h"
#include "filesystem.h"
#include "options.h"

int verbose=0;          /* declared in options.h */

void usage(char *name) {
    fprintf(stderr,
        "usage: %s [ -v ] [ -p num [ -s num ] ] imagefile "
        "minixpath [ hostpath ]\n",
        name);
    fprintf(stderr, "Options:\n");
    fprintf(stderr,
        "\t-p\t part    --- select partition for filesystem "
        "(default: none)\n");
    fprintf(stderr,
        "\t-s\t sub     --- select subpartition for filesystem "
        "(default: none)\n");
    fprintf(stderr,
        "\t-h\t help    --- print usage information and exit\n");
    fprintf(stderr,
        "\t-v\t verbose  --- increase verbosity level\n");
    exit(-1);
}

int parse_options(int argc, char *argv[], struct options *opt){
    /*
     * This function parses the command line and sets parameters
     * based upon command line options.
     */
    extern int optind;
    char *end;
    int c;

    opt->part = NOTSET;
    opt->subpart = NOTSET;
    opt->imagefile=NULL;
    opt->srcpath = NULL;
    opt->dstpath = NULL;

    while ((c=getopt(argc, argv, "p:s:vh")) > 0 ) {
        switch (c) {
            case 'h': /* help */
                usage(argv[0]);
                break;
            case 'p': /* long listing */
                if ( opt->part != NOTSET ) {
                    fprintf(stderr, "%s: more than one partition specified.\n", argv[0]);
                    usage(argv[0]);
                }
                opt->part = strtol(optarg, &end, 0);
                if ( *end ) {
                    fprintf(stderr, "%s: badly formed integer.\n", optarg);
                    usage(argv[0]);
                }
                else if ( opt->part < 0 || opt->part > 3 ) {
                    fprintf(stderr, "Partition %d out of range. Must be 0..3.\n",
                        opt->part);
                    usage(argv[0]);
                }
                break;
            case 'v': /* long listing */
                verbose++;

```

```

    break;
case 's':
    /* long listing */
    if ( opt->subpart != NOTSET ) {
        fprintf(stderr,"%s: more than one subpartition specified.\n",argv[0]);
        usage(argv[0]);
    }
    opt->subpart = strtol(optarg,&end,0);
    if ( *end ) {
        fprintf(stderr,"%s: badly formed integer.\n",optarg);
        usage(argv[0]);
    } else if ( opt->subpart < 0 || opt->subpart > 3 ) {
        fprintf(stderr,"Subpartition %d out of range. Must be 0..3.\n",
            opt->subpart);
        usage(argv[0]);
    }
    break;
}
}

/* check for sanity */
if ( opt->part == NOTSET && opt->subpart != NOTSET ) {
    fprintf(stderr,"Cannot have a subpartition without a partition.\n");
    usage(argv[0]);
}

/* now the three filenames */
if ( optind < argc )
    opt->imagefile = argv[optind++];
else
    usage(argv[0]);

if ( optind < argc )
    opt->srcpath = argv[optind++];
else
    usage(argv[0]);

if ( optind < argc )
    opt->dstpath = argv[optind++];

if ( optind != argc )
    usage(argv[0]);
    /* extras? */

return optind;
}

void print_opts(FILE *where,struct options *opt) {
    fprintf(stderr," opt->part      %d\n",opt->part );
    fprintf(stderr," opt->subpart  %d\n",opt->subpart );
    fprintf(stderr," opt->imagefile %s\n",
        opt->imagefile?opt->imagefile:"(null)");
    fprintf(stderr," opt->srcpath   %s\n",opt->srcpath?opt->srcpath:"(null)");
    fprintf(stderr," opt->dstpath   %s\n",opt->dstpath?opt->dstpath:"(null)");
}

int main(int argc, char *argv[]) {
    struct options opt;
    struct disk disk;
    superblock sb;
    int inum;
    int err;
    struct inode ino;
    FILE *where;

    parse_options(argc,argv,&opt); /* parse options or don't return */

    if ( verbose > 1 ) {
        fprintf(stderr,"%nOptions:\n");
        print_opts(stderr,&opt);
    }

