# Assignment 4 Solutions

cpe 453 Spring 2010

```
Are there those in the land of the brave
Who can tell me how I should behave
  When I am disgraced
  Because I erased
A file I intended to save?
```

— /usr/games/fortune

Due by 11:59:59pm, Wednesday, June 2nd.
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: `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 ommitted 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 ommitted, 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

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/ibm/partition.h` | the partition table structure is here |
| `servers/fs/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 incompatable.

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**

| *Address* | +0 | +1 | +2 | +3 |
|---|---|---|---|---|
| | *DD* | *CC* | *BB* | *AA* |

**Big Endian**

| *Address* | +0 | +1 | +2 | +3 |
|---|---|---|---|---|
| | *AA* | *BB* | *CC* | *DD* |

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.ed`, and non-Intel Macintosh processors are big-endian.

## 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 intiutive. 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 $0 \ldots n$. For our purposes—reading a linear file from a disk—this is clearly preferable.

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.

| Structure | Size |
|---|---|
| Sector size: | 512b |
| Block size: | *in superblock* |
| Zone size: | $k \times$ blocksize |
| Endianness: | little |

Figure 1: Sizes used in the MINIX filesystem.

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.

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[1] sector number of the partition and `lFirst` + `size` − 1 gives last.

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

Figure 2: Partition table entry

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.

[1] That is, even for the subpartition table, the sector numbers are relative to the beginning of the disk, not the partition.

## 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 determinies the geometry of the rest of the filesystem.

*(see Figure 5-35 in T&W)*

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

| | |
|---|---|
| 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

## 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:

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

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 section 5.6.4 and Figure 5-36 in T&W)*

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. 3) is an unsigned long 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.

| Type | Name | Meaning |
|---|---|---|
| unsigned long | inode | inode number |
| unsigned char[60] | name | filename string |

Figure 3: A MINIX directory entry

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) | | | |
|---|---|---|---|
| **Start** | **Number** | **Contents** | **Description** |
| 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.<br>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 disk. The number of the first data zone is included as part of the superblock.<br>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

**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 a 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.

**Output**

The output for `minget` 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.

## Coding Standards and Make

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

## What to turn in

Submit via `handin` in the CSL to the `Asgn4` directory of the `pn-cs453` account:

- your well-documented source files.
- A makefile (called `Makefile`) that will build both programs with "`make all`".

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

Table 4: Some potentially useful system calls and library functions

- 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/Asgn4` 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 partition 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:
  unsigned short mode        0x41ff (drwxrwxrwx)
  unsigned short links            3
  unsigned short uid              2
  unsigned short gid              2
  unsigned long  size           384
  unsigned long  atime    1141098157 --- Mon Feb 27 19:42:37 2006
  unsigned long  mtime    1141098157 --- Mon Feb 27 19:42:37 2006
  unsigned long  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
  unsigned long  indirect         0
  unsigned long  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:**

```
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

LSOBJS  = minls.o $(COMMON)

GETOBJS = main.o $(COMMON)

HDRS    =

EXTRACLEAN = minls minget

.phony:

all:     $(PROGS)

minget: $(GETOBJS)
        $(LD) $(LDFLAGS) -o $@ $^

minls: $(LSOBJS)
        $(LD) $(LDFLAGS) -o $@ $^

allclean: clean
        @rm -f $(EXTRACLEAN)

clean:
        rm -f $(LSOBJS) $(GETOBJS) *~ 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
```

```
#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*SECTORSIZE;
    if ( -1 == fseek(disk->fp,offset,SEEK_SET) ) {
      rval=NULL;
    } else {
      if ( 1 != fread(buffer,SECTORSIZE,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*SECTORSIZE;
    if ( -1 == fseek(disk->fp,offset,SEEK_SET) ) {
      rval=NULL;
    } else {
      if ( 1 != fwrite(buffer,SECTORSIZE,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;
}

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;
}

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


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 %10lu %10lu\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
         );

}


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;       /* initial base pointer is zero, of course */
  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);                                                    220
        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;            230
        dp->fssize = ptable[opts->subpart].size * SECTORSIZE;
      } else {
        fprintf(stderr,"Not a Minix subpartition.\n");
        close_disk(dp);
        return NULL;
      }
    }
  }
  return dp;
}                                                                          240

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

