Operating Systems **CSI3131** ***Lab 5*** *Winter 2011*

File Systems

Goal

1. Explore the creation of a UNIX like file system – The MINIX file system.

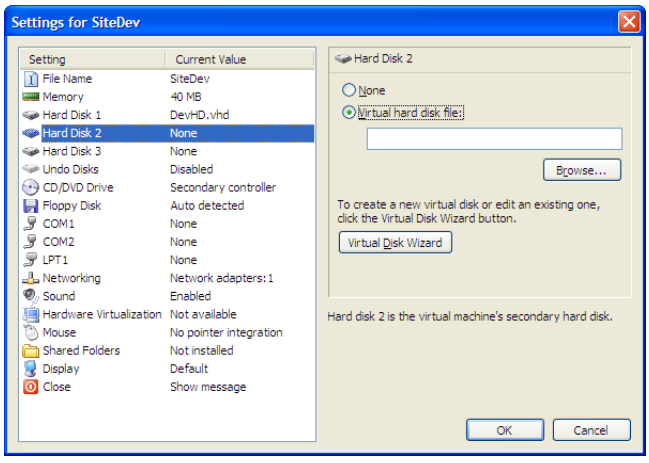
Environment

The lab will be completed using the Linux OS running under VirtualPC. You will not be able to use the SITE Linux systems, as you will need the superuser account *root* to complete your work. Virtual PC and the Linux virtual machines have been installed in the PCs in SITE 0110 lab. This is the same environment used for Lab 1. You can also install the environment in your system by downloading VirtualPC from http://www.microsoft.com/windows/virtualpc/default.mspx and download the "sitedev" Linux Virtual Machine via the link http://www.site.uottawa.ca/msdnaa/VirtualMachine_W07/SiteDev.zip (documentation on installing the virtual machine is available at http://www.site.uottawa.ca/msdnaa/VirtualMachine_W07/SiteLinuxVMs.pdf).
More information is available on the environment in the "Extra" lab documentation provided in Virtual campus.

Preparing the Hard Disk

Use the following steps to add a hard disk file to the virtual machine and start the Linux OS.

1. In the directory "*C:\VirtualMachine\SiteDev*", unzip the file *MinixFS.vhd* (from the zip file *MinixFS.zip*). This file will serve as a hard drive and includes a partition for creating a MINIX file system. The following steps allow you to add the file as a hard drive to the Linux Virtual machine.
   a. Start VirtualPC.
   b. Highlight the virtual machine named *SiteDev* click on *Settings*.
   c. Click on *Hard Drive 2*.
   d. On the right hand side of window, select *Virtual disk file* and used the *Browse* button to add the *MinixFS.vhd* file as the drive (see the illustration the right).
   e. Close the window by clicking on OK.



and the *hard* hard to

2. You can now start the Linux OS by clicking on the *Start* button in the main VirtualPC window.

3. A new terminal window will appear. Linux will boot and you will be prompted for a user name.
4. Start a Secure Shell (SSH), and connect the Linux OS, by connecting to the address 192.168.57.2. Do your work with SSH. Use the account *root* with password *site*.
5. Log in as root, and transfer the file *mkfs.minix.tar* to the root home directory. Untar the files. You will be using these files for the lab.

Creating an MINIX V1 File System in a 64 Mbytes Partition.

Consult Annex 1 for details on the organization of the MINIX file system. This lab will allow you to explore and complete software used to create a MINIX file system.

Given a 64 Mbytes Partition with 1K Blocks (64 Kilo Blocks), consider the following organization of the MINIX file system layout, using 30 byte file names (note that a zone is equivalent to a 1K disk block):

Book Block: 1 block (1K) – Zone 0
Super Block: 1 block (1K) – Zone 1
Inode Bit Map: 2 blocks (2K) – Zones 2 and 3 (8192 * bits = 16,384-1 = 16383 Inodes; Bit 0 not used).
Zone Bit Map: 8 Blocks (8K) – Zones 4 to 11
Inode Table: 512 blocks (512K) – Zones 12 to 523: 16383/32 = 512 Zones required for the inode table.
Data Zones: 65012 Blocks (65,012 K) – Zones 524 to 65535