    /* open the disk image */
    if ( !open_disk(opt.imagefile,"r",&opt,&disk) ) {

```

```
fprintf(stderr,"Unable to open disk image \"%s\".\n",opt.imagefile);
exit(3);
}

if ( !read_superblock(&disk,&sb) ) {
    fprintf(stderr,"This doesn't look like a MINIX filesystem.\n");
    exit(-1);
}

if ( verbose ) {
    fprintf(stderr,"\nSuperblock Contents:\n");
    print_superblock(stderr,&sb);
    fprintf(stderr,"\n");
}

/* now, do the thing */
err = 0;
inum = path_to_inode(&sb,opt.srcpath);
if ( !inum ) { /* not found. Bummer */
    fprintf(stderr,"%s: File not found.\n",opt.srcpath);
    err++;
} else {
    read_inode(&sb,inum,&ino);
    if ( verbose ) {
        fprintf(stderr,"File inode:\n");
        print_inode(stderr,&ino);
    }
    if ( ! MIN_ISREG(ino.mode) ){
        fprintf(stderr,"%s: Not a regular file.\n",opt.srcpath);
        err++;
    } else {
        if ( opt.dstpath ) {
            if ( NULL == (where = fopen(opt.dstpath,"w")) ) {
                perror(opt.dstpath);
                exit(-2);
            }
        } else {
            where = stdout;
        }
        copy_file(&sb,inum,where); /* do it */
    }
}

/* close up and go home */
close_disk(&disk);
return err;
}
```

```

#include <stdlib.h>
#include <stdio.h>
#include <getopt.h>

#ifdef _sun_
extern int getopt(int, char *const *, const char *);
#else
#include <getopt.h>
#endif

#include "diskstuff.h"
#include "filesystem.h"
#include "options.h"

int verbose=0;          /* declared in options.h */

void usage(char *name) {
    fprintf(stderr,
        "usage:  %s  [ -v ] [ -p num [ -s num ] ] imagefile [ path ]\n",
        name);
    fprintf(stderr, "Options:\n");
    fprintf(stderr,
        "\t-p\t part    --- select partition for filesystem "
        "(default: none)\n");
    fprintf(stderr,
        "\t-s\t sub     --- select subpartition for filesystem "
        "(default: none)\n");
    fprintf(stderr,
        "\t-h\t help    --- print usage information and exit\n");
    fprintf(stderr,
        "\t-v\t verbose  --- increase verbosity level\n");
    exit(-1);
}

int parse_options(int argc, char *argv[], struct options *opt){
    /*
     * This function parses the command line and sets parameters
     * based upon command line options.
     */
    extern int optind;
    char *end;
    int c;

    opt->part = NOTSET;
    opt->subpart = NOTSET;
    opt->imagefile=NULL;
    opt->srcpath = NULL;
    opt->dstpath = NULL;

    while ((c=getopt(argc, argv, "p:s:vh")) > 0 ) {
        switch (c) {
            case 'h': /* help */
                usage(argv[0]);
                break;
            case 'p': /* long listing */
                if ( opt->part != NOTSET ) {
                    fprintf(stderr, "%s: more than one partition specified.\n", argv[0]);
                    usage(argv[0]);
                }
                opt->part = strtol(optarg, &end, 0);
                if ( *end ) {
                    fprintf(stderr, "%s: badly formed integer.\n", optarg);
                    usage(argv[0]);
                }
                else if ( opt->part < 0 || opt->part > 3 ) {
                    fprintf(stderr, "Partition %d out of range. Must be 0..3.\n",
                        opt->part);
                    usage(argv[0]);
                }
                break;
            case 'v': /* long listing */
                verbose++;
                break;
        }
    }
}

```



```

case 's':
    /* long listing */
    if ( opt->subpart != NOTSET ) {
        fprintf(stderr,"%s: more than one subpartition specified.\n",argv[0]);
        usage(argv[0]);
    }
    opt->subpart = strtol(optarg,&end,0);
    if ( *end ) {
        fprintf(stderr,"%s: badly formed integer.\n",optarg);
        usage(argv[0]);
    } else if ( opt->subpart < 0 || opt->subpart > 3 ) {
        fprintf(stderr,"Subpartition %d out of range. Must be 0..3.\n",
            opt->subpart);
        usage(argv[0]);
    }
    break;
}
}

