Defining a new File System on Minix 3.1.2a: Manual

Before you can start using the new file system, you have to make its presence felt by the system. The following source code changes,in explain how to do it whilst providing an insight into the design of the minix filesystem.

1. **/usr/src/servers/fs/type.h**
   This file defines the disk inode structure. Since our file system is totally new, we'll be adding a new entry here

```
typedef struct {    /* V2.x disk inode */
  mode_t d2_mode;   /* file type, protection, etc. */
  u16_t d2_nlinks;    /* how many links to this file. HACK! */
  uid_t d2_uid;     /* user id of the file's owner. */
  u16_t d2_gid;     /* group number HACK! */
  off_t d2_size;    /* current file size in bytes */
  time_t d2_atime;    /* when was file data last accessed */
  time_t d2_mtime;    /* when was file data last changed */
  time_t d2_ctime;    /* when was inode data last changed */
  zone_t d2_zone[V2_NR_TZONES]; /* block nums for direct, ind, and dbl ind */
  char d2_keyEFS[64]; /* Key for EFS */
} d4_inode;
```

   As you can see that I just created a new inode structure. It copies everything from V2 inode structures, used in Minix 2 and 3 file-systems. The new inode introduces a 64 byte entry for the key. Although a 32 byte entry would have sufficed, a 64 byte entry is needed to avoid the overlapping with the disk block boundary.

2. **/usr/src/servers/fs/const.h**
   Next you need to add some constants to identify the file system itself from the superblock. Some macros to calculate the number of inodes per block are required too.

```
#define V4_NR_DZONES       7  /* # direct zone numbers in a V2 inode */
#define V4_NR_TZONES      10  /* total # zone numbers in a V2 inode */

#define SUPER_EFS     0x1701  /* magic # for EFS file systems: V4 */
#define V4                 4  /* version number of V4 file systems */

#define V4_ZONE_NUM_SIZE usizeof (zone_t)  /* # bytes in V2 zone  */
#define V4_INODE_SIZE    usizeof (d4_inode)  /* bytes in V2 dsk ino */
```

```
#define V4_INDIRECTS(b)  ((b)/V4_ZONE_NUM_SIZE)  /* # zones/indir block */
#define V4_INODES_PER_BLOCK(b) ((b)/V4_INODE_SIZE)/* # V2 dsk inodes/blk */
```

This information is used by the mount system call to perform the mount operation and by read() system call to identify your filesystem as EFS.

3. **/usr/src/servers/fs/inode.h**
   The structure of the inode table has to be modified to allow space for the key field.

```
 EXTERN struct inode {
  mode_t i_mode;    /* file type, protection, etc. */
  nlink_t i_nlinks; /* how many links to this file */
  uid_t i_uid;      /* user id of the file's owner */
  gid_t i_gid;      /* group number */
  off_t i_size;     /* current file size in bytes */
  time_t i_atime;   /* time of last access (V2 only) */
  time_t i_mtime;   /* when was file data last changed */
  time_t i_ctime;   /* when was inode itself changed (V2 only)*/
  zone_t i_zone[V2_NR_TZONES];
                    /* zone numbers for direct, ind, and dbl ind */
  char keyEFS[64];  /* EFS Key for accessing file */
  /* The following items are not present on the disk. */
  dev_t i_dev;      /* which device is the inode on */
  ino_t i_num;      /* inode number on its (minor) device */
  int i_count;      /* # times inode used; 0 means slot is free */
  int i_ndzones;    /* # direct zones (Vx_NR_DZONES) */
  int i_nindirs;    /* # indirect zones per indirect block */
  struct super_block *i_sp;
                    /* pointer to super block for inode's device */
  char i_dirt;      /* CLEAN or DIRTY */
  char i_pipe;      /* set to I_PIPE if pipe */
  char i_mount;     /* this bit is set if file mounted on */
  char i_seek;      /* set on LSEEK, cleared on READ/WRITE */
  char i_update;    /* the ATIME, CTIME, and MTIME bits are here */
} inode[NR_INODES];
```