```
#ifndef DISKSTUFF
#define DISKSTUFF

#include <stdio.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 partition {
  /* see include/ibm/partition.h */
  unsigned char bootind;
  unsigned char start_head;     /* start head */
  unsigned char start_sec;      /* start sector */
  unsigned char start_cyl;      /* start cylinder */
  unsigned char type;
  unsigned char end_head;       /* end head */
  unsigned char end_sec;        /* end sector */
  unsigned char end_cyl;        /* end cylinder */
  unsigned long lFirst;         /* logical first sector */
  unsigned long 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 <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 unsigned long *indzone; /* for reading out indirect zones */
  static unsigned long *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 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 <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(unsigned long));
    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       %6lu\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  %10lu\n", sb->sb.max_file           );
  fprintf(stderr,"  magic         0x%04x\n",sb->sb.magic          );
  fprintf(stderr,"  zones         %6lu\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 %6lu\n", sb->sb.ptrs_per_zone      );
  fprintf(stderr,"  ino_per_block %6lu\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 poitner
   * 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.firstIblock+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,"  unsigned long  size      %10lu\n",inode->size  );
  fprintf(stderr,"  unsigned long  atime     %10lu",inode->atime );
  fprintf(stderr,"\t--- %s",ctime((time_t *)&inode->atime));
  fprintf(stderr,"  unsigned long  mtime     %10lu",inode->mtime );
  fprintf(stderr,"\t--- %s",ctime((time_t *)&inode->mtime));
  fprintf(stderr,"  unsigned long  ctime     %10lu",inode->ctime );
  fprintf(stderr,"\t--- %s\n",ctime((time_t *)&inode->ctime));
  fprintf(stderr,"  Direct zones:\n");
  for(i=0;i<DIRECT_ZONES;i++)
    fprintf(stderr,"                zone[%d]    = %10lu\n",i,inode->zone[i]);

  fprintf(stderr,"  unsigned long  indirect %10lu\n",inode->indirect);
  fprintf(stderr,"  unsigned long  double   %10lu\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.
   */
  struct inode ino;
  int  i,num;
  int  togo;
  minfile *file;
  char *zone;
  int znum;

  if ( read_inode(sb,inumber,&ino) ) { /* grab the file's inode */
    if ( !MIN_ISREG(ino.mode) ) {
      fprintf(stderr,"Inode %d:  not a regular file.\n",inumber);
      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);
          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);
    }
  } else {
    fprintf(stderr,"unable to read inode %d\n",inumber);
    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
   * 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 %9lu %.*s\n",
            permstring(ino.mode),
            ino.size,DIRSIZ,
            name);
  } else {
    fprintf(where,"%s %9lu %s\n",
            permstring(ino.mode),
            ino.size,name);
  }
}

void listdir(superblock *sb, int inumber, FILE *where){
  /* 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 */

    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;
  int res;

  res = 0;

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

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

    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 ( !strncmp(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;
}
```

```
#ifndef FILESYTEMH
#define FILESYTEMH

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

/* types */
struct superblock {          /* this structure found in fs/super.h
                            * in minix 2.0.3
                            */                                                  10
  /* on disk.  These fields and orientation are non-negotiable */
  unsigned long  ninodes;          /* number of inodes in this filesystem */
  unsigned short pad1;             /* make things line up properly        */
  short          i_blocks;        /* # of blocks used by inode bit map    */
  short          z_blocks;        /* # of blocks used by zone bit map     */
  unsigned short firstdata;       /* number of first data zone */
  short          log_zone_size;   /* log2 of blocks per zone   */
short          pad2;             /* make things line up again */
  unsigned long  max_file;         /* maximum file size         */
  unsigned long  zones;            /* number of zones on disk   */          20
  short          magic;           /* magic number              */
short          pad3;             /* make things line up again */
  unsigned short 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 */
  unsigned short firstIblock;     /* number of first block with inodes */
  unsigned int   zonesize;        /* used all over the place */
  unsigned long  ptrs_per_zone;   /* number of zone pointers storable in          30
                                 * a zone */
  unsigned long  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];                                                   40
  struct superblock sb;
} superblock;

#define DIRECT_ZONES 7

struct inode {
  unsigned short mode;          /* mode */
  unsigned short links;         /* number or links */
  unsigned short uid;
  unsigned short gid;                                                           50
  unsigned long  size;
  unsigned long  atime;
  unsigned long  mtime;
  unsigned long  ctime;
  unsigned long  zone[DIRECT_ZONES];
  unsigned long  indirect;
  unsigned long  two_indirect;
  unsigned long  unused;
};
                                                                              60
#ifndef DIRSIZ
#define DIRSIZ  60
#endif

struct fileent {
  unsigned long  ino;
  char           name[DIRSIZ];
};

/* constants  (for v3) */                                                       70
#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);                    80
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);

/* macros */
#define MIN_ISREG(m) (((m)&0170000)==0100000)
#define MIN_ISDIR(m) (((m)&0170000)==0040000)                                90
#define NOLIMIT 0

#endif
```

```
#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 partition 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 */
      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 */
      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 */                                                    150
  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);                                              160
    }
    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);
        }                                                                    170
      } else {
        where = stdout;
      }
      copy_file(&sb,inum,where);  /* do it */
    }
  }

  /* close up and go home */
  close_disk(&disk);
  return err;                                                              180
}
```

```
#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 partition 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 */
      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 */
      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);                                    150
  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) ){                                             160
      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;                                                               170
}
```

```
#ifndef OPTIONSH
#define OPTIONSH

struct options {
  int  part;
  int  subpart;
  int  zshift;              /* for use by mkfs */
  int  inodes;              /* for use by mkfs */
  char *imagefile;                                                    10
  char *srcpath;
  char *dstpath;
};

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

#endif
```

```
#include <stdlib.h>
#include <stdio.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;
}
```

```
#ifndef UTILH
#define UTILH
void *safe_malloc(int size);
#endif
```