/* check for sanity */
if ( opt->part == NOTSET && opt->subpart != NOTSET ) {
    fprintf(stderr,"Cannot have a subpartition without a partition.\n");
    usage(argv[0]);
}

/* now the three filenames */
if ( optind < argc ) /* image */
    opt->imagefile = argv[optind++];
else
    usage(argv[0]);

if ( optind < argc ) /* srcpath */
    opt->srcpath = argv[optind++];

if ( optind < argc )
    usage(argv[0]); /* extras? */

return optind;
}

void print_opts(FILE *where,struct options *opt) {
    fprintf(stderr," opt->part %d\n",opt->part );
    fprintf(stderr," opt->subpart %d\n",opt->subpart );
    fprintf(stderr," opt->imagefile %s\n",
        opt->imagefile?opt->imagefile:"(null)");
    fprintf(stderr," opt->srcpath %s\n",opt->srcpath?opt->srcpath:"(null)");
    fprintf(stderr," opt->dstpath %s\n",opt->dstpath?opt->dstpath:"(null)");
}

int main(int argc, char *argv[]) {
    struct options opt;
    struct disk disk;
    superblock sb;
    int inum;
    int err;
    struct inode ino;

    parse_options(argc,argv,&opt); /* parse options or don't return */

    if ( verbose > 1 ) {
        fprintf(stderr,"\nOptions:\n");
        print_opts(stderr,&opt);
    }

    /* open the disk image */
    if ( !open_disk(opt.imagefile,"r",&opt,&disk) ) {
        fprintf(stderr,"Unable to open disk image \"%s\".\n",opt.imagefile);
        exit(3);
    }

    if ( !read_superblock(&disk,&sb) ) {
        fprintf(stderr,"This doesn't look like a MINIX filesystem.\n");
        exit(-1);
    }

```

```
}

if ( verbose ) {
    fprintf(stderr, "\nSuperblock Contents:\n");
    print_superblock(stderr, &sb);
    fprintf(stderr, "\n");
}

/* now, do the thing */
if ( opt.srcpath == NULL )
    opt.srcpath = "/"; /* default to root */

err = 0;
inum = path_to_inode(&sb, opt.srcpath);
if ( !inum ) { /* not found. Bummer */
    fprintf(stderr, "%s: File not found.\n", opt.srcpath);
    err++;
} else {
    read_inode(&sb, inum, &ino);
    if ( verbose ) {
        fprintf(stderr, "File inode:\n");
        print_inode(stderr, &ino);
    }
    if ( MIN_ISDIR(ino.mode) ) {
        printf("%s:\n", opt.srcpath);
        listdir(&sb, inum, stdout); /* list a directory */
    } else {
        listfile(&sb, inum, opt.srcpath, NOLIMIT, stdout); /* list a file */
    }
}

/* close up and go home */
close_disk(&disk);
return err;
}
```

150

160

170

180

```
#ifndef OPTIONSH
#define OPTIONSH
```

```
struct options {
    int part;
    int subpart;
    int zshift;          /* for use by mkfs */
    int inodes;          /* for use by mkfs */
    char *imagefile;
    char *srcpath;
    char *dstpath;
};
```

```
extern int verbose;    /* used to control verbosity */
```

```
#endif
```

10

```
#ifndef UTILH
#define UTILH
#include <time.h>
void *safe_malloc(int size);
char *int_ctime(time_t t);
#endif
```

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void *safe_malloc(int size) {
    /* come back with memory, or don't come back */
    void *res;
    if ( NULL == ( res=malloc(size)) ) {
        perror("malloc");    /* if malloc() fails, just bail */
        exit(-1);
    }
    return res;
}

char *int_ctime(time_t t) {
    /* treat an int as a time and parse it.  Will work until 2038 :) */
    return ctime(&t);
}
```

10