4. **/usr/src/servers/fs/buf.h**
   The system maintains arrays of LRU lists of the disk blocks. The buffer that does that is file system agnostic, i.e. it does not maintain separate lists for different file system. This is achieved by declaring the data portion of the buffer as a union of all the possible inode formats. The following code illustrates the afore mentioned fact:

```
EXTERN struct buf {
  /* Data portion of the buffer. */
  union {
    char b__data[_MAX_BLOCK_SIZE];         /* ordinary user data */
/* directory block */
    struct direct b__dir[NR_DIR_ENTRIES(_MAX_BLOCK_SIZE)];
/* V1 indirect block */
    zone1_t b__v1_ind[V1_INDIRECTS];
/* V2 indirect block */
    zone_t  b__v2_ind[V2_INDIRECTS(_MAX_BLOCK_SIZE)];
/* V1 inode block */
    d1_inode b__v1_ino[V1_INODES_PER_BLOCK];
/* V2 inode block */
    d2_inode b__v2_ino[V2_INODES_PER_BLOCK(_MAX_BLOCK_SIZE)];
/* V4 inode block */
    d4_inode b__v4_ino[V4_INODES_PER_BLOCK(_MAX_BLOCK_SIZE)];
/* bit map block */
    bitchunk_t b__bitmap[FS_BITMAP_CHUNKS(_MAX_BLOCK_SIZE)];
  } b;

  /* Header portion of the buffer. */
  struct buf *b_next;   /* used to link all free bufs in a chain */
  struct buf *b_prev;   /* used to link all free bufs the other way */
  struct buf *b_hash;   /* used to link bufs on hash chains */
  block_t b_blocknr;    /* block number of its (minor) device */
  dev_t b_dev;      /* major | minor device where block resides */
  char b_dirt;      /* CLEAN or DIRTY */
  char b_count;     /* number of users of this buffer */
} buf[NR_BUFS];
```

5. **/usr/src/servers/fs/proto.h**
   You need to define system call for logging into the system. My implementation defines do_EFSlogin( ) to do this. Please refer to TA's manual for an indepth implementation detail of a system call. The required entry for the system call was:

```
      /* keym.c */
_PROTOTYPE( int do_EFSlogin, (void)         );
```

6. **/usr/src/commands/simple/mkfs.c**
   The main processing starts here. This is where you plugin the appropriate super block identifier and give a logical structure to your disk. If the user is trying to make an encrypted file system then he is prompted for a password. The following code snipped from main() shows how the file system version is set and how the block size is allocated:

```
.
.
.
   case ’e’:
    printf("Case 4");
        if(getuid()!=0)
        {
                printf("Only root permitted to make an EFS\n");
                return(EPERM);
        }
        passwd=getpass("EFS login :");

==========================
Setup for authentication
==========================
        fs_version = 4;
        break;
.
.
.
 else if(fs_version == 4) {
    if(!block_size) block_size = _MIN_BLOCK_SIZE; /* MAX to min EFS */
    if(block_size%SECTOR_SIZE || block_size < _MIN_BLOCK_SIZE) {
      fprintf(stderr, "block size must be multiple of sector (%d) "
        "and at least %d bytes\n",
        SECTOR_SIZE, _MIN_BLOCK_SIZE);
      pexit("specified block size illegal");
    }
    if(block_size%V4_INODE_SIZE) {
      fprintf(stderr, "block size must be a multiple of inode
                       size (%d bytes)\n", V4_INODE_SIZE);
      pexit("specified block size illegal");
    }
  }
```

Changes need to be made in the super block too. The function super() takes care of that. The easiest way to make those changes are to copy the appropriate code from the V3 filesytem processing code and modify it so that it works for your V4(EFS) code:

```
else /* Retaining the same functionality */
    {
      // Doing calculations for double indirect blocks
      v2sq = (zone_t) V4_INDIRECTS(block_size) * V4_INDIRECTS(block_size);
      //total zones accessible
      zo = V4_NR_DZONES + (zone_t) V4_INDIRECTS(block_size) + v2sq;
      sup->s_magic = SUPER_EFS;     // Defined in const.h
      sup->s_block_size = block_size; // defined in main()
      sup->s_disk_version = 0; // 0 by default
#define MAX_MAX_SIZE  ((unsigned long) 0xffffffff)
      if(MAX_MAX_SIZE/block_size < zo) {
        sup->s_max_size = MAX_MAX_SIZE;
      }
      else {
        sup->s_max_size = zo * block_size;
      }
```

I have commented the code above, but you need to read the complete chapter 5 for indepth understanding.