Boot	SB	Inode Map	Zone Map	Inode Table	Data Zones

Although Linux still supports the MINIX Version 1 and Version 2 FS, the mkfs.minix utility is not supplied with the latest version of the OS. This is an opportunity to experiment with implementing the utility. The code supplied with this document is a project that creates a very basic implementation of the mkfs.minux utility creating a file system with the above defined layout. It consists of the following modules:

mkfs.minix.c: Contains the `main()` for processing command arguments (which is the name of a device corresponding to a hard disk partition that is at least 64 Mbytes in size). It also contains the function that creates the FS by first setting the first 512 bytes in the boot block to all zeros and then calling the other modules to 1) initialize the super block, 2) initialize the bit maps, and 3) create the root directory.
create_msblk.c: Contains the function `create_msblk()` that creates the super block.
create_inodes.c: Creates the inodes and bitmaps for inodes and zone. Note that the bits for inode 0 and data zone 0 are not used, that is, inodes and zones are indexed starting at 1. This allows for functions to return 0 when the inode table is full or all data zones are used up.
create_mroot.c: Creates the root directory for the file system. The root directory inode is created in inode 1 and initially contains the "." and ".." entries both pointing to inode 1.

A file called *Makefile* is provided to compile the source code into the *mkfs.minix* utility. Simply type "make" and the project is recompiled when a change is made to any of the source files.

PART A – The Overall Process

Open the *mkfs.minix.c* file and review the code. Make note of the following:

1. The *main* function is expecting that a device file is provided. Such a file gives direct access to a disk partition. In this lab, the device */dev/hdb1* allows access to the disk partition on the *MinixFS.vhd* hard disk file. VirtualPC presents this file to the Linux OS as a hard disk. Note that the hard disk was previously partitioned with *fdisk* (see notes in the section "Some extra notes" for more information on *fdisk*; you can also consult the Extra lab).
2. When the device file is opened, the disk partition is seen as one large file containing a sequence of bytes. Thus you are working directly on the file system. The *main* function opens this file for both reading and writing. The file description is passed on to the other functions for updating the file system.
3. The function *create_mfs* is called by main to create a MINIX file system on the partition. The function first zeros the boot block (first 512 bytes in Block 0) before calling the functions create_msblk, create_inodes, and *create_mroot* to create the superblock, create the bitmaps, and create the root directory in the partition. To write zeros to the boot block:
    a. Declare a character array of BOOTSIZE bytes (512) called *boot*;
    b. Zero the contents *boot* using the *memset* function.
    c. The *lseek()* call is used to move the file pointer to a position in the open file. When the file system file (*ex: /dev/hdb1*) is opened, the complete file system is seen as a sequence of bytes (sequential access). The *lseek* function allows direct access to the sequential file, that is, can move the file pointer throughout the file before read and write calls. In this case, *lseek()* moves the file pointer to the start of the file system (position 0).
    d. Write the contents of *boot* which will write BOOTSIZE (512) bytes at the start of the file system, i.e. in Block 0.

PART B – Creating the Super Block

Open the *create_msblk.c* file and review the code to understand how the superblock is being created. The super block will contain the following values. Macro definitions are defined in the table and used in a header file *mkfs.minix.h*.

Super Block Member	Value	Definition in mkfs.minix.h
s_inodes	(2*8*1024)-1 = 16383	NUMINODES
s_nzones	(64*1024)-1 (can only count to 65535)	TOTALBLOCKS
s_imap_blocks	(NUMINODES/8*1024)	NUMIMAPBLOCKS
s_zmap_blocks	8 (provides 64kbits)	NUMZMAPBLOCKS
s_firstdatazone	524	FIRSTZONE
s_log_zone_size	0	
s_max_size	((7+512+512*512)*1024)	MAXFILESIZE
s_magic	0x138f (MINIX_SUPER_MAGIC2 from minix_fs.h – see annex 1)	MAGICNUM
s_state	0x01	STATE
s_zones	00	Not used for V1.

Notes:

1. The *malloc()* system call allocates memory to the process for its use. By assigning the returned pointer to the *sblk_ptr* variable (a pointer to a superblock structure), it is then possible to assign values to the members of the *minix_super_block* structure. Then writing the superblock on disk can be done by simply writing the contents of the memory to the disk with *write* call. Notice also that to access a structure member, the "->" operator is used with a pointer. Note that the amount of memory allocated is a block size which is much larger than the actual superblock structure. Unused bytes in the block are set to zero.
2. The *memset()* call simply writes the character 0 to all of memory to zero any bytes in the superblock before setting any values.
3. Notice the use of constants in the assignment of the structure members. All these constants are defined in the *mkfs.minix.h* file.
4. The *lseek* call is used in this module to place the file pointer at the Block 1 in the file system (i.e. the superblock) before writing the superblock data.

