

How To Manipulate the Inode Structure?

Created by Karthick Jayaraman
Syracuse University

TABLE OF CONTENTS

1.	INTRODUCTION.....	2
2.	HOW TO REFERENCE THE FILENAME FOR THE BINARY OF AN EXECUTING PROGRAM IN EXEC CALL?.....	2
3.	HOW TO REFERENCE THE INODE OF A FILE USING THE POINTER TO THE “STRUCT FILP”?	2
4.	HOW TO REFERENCE THE RIGHT FPROC STRUCTURE WHEN CALLING A FS SYSTEM CALL FROM MM?	3
5.	HOW TO STORE SOME DATA IN THE INODE?.....	3

1. Introduction

Some projects like in the computer security class involve storing and retrieving some security related attributes in the Inode. This is the trickiest part of the project, because there is no documentation about minix file system. Also, some tasks may involve writing filesystem system calls that are called in the memory management system (MM) in places like the `exec.c`. Calling a FS system call from a user program is different from calling it inside the MM. When a user program calls a FS system call the ***fp*** in the FS references the `fproc` structure of the caller program. But, when the FS system call is called from MM, the ***fp*** references the `fproc` of the MM process. This document is essentially a “HOW TO” document for these gray areas.

2. How to reference the filename for the binary of an executing program in exec call?

```
PUBLIC int do_exec()
{
/* Perform the execve(name, argv, envp) call. The user library builds a
 * complete stack image, including pointers, args, environ, etc. The stack
 * is copied to a buffer inside MM, and then to the new core image.
 */

register struct mproc *rmp;
struct mproc *sh_mp;
int m, r, fd, ft, sn, fd2;
static char mbuf[ARG_MAX]; /* buffer for stack and zeroes */
static char name_buf[PATH_MAX]; /* the name of the file to exec */
```

In projects that involve retrieving certain details from the inode you may want to reference the filename of the executing binary. The exec code contains a static char buffer ***name_buf*** that contains the complete filename(including path) for the binary of an executing program. If you have want to write FS system call that reads from the executing file’s Inode, open the file in the exec code where you have access to the name of the file and pass the file descriptor of the file to the FS system call.

3. How to reference the Inode of a File using the pointer to the “struct filp”?

The “***struct filp***” of each file contains a pointer to the inode “***filp_ino***”.

```

EXTERN struct filp {
  mode_t filp_mode;           /* RW bits, telling how file is opened */
  int filp_flags;            /* flags from open and fcntl */
  int filp_count;           /* how many file descriptors share this slot?*/
  struct inode *filp_ino; /* pointer to the inode */
  off_t filp_pos;           /* file position */
}

```

4. How to reference the right fproc structure when calling a FS system call from MM?

The MM code contains a global variable “*who*” that contains the proc number of the currently executing code. The “*who*” variable can be used to index the fproc array to get the right fproc structure. Write FS system call such that you pass the “*who*” variable and use it in the system call to reference the right fproc structure.

Example: Calling a FS system call from the exec call in MM.

Code snippet from Exec.c:

```

name = name_buf; /* name of file to exec. */
fd2 = open(name,O_RDONLY);
assigncap(fd2,who); “who” is passed to the FS system call.
close(fd2);

```

code snippet from the implementation of the system call.

```

tfp=&fproc[m.m1_i2];

```

The m.m1_i2 contains the variable “*who*” and is used to select the right fproc structure from the fproc array.

5. How to store some data in the Inode?

A Minix file system has six components:

1. The Boot Block which is always stored in the first block. It contains the boot loader that loads and runs an operating system at system startup.
2. The second block is the Superblock which stores data about the file system, that allows the operating system to locate and understand other file system structures. For example, the number of inodes and zones, the size of the two bitmaps and the starting block of the data area.
3. The inode bitmap is a simple map of the inodes that tracks which ones are in use and which ones are free by representing them as either a one (in use) or a zero (free).
4. The zone bitmap works in the same way as the inode bitmap, except it tracks the zones

5. The inodes area. Each file or directory is represented as an inode, which records metadata including type (file, directory, block, char pipe), ids for user & group, three timestamps that record the data & time of last access, last modification and last status change. An inode also contains a list of addresses that point to the zones in the data area where the file or directory data is actually stored.
6. The data area is the largest component of the file system, using the majority of the space. It is where the actual file and directory data are stored.