7. **/usr/src/servers/fs/inode.c**
   The file system architecture is quite rigid on Minix(Or for that matter on on OS I know of). All the inode operations are hard coded for each type of file system. We need to specify how inodes will be read/written from the disk. A new function new_icopy4() takes are of reading from the disk into the memory and vice versa.

```
/*==========================================================*
 *         rw_inode             *
 *==========================================================*/
PUBLIC void rw_inode(rip, rw_flag)
register struct inode *rip; /* pointer to inode to be read/written */
int rw_flag;      /* READING or WRITING */
{
/* An entry in the inode table is to be copied to or from the disk. */

  register struct buf *bp;
  register struct super_block *sp;
  d1_inode *dip;
  d2_inode *dip2;
  d4_inode *dip4; /*EFS*/
...
...
 dip4 = bp->b_v4_ino + (rip->i_num - 1) %  /*EFS*/
     V4_INODES_PER_BLOCK(sp->s_block_size);
...
...
if (sp->s_version == V1)
  old_icopy(rip, dip,  rw_flag, sp->s_native);
  else if(sp->s_version == V4)
       /* Takes care of the new inode structure for EFS */
       new_icopy4(rip, dip4, rw_flag, sp->s_native);
  else
       new_icopy(rip, dip2, rw_flag, sp->s_native);
...
...
}

/*=============================================================*
 *         new_icopy4 :EFS           *
 *=============================================================*/
PRIVATE void new_icopy4(rip, dip, direction, norm)
```

```
register struct inode *rip; /* pointer to the in-core inode struct */
register d4_inode *dip; /* pointer to the d2_inode struct */
int direction;      /* READING (from disk) or WRITING (to disk) */
int norm;     /* TRUE = do not swap bytes; FALSE = swap */

{
/* Same as new_icopy, but to/from V4 disk layout. */

  int i;
  if (direction == READING) {
  /* Copy V4.x inode to the in-core table, swapping bytes if need be. */
  rip->i_mode    = conv2(norm,dip->d2_mode);
  rip->i_uid     = conv2(norm,dip->d2_uid);
  rip->i_nlinks  = conv2(norm,dip->d2_nlinks);
  rip->i_gid     = conv2(norm,dip->d2_gid);
  rip->i_size    = conv4(norm,dip->d2_size);
  rip->i_atime   = conv4(norm,dip->d2_atime);
  rip->i_ctime   = conv4(norm,dip->d2_ctime);
  rip->i_mtime   = conv4(norm,dip->d2_mtime);
  rip->i_ndzones = V2_NR_DZONES;
  rip->i_nindirs = V4_INDIRECTS(rip->i_sp->s_block_size);
  for (i = 0; i < V2_NR_TZONES; i++)
    rip->i_zone[i] = conv4(norm, (long) dip->d2_zone[i]);
  memcpy(rip->keyEFS,dip->d2_keyEFS,64);
  } else {
  /* Copying V4.x inode to disk from the in-core table. */
  dip->d2_mode   = conv2(norm,rip->i_mode);
  dip->d2_uid    = conv2(norm,rip->i_uid);
  dip->d2_nlinks = conv2(norm,rip->i_nlinks);
  dip->d2_gid    = conv2(norm,rip->i_gid);
  dip->d2_size   = conv4(norm,rip->i_size);
  dip->d2_atime  = conv4(norm,rip->i_atime);
  dip->d2_ctime  = conv4(norm,rip->i_ctime);
  dip->d2_mtime  = conv4(norm,rip->i_mtime);
  for (i = 0; i < V2_NR_TZONES; i++)
    dip->d2_zone[i] = conv4(norm, (long) rip->i_zone[i]);
  memcpy(dip->d2_keyEFS,rip->keyEFS,64);
  }
}
```

8. **/usr/src/servers/fs/super.c**
   The functions in this file are called while reading a super block during a file system mount. The main job is just to initialize the filesytems using the values read from the superblock.

```
PUBLIC int read_super(sp)
register struct super_block *sp; /* pointer to a superblock */
{
```

```
/* Read a superblock. */
...
..
 /* Get file system version and type. */
  if (magic == SUPER_MAGIC || magic == conv2(BYTE_SWAP, SUPER_MAGIC)) {
  version = V1;
  native  = (magic == SUPER_MAGIC);
  } else if (magic == SUPER_V2 || magic == conv2(BYTE_SWAP, SUPER_V2)) {
  version = V2;
  native  = (magic == SUPER_V2);
  } else if (magic == SUPER_V3) {
  version = V3;
    native = 1;
  } else if (magic == SUPER_EFS) {
    version = V4;
    native =1;
  } else {
  return(EINVAL);
  }
..
...
}
```

9. **/usr/src/servers/fs/main.c**
   This file contains the code for initializing the file system. It has certain hard wired conditions which, if not taken care of, can lead to a FS panic, hence a system failure.

```
void fs_init()
{
...
if (V1_INODE_SIZE != 32) panic(__FILE__,"V1 inode size != 32", NO_NUM);
if (V2_INODE_SIZE != 64) panic(__FILE__,"V2 inode size != 64", NO_NUM);
if (V4_INODE_SIZE != 128) panic(__FILE__,"V2 inode size != 128", NO_NUM);
...
```

   The last condition was added by me. We just added a 64 byte key to the V2 inode, therfore the size of our V4 inode is 128 bytes.

Wrapping up

1. Execute **#make install** after any file modification to compile your changes.
2. Modify /usr/src/servers/fs/Makefile. Add keym.o to **OBJ** and -lcrypt to **LIBS** (assuming that you have installed libcrypt library)
3. Goto /usr/src and $make world. This compiles everything that is modified and installs the new image.