IMPLEMENTING IMMEDIATE FILES IN MINIX OPERATING SYSTEM

A THESIS
Submitted by

Shrishty Chandra B110076CS
and
Pragati Maan B110836CS

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Under the guidance of
DR MURALI KRISHNAN

DEPARTMENT OF COMPUTER ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY CALICUT
NIT CAMPUS PO, CALICUT
KERALA, INDIA 673601

May 18, 2015

ACKNOWLEDGEMENTS

We would like to express our gratitude and appreciation to all those who helped us complete this project. First and foremost, we would like to thank our project guide Dr Murali Krishnan, for his guidance and encouragement. We would also like to thank Sharath Hari N and Sudev A C, passouts of 2014 batch for their help in getting us started with the project.

PRAGATI MAAN
SHRISHTY CHANDRA

DECLARATION

"I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text".

Place:
Date:

Signature :
Name :
Reg.No:

CERTIFICATE

*This is to certify that the thesis entitled: "***IMPLEMENTING IMMEDIATE FILES IN MINIX OPERATING SYSTEM***" submitted by Sri/Smt/Ms to National Institute of Technology Calicut towards partial fulfillment of the requirements for the award of Degree of Bachelor of Technology in Computer Science Engineering is a bonafide record of the work carried out by him/her under my/our supervision and guidance.*

Signed by Thesis Supervisor(s) with name(s) and date

Place:
Date:

Signature of Head of the Department

Office Seal

Contents

Chapter

Abstract

In most of the computer systems, accessing disk files acts as the bottleneck in performance. So while designing, we try to minimise the number of disk accesses. Immediate file is one solution to this problem. The idea is to store files of small size in the inode itself, instead of storing the pointers to the disk blocks. Once the size of the file exceeds the memory available for block pointers in the inode, it is converted to a regular file. This also saves the disk block, which would otherwise have been wasted to store a very small size of data ie. internal fragmentation. Minix is an operating system that was basically created for educational purposes. It consists of a microkernel that is considered to be highly reliable. We aim at implemeting the immediate file system in minix 3.2, which includes an abstract layer called the virtual file system besides the minix file system

Figures

Figure

Chapter 1

Problem Definition

Implement support for immediate files in Minix Operating System.

Chapter 2

Introduction

2.1 Minix

Minix was initially developed for educational purposes by Prof. Andrew Tanenbaum. Minix 3 has a highly reliable, secure and flexible microkernel OS. A minimal kernel provides

- interrupt handlers
- a mechanism for starting and stopping processes
- a scheduler
- interprocess communication
- deadlock detection

The **file system**, **device drivers**, **the network server** and **high level memory management** run as appropriate user processes that are encapsulated in their private address space.

2.2 Immediate File System

In minix, the **metadata** of a file is stored in form of inodes. Inodes contain information such as last access time, modification time, file size, permissions etc. along with the pointers to the disk block where the data of a file is stores. These

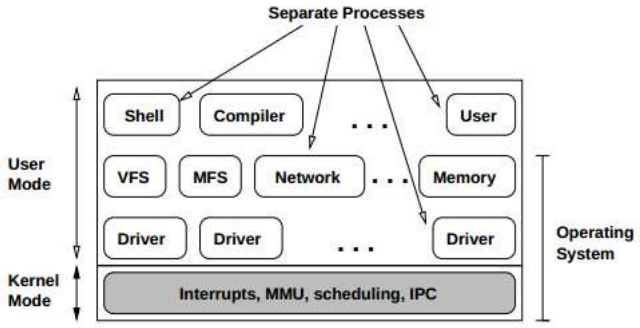

Figure 2.1: Minix Layred Structure

pointers either directly refer to a disk block, or they refer to a list of additional pointers to data blocks (such pointers are called indirect). The problem with a regular file is that even when it is very short, a complete disk block needs to be allocated. This wastes disk space.

In immediate files, the data is stored directly in the inode instead of the disk. An inode in Minix is 64 bytes long, and 40 bytes are used to hold pointers to data blocks. When no data blocks are used, these 40 bytes can be used to store the file content directly. Thus, for files up to 40 bytes, immediate files work and hence getting rid of fragmentation. Another important thing about immediate files is that the number of disk accesses is reduced for short files and hence reducing the access time.

2.3 File System in Minix

In minix, File System is basically a network file server that happens to be running on the same machine as the caller. The communication between various abstract layer is via messages. Messages from user include - **access, chdir, chmod, chown, chroot, close, creat** etc system calls. The main program in the file system waits for new messages to arrive and handles the work according to the parameters passed in the message. There are six sections within the Minix

File System:

- **Blank Block** first block is reserved for boot code information
- **Super Block** second block stores the Super Block, or information about the Minix File System
- **Inode Map** section made up of bits, where one bit represents one inode. Tracks used and unused inodes.
- **Zone Map** section made up of bits to track used and unused zones.
- **Inode Table** manages file and device information
- **Data Zone** majority of volume which contains files and directories.

