Assignment 5

cpe 453 Winter 2023

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, March 15th.
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 MINIXfilesystems 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 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

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 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					**Big Endian**			
Address	+0	+1	+2	+3	*Address*	+0	+1	+2	+3
	DD	*CC*	*BB*	*AA*		*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.edu`[1], 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 `int`s 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 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.

[1] 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:	*in superblock*
Zone size:	$k \times$ 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[2] 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

[2]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 determinies 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   */
  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 */
  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:

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

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 in 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) 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_{imap}	inode bitmap	a bitmap indicating which inodes are free The number of blocks (B_{imap}) used by the inode bitmap is contained in the superblock.
$2 + B_{\text{imap}}$	B_{zmap}	zone bitmap	a bitmap indicating which data zones are free. The number of blocks used by the zone bitmap (B_{zmap}) is contained in the superblock.
$2 + B_{\text{imap}} + B_{\text{zmap}}$	B_{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_{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. 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 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. 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.

`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)`
`long ftell(3)`
`void rewind(3)`
`int fgetpos(3)`
`int fsetpos(3)`
	File pointer positioning functions

`fread(3)` `fwrite(3)`	Stdio analogues of `read(2)` and `write(2)`
`strncpy(3)` `strncmp(3)`	Functions for manipulating limited-length strings
ctime(3)	parse a time into a string
getopt(3)	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.

%
```