Each file or directory has an Inode. The data area of the disk is split into structures called blocks. Data associated with the file/directory is stored in the blocks and a set of blocks makes up a zone. The Inode for the file/directory contains pointers to the zones. The zone pointer is of data type `zone_t`, which is an unsigned long containing the zone number. There are three kinds of zone pointers depending on how they map on to zone containing data.

```
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 */

    /* 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];
```

The first category is called the direct zones. The first seven zone pointers [0-6] in the Inode are direct zones. These zones identified by the direct zone pointers contain the data by themselves. Zone pointer 7 is called single indirect zone pointer. A single indirect zone pointer points to a zone, which in turn contains zone pointers that point to the zones containing data.

Zone pointer 8 is called double indirect zone pointer. In this case, there are two levels of indirection starting from the zone pointer in the Inode to reach the zone containing data. Zone pointer 9 is called triple indirect zone pointer, which has three levels of indirection to reach the zone containing data.

The `alloc_zone` function is used to allocate a block. This function returns the zone number of an unused zone. As mentioned earlier, each zone contains a fixed number of blocks. The \log_2 (Number of blocks per zone) is stored in the member variable `log_zone_size` of the superblock structure in the inode. The member variable `i_sp` in the Inode points to the super block.

```
struct super_block {
    ino_t s_ninodes;           /* # usable inodes on the minor device */
    zone1_t s_nzones;        /* total device size, including bit maps etc */
    short s_imap_blocks;     /* # of blocks used by inode bit map */
    short s_zmap_blocks;     /* # of blocks used by zone bit map */
    zone1_t s_firstdatazone; /* number of first data zone */
    short s_log_zone_size;  /* log2 of blocks/zone */
    off_t s_max_size;       /* maximum file size on this device */
    short s_magic;          /* magic number to recognize super-blocks */
    short s_pad;            /* try to avoid compiler-dependent padding */
    zone_t s_zones;         /* number of zones (replaces s_nzones in V2) */
} super_block;
```

```
/* Scaling factor: This number is used for zone -> block conversion */
scale = rip->i_sp->s_log_zone_size;
```

The mapping from blocks to zone is many-to-one, organized such that each consecutive set of blocks, belong to consecutive zones. Say each zone contains 8 blocks. Under this arrangement, the first 8 blocks belong to zone 1, the next 8 blocks belong to zone 2 and so on. To obtain the first block in a zone, we simply have to left shift the zone number using the scaling factor contained in `s_log_zone_size`.

```
b = (block_t)rip->i_zone[9] << scale;
```

This operation is equivalent to performing `Zone[9]*(number of blocks per zone)`. For most of the files, the triple indirect pointer `zone[9]` is unused. The `zone[9]` can be safely used to store some additional meta data about the file without crashing the filesystem.

To use the `zone[9]`, you need to allocate a zone for it using the `alloc_zone` call. We need to ensure that `zone[9]` is unallocated. This can be done by simple comparing `zone[9]` with the macro `NO_ZONE`. Then, we can obtain the block number of the first block by simple left shifting of the zone number by number stored in `log_zone_size`. We can initialize the block to contain all zeroes using the function `zero_block`.

```
if(rip->i_zone[9] == NO_ZONE)
{
    rip->i_zone[9] = alloc_zone(rip->i_dev,rip->i_zone[9]);
    b = (block_t)rip->i_zone[9] << scale;
    bp = get_block(rip->i_dev,b,NORMAL);
    zero_block(bp);
}
```

```

        printf("\nzone is allocated for the program \n");
    }
    else
    {
        printf("Retrieving the zone[9] from the inode \n");
        b = (block_t)rip->i_zone[9] << scale;
        bp = get_block(rip->i_dev,b,NORMAL);
    }

```

Once we get the block number, we can get a pointer to the disk block using the `get_block` system call. This function takes the block number and returns a pointer to the on disk block, which a pointer to structure of kind (struct buf *).

```

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

```

The member variable `b_data` in struct `buf` is used to store user data. The `b_data` member can be indexed like an ordinary byte array and can be used to store data. For instance, in our implementation we use each byte in `b_data` array to indicate one of the five capabilities. A 1 stored in byte 1 of `b_data` array indicates the presence of capability 1, and a 0 indicates the apposite.

```

    /* flag 1 means to add capability and 2 means to revoke capability */
    if(flag == 1)
        bp->b_data[cap] = 1;
    if(flag == 2)
        bp->b_data[cap] = 0;

```

Once we make changes to the block, the dirty flag of both the inode and block should be set and we should update both the inode and the block on the disk.

```

    bp->b_dirt=DIRTY;
    rip->i_dirt=DIRTY;
    put_block(bp,FULL_DATA_BLOCK);
    put_inode(rip);

```