PART C – Creating the Inode Table and the Bit Maps

Open the create_inodes.c file and review the code. Two bitmaps are initialised: the inode bitmap and zone (data) block bitmap. The inode table is also initialised by writing zeros to the table.

A bit corresponding to a number X (inode or data zone) in a bit map is determined as follows:

The address of the byte containing the bit can be computed using X>>3. The operation divides the number by 8. Consider X = 0x344F. Then byte 0x0589 will contain the bit. The bit is identified with the operation 1<<(X&0x7). Thus for our example, it is bit 7 in the addressed byte that corresponds to X. Note that this approach treats addresses lower order bits first. Thus when initializing the inode bit map to indicate that 0 is not used and inode 1 is used for the root directory, first byte is set to 0x03 (and not 0xC0 as one might expect).

Note the following:

1. The library functions *malloc* and *realloc* are used to allocate memory for various structures manipulated: the two bit maps (inode and zone), and an inode. The structures are manipulated in memory before being written onto the disk.
2. It is assumed that all bits are used in the inode bit map, i.e. that the inode table contains a multiple of 8192 (1024*8) inodes. Thus initialising the inode bit map consists of zeroing all bits in the bit map blocks. (As will be seen in the next part, inode 0 is not used and its bit will be set).
3. In the case of the zone bit map, not all bits will be used. Thus the bits from NUMZONES to TOTALBLOCKS are set since there is no corresponding data blocks for those bits.
4. The last action taken by the function is to initialize the inode table. It zeros all the entries in the inode table, i.e. all blocks in the inode table are filled with zeros.

PART D – Creating the root directory

Your task in the lab is to complete the *create_mroot.c* code that creates the root node of the file system. Creating the root directory consists of two steps:

1. Creating the inode for the root directory using the data defined in the table below.
2. Creating the directory table in the first data block (at FIRSTZONE).

The root inode (store in inode 1) contains the following values.

struct minix_inode Member	**Description**
i_mode	Set to S_IFDIR + 0755 (see "man 2 stat" for definition of S_IFDIR).
i_uid	getuid() (root user id – see "man getuid" for details).
i_size	2*DIRENTRYSIZE (2 entries will be added).
i_time	time(null) (Current time – see "man time").
i_gid	getgid() (root group id – see "man getgid").
i_nlinks	2 (2 directory entries)
i_zone[9]	FIRSTZONE (in first data zone block).

The directory table contains two entries, as shown in the tables below. Note that each entry is 32 bytes wide with the first 2 bytes containing the inode number and the other 30 bytes containing the name of the entry.

Inode Number (2 bytes)	**Name (30 bytes)**
1	..
1	.

Note that both entries refer to the inode 1. Only the root inode has the same inode number for both ".." and ".", that is, the root directory has no parent directory. Inode number 0 is not used (that is the inodes in the table are numbered starting at 1). This inode number is used to delete entries in the directory table, that is, when a name in a directory is deleted, its inode number is set to zero. Any new addition to the directory table can used these "deleted" entries.

Take the following steps:

1. Create the inode in memory, updates its values according to the above table, lseek to the start of the inode table (use START_INODE_TBL) and write the inode to the file system.
2. Write the value 0x03 in the first byte of the inode bitmap. This sets the bits for inode 0 and inode 1 as used.
3. Create the two entries for the root directory table in memory, seek to the first data block (at FIRSTZONE) and write the to the directory table to the data block.
    a. Note that inode values occupy 16 bits. For example if you create the array en memory "char rootd[BLOCK_SIZE]" to create the root directory table, to add an inode into the first entry of the array use:

```
*(__u16 *)rootd = 1;
```

`__u16`, defines a type of 16 bits;

`__u16 *`, defines a pointer to the type `__u16`;

`(__u16 *) rootd`, casts the adress `rootd` to an address to type `__u16`;

`* (__u16 *) rootd`, represents the contents addressed by an address to a `__u16`.

4. Once you have successfully completed your code, compile the project, and run the utility, you should be able to mount the MINIX file system using the following commands:
    a. *mkdir /dev/mnt1*: This is to create a mount point.
    b. *mount /dev/hdb1 /mnt1*: Will mount the file system to */mnt1*.
    c. You should see the mounted file system with the commands “df -T” and “mount”.
5. You can examine the contents of the file system using the “*od*” program as described in the section “Some extra notes”.
6. Try creating a few files and sub-directories in the mounted file system and examine the changes using “*od*”. Note that you will need to run the command “sync” to write any changes to the hard disk since the operating system uses buffering of file system data.