2.4 File System in Minix 3.2

The file system in minix is more modular than the earlier versions because of inclusion on Virtual File System. It makes the access to the file systems easier by providing a uniform interface. When any file operation is to be done, first a call is made from the user program to the virtual file system, which consecutively passes in to the appropriate file system.

2.4.1 Virtual File Systems

All the system calls are directed to the Virtual File System, which directs them to the appropriate File Systems using messages and setting the necessary flags. The response from the File Systems also arrives at the VFS which sends it to the user level programs using appropriate message formats and flag sets. It makes adding new file systems very easy since the interface is taken care of by the VFS. Roles of VFS are as follows:

- **Handles POSIX** system calls.
- **Maintains state** - cooperates with the process manager to handle fork, exec and exit system calls.
- Keeps **track** of endpoints that are drivers for character or block special files.

VFS is **synchronous**. It sends a request to FS process and waits until the response arrives. It contains data structures corresponding to almost all the data structures in File Systems.

- Virtual Nodes : Vnode Object- abstract correspondence of a file. It contains inode number of file , FS Process kernel endpoint number

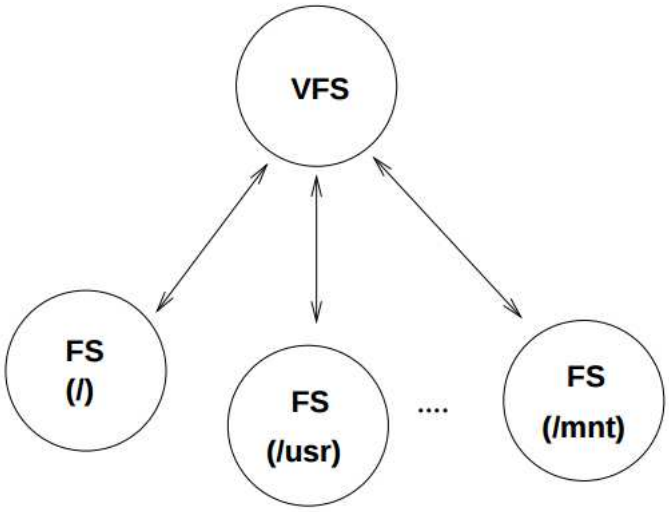

Figure 2.2: VFS Layer

- Virtual Mounts : Vmnt Object- Stores information about the mounted partitions.
- Contains the kernel endpoint number of the File System that manages the given partition, device number, mount flags etc.

VFS spawns **worker-threads** at startup. Main thread fetches requests and replies and hands them off to the idle or reply pending workers. **open.c** is an important file in VFS. It contains procedures for creating, opening, closing and seeking on files- **CREAT, OPEN, MKNOD, MKDIR, CLOSE, LSEEK** are the entry points to this call.

request.c is the file which consists of the functions which pass on the request to file systems, in proper response messages.

2.4.2 System Calls in MFS

System call is how a user program requests a service from an operating system's kernel. Generally, systems have a library which define these system calls.

In minix, there are two components to a system call -

- **User Library** - Packages the parameters for system calls and calls the handler on appropriate server.
- **System call** handler -executed in serve processes, called in response to a user requesting a system call.

In each server directory, there are two important files - table.c and proto.h.

- **table.c** - contains the information regarding which file is to be called in reponse to which system call number.
- **proto.h** - declares the prototype of system call handler.

misc.c, stadir,c, write.c and read.c contain definitions for system call handler functions.

2.4.3 Example: Read System Calls in MFS

$$n = read(fd, buffer, nbytes) \tag{2.1}$$

Library procedure read is called with three parameters - file descriptor, buffer, and number of bytes to be read. It builds a message containing these parameters along with the code for read as message type, sends the message to VFS and blocks awaiting the reply. The VFS implementation of the system call is already explained in VFS section.

In the corresponding file system, a procedure extracts the file descriptor from the message and uses it to locate the filp (open file table)) entry and then the inode. The requests are broken up into pieces such that each piece fits within a block. For each piece, chunk is made to see if the relevant block is in cache. If not then LRU algorithm is applied. Once the block is in cache, the file system sends message

to the system task asking it to copy the data to appropriate place in the user's buffer. FS sends the reply message to the user, specifying how many bytes have been copied.

2.4.4 Message passing

There are many types of messages requesting work in the File System. Message passing is basically dealt by the kernel, so, for file system purposes, we just need to understand how to use messages. Different types of flags and fields are passed via messages across different layers of the operating system.

Chapter 3

Design and Implementation

3.1 Basic File Structure

The **/usr/src/servers/mfs** directory contains the source code for FS in minix3.2 operating system. Some of the important files in mfs are **main.c, inode.h, open.c, write.c, buf.h, super.h, super.c,** etc. The main function of each file in **mfs** are listed below:

- **buf.h** - Defines the block cache. It contains a **union** named **fsdata_u** with following attributes:
  - **b__data**[**MAX_BLOCK_SIZE**], a character array, containing ordinary user data.
  - **b__dir**[**NR_DIR_ENTRIES(_MAX_BLOCK_SIZE)**] - directory block
  - **b__bitmap**[[**FS_BITMAP_CHUNKS(_