Some extra notes:

1. An empty disk can be created using Virtual PC as follows:
    a. Shutdown the Linux VM.
    b. Select the settings of the Virtual Machine.
    c. Click on one of the Hard Disks in the Settings window.
    d. A button "Virtual Disk Wizard" can be clicked to create a new drive. Follow instructions to create a fixed drive of 64Mbytes.
    e. Attach the new drive to the Virtual Machine in the settings dialog.
    f. Start the Virtual Machine. A new device such as */dev/hdb* will be found in the */dev/* directory. The utility *fdisk* is used to create and manage partitions on this drive (see next point).
2. The utility fdisk can be used to create a partition on the newly created hard drive.
    a. As super user execute "*fdisk /dev/hdb*", where *hdb* is the name of the new drive.
    b. Use "*m*" to get help on commands for *fdisk*.
    c. To create a new partition, use the n command.
    d. For example, you may create a primary partition that contains all of the hard drive.
    e. You can use "*p*" to print the new partition table.
    f. Use "*w*" to save and exit (be careful not to use "*q*" which does not update the partition.
    g. Reboot the system to see the new partition. You will be able to access it using the device name */dev/hdb1*, where hdb is the name of your hard drive.
    h. You can now create a file system on the new partition. To create a MINIX partition, use "*mkfs.minix /dev/hdb1*"
3. Other useful commands includes:
    a. *cat /dev/zero >/dev/hdb1*, which zeros the contents of the partition. It will be easier to see any updates make to the partition.
    b. "*od –Ax –t x1 /dev/hdb1 >jnk*. This command does a hex dump of the contents of the partition. It allows you to examine the layout of the hard disk. See "*man od*" for options on changing the output format.

Annex 1 - MINIX File System

Andrew Tanenbaum's MINIX Operating System served as the basis for the development of the Linux OS. The first version (developed in 1987) was designed to run on a 256K 8088 based IBM PC with 2 diskette drives and no hard disk and based on the operating system UNIX System V Version 7. The second version of MINIX can run on large Pentiums (in 32-bit protected mode) with more memory and large hard disks and was based on the POSIX standard (and international standard for the UNIX OS). A third version, MINIX 3, has just been published. It has migrated the operating system from a monolithic OS to a more modular system. This document presents the version 1 and version 2 MINIX file systems layout for the hard drive. Both these versions are still supported by the Linux operating system and will allow students to experiment with simple implementations of a file system.

The File System Layout

The MINIX file system layout is defined in terms of zones. A zone consists of a multiple number of blocks: 1, 2, 4, 8 .... (i.e. 2^n) blocks per zone. The initial version of the FS used 16 bit values as zone numbers. Using 1 block per zone (1Kbyte blocks) limited the file system to 64Mbytes. The file system could be increased by more blocks to each zone, but this could cause waste. In version 2, the zone numbers were increased to 32 bits, so that now FS system up to 4 Terabytes (with 1K Blocks and 1 Block/Zone). This document assumes that zones contain 1 block and thus the MINIX 1 FS is limited to 64 Mbytes. The layout of the MINIX FS is shown below. Both versions use the same layout.

Disk Layout for MINIX File System

Boot	SB	Inode Map	Zone Map	Inode Table	Data Zones

- Boot Block – used to boot from a partition. Although not need on all file systems, the first block is always reserved for the boot block for consistency. A magic number is used to determine if the boot block contains code for booting the system.
- Super Block – the second block in the FS contains information on the layout of the FS, including but not limited to number of blocks, size of blocks, number of inodes, maximum file size, magic number (to determine the version). It basically contains configuration information on the FS.
- I-Node Bit Map – one or more blocks containing a bit-map to keep track of free and busy inodes. A bit exists for each inode in the inode table and is set to 1 when the corresponding inode is busy and 0 when it is free.
- Zone Bit Map –The bit map works the same way the I-Node bit map, that is, a bit is used to indicate whether a zone is free or busy.
- Inode Table – A number of blocks containing the inodes for the FS. The inode contains status information on files and directories and the zone numbers of assigned zones. See details further in this document.
- Data – the rest of the FS contain the data zones (data blocks).

MINIX FS Version 1

This section will describe the Super Block and Inodes for Version 1 of MINIX. See the annex for the header file the structure declarations. Note that it does not contain a super block structure for version 1 of MINX since version 2 Super Block supports both versions of the layout.

Super Block – Version 1 MINIX FS		
Struct minix_super_block Member	**Size (bits)**	**Description**
s_inodes	16	Number of inodes in the inode table. The inodes are 32 bytes in length, which means that a 1 Kbytes Block can contain up to 32 inodes. A typical value for inodes is one third of the total number of disk blocks, rounded to completely fill blocks allocated to the inode table.
s_nzones	16	Total number of zones in the FS. This includes all zones and not just the data zones. The maximum number of zones is 2^{16} = 65536 zones. Thus for zones containing 1 K blocks, the maximum size of the FS is 64 Mbytes.
s_imap_blocks	16	The number of blocks allocated to the inode bit map. This corresponds to s_inodes/(number of bits in block). For a 1 K block that contains 8192 bits, this member is set to #inodes/8192.
s_zmap_blocks	16	The number of blocks allocated to the zone bit map. The number of blocks required can be computed using the following (for Version 1): (s_nzones – (1+s_imap_blocks+ s_inodes/32))/(number of bits in block). Unused bits in the bit map are set to 1 during initialization.
s_firstdatazone	16	First data zone in the system. (2+s_imap_blocks+ s_zmap_blocks+ s_inodes/32)
s_log_zone_size	16	This member gives the size of the zone in terms of the $\log_2$(number of blocs). The idea was to allow conversion of a zone number to a block number by using a shift operation. For example, when a zone contains 4 blocks (i.e. $\log_2(4) = 2$), the block number is determine using the operation Z#<<2, which multiplies the zone number (Z#) by 4. For our discussion, this value will always be set to 0, i.e. zone number corresponds to the block number.
s_max_size	32	The maximum size of a file, consist of maximum number of blocks allocation * size of block = (7+512+512*512)*1024 = 268,966,912 (approx 256 Mbytes) (see inode description for details).
s_magic	16	Magic number – set to 0x137F for Version 1 MINIX FS.

Inode – MINIX Version 1		
struct minix_inode Member	**Size (bits)**	**Description**
i_mode	16	Mode for file that defines permissions and type of device. See stat() for details on this value.
i_uid	16	User identifier that defines the owner of the file.
i_size	32	Size of the file in bytes.
i_time	32	The time last the file was modified.
i_gid	8	Group identifier of the file.
i_nlinks	8	Number of links to the file (number of times referenced from a directory files) or number of entries in directory.
i_zone[9]	16	Array that contains zone numbers. The first 7 contain data entries, the 8th a zone number to an indirect zone (contains set of zone number of data zones) and the 9^{th} contains the zone number of a double indirect block (zone numbers of data zones containing other zone numbers of data zones).

Directory File

The directory file for MINIX version 1 contains 16 byte entries, where the first two bytes contain an inode number and the following 14 bytes contains a null terminated file/directory name. The directory data block can contain up to 64 directory entries.

When Version 2 of the file system was created, the directory table entries were increased to 32 bytes. Versions with file names sizes of 30 bytes are also supported for version 1. When using 14 byte names, a directory data block can contain up to 64 entries and when using 30 byte names, a directory data block can contain up to 32 entries.

MINIX FS Version 2

This section will describe the Super Block and Inodes for Version 2 of MINIX. See the annex for the header file the structure declarations. The super block version 2 was designed to support both versions (1 and 2) of the MINIX FS.

Super Block – Version 2 MINIX FS		
Struct minix_super_block Member	**Size (bits)**	**Description**
s_inodes	16	Number of inodes in the inode table. Same as Version 1.
s_nzones	16	Total number of zones in the FS for a Version 1 FS. See s_zones for Version 2 number of zones below.
s_imap_blocks	16	The number of blocks allocated to the inode bit map. Same as version 1.
s_zmap_blocks	16	The number of blocks allocated to the zone bit map. Same as Version 1.
s_firstdatazone	16	First data zone in the system. (2+s_imap_blocks+ s_zmap_blocks+ s_inodes/16)
s_log_zone_size	16	This member gives the size of the zone in terms of the $\log_2$(number of blocs). See version 1 for a complete description. For our discussion, this value will always be set to 0, i.e. zone number corresponds to the block number.
s_max_size	32	Since version 2 inode provides a triple-indirect block, it is now possible to define a file size of 2^{32}-1 (i.e. 0xFFFFFFFF).
s_magic	16	Magic number – set to 0x137F for Version 1 MINIX FS and 0x2468 for Version 2. Two other versions that allow 30 byte names in the directory are defined for both versions. The magic numbers become 0x138f (V1 with 30 byte names) and 0x2478 (V2 with 30 byte names).
Padding	16	Padding. In Linux implementation defined as s_state, to indicate the state of the FS.
s_zones	32	Number of zones. Now can define up to 4 G-Zones (2^{32}), i.e. FS sizes of up to 4 Terabytes for 1 block Zones with 1 K-Byte blocks.

Inode – MINIX Version 2		
struct minix2_inode Member	**Size (bits)**	**Description**
i_mode	16	Mode for file that defines permissions and type of device. See stat() for details on this value. Same as V1.
i_nlinks	16	Number of links to the file (number of times referenced from a directory files) or number of entries in directory.
i_uid	16	User identifier that defines the owner of the file.
i_gid	16	Group identifier of the file.
i_size	32	Size of the file in bytes.
i_atime	32	The time last the file was accessed (contents read).
i_mtime	32	The time last the file was modified.
i_ctime	32	The time last the status of the file changed (includes changes to inodes).
i_zone[10]	32	Array that contains zone numbers. The first 7 contain data entries, the 8 a zone number to an indirect zone (contains set of zone number of data zones) and the 9^{th} contains the zone number of a double indirect block (zone numbers of data zones containing other zone numbers of data zones), and the 10^{th} is a triple indirect zone. The original version of MINIX 2 FS made the use of the triple indirect zone optional.

Directory File

The directory file for MINIX version 2 also contains 16 byte entries, where the first two bytes contain an inode number and the following 14 bytes contains a null terminated file/directory name. Versions with file names sizes of 30 bytes are also supported for both FS V1 and V2. When using 14 byte names, a directory data block can contain up to 64 entries and when using 30 byte names, a directory data block can contain up to 32 entries.

Annex 2 – Header file /usr/include/linux/minix_fs.h

```
#ifndef _LINUX_MINIX_FS_H
#define _LINUX_MINIX_FS_H
/*
 * The minix filesystem constants/structures
 */
/*
 * Thanks to Kees J Bot for sending me the definitions of the new
 * minix filesystem (aka V2) with bigger inodes and 32-bit block
 * pointers.
 */

#define MINIX_ROOT_INO 1

/* Not the same as the bogus LINK_MAX in <linux/limits.h>. Oh well. */
#define MINIX_LINK_MAX  250
#define MINIX2_LINK_MAX 65530

#define MINIX_I_MAP_SLOTS     8
#define MINIX_Z_MAP_SLOTS     64
#define MINIX_SUPER_MAGIC     0x137F            /* original minix fs */
#define MINIX_SUPER_MAGIC2    0x138F            /* minix fs, 30 char names */
#define MINIX2_SUPER_MAGIC    0x2468            /* minix V2 fs */
#define MINIX2_SUPER_MAGIC2   0x2478            /* minix V2 fs, 30 char names
*/
#define MINIX_VALID_FS        0x0001            /* Clean fs. */
#define MINIX_ERROR_FS        0x0002            /* fs has errors. */

#define MINIX_INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct minix_inode)))
#define MINIX2_INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct minix2_inode)))

#define MINIX_V1        0x0001           /* original minix fs */
#define MINIX_V2        0x0002           /* minix V2 fs */
```

```
/* This is the original minix inode layout on disk.
 * Note the 8-bit gid and atime and ctime. */
struct minix_inode {
      __u16 i_mode;
      __u16 i_uid;
      __u32 i_size;
      __u32 i_time;
      __u8  i_gid;
      __u8  i_nlinks;
      __u16 i_zone[9];
};

/* The new minix inode has all the time entries, as well as
 * long block numbers and a third indirect block (7+1+1+1
 * instead of 7+1+1). Also, some previously 8-bit values are
 * now 16-bit. The inode is now 64 bytes instead of 32.*/
struct minix2_inode {
      __u16 i_mode;
      __u16 i_nlinks;
      __u16 i_uid;
      __u16 i_gid;
      __u32 i_size;
      __u32 i_atime;
      __u32 i_mtime;
      __u32 i_ctime;
      __u32 i_zone[10];
};

/* minix super-block data on disk */
struct minix_super_block {
      __u16 s_ninodes;
      __u16 s_nzones;
      __u16 s_imap_blocks;
      __u16 s_zmap_blocks;
      __u16 s_firstdatazone;
      __u16 s_log_zone_size;
      __u32 s_max_size;
      __u16 s_magic;
      __u16 s_state;
      __u32 s_zones;
};

struct minix_dir_entry {
      __u16 inode;
      char name[0];
};
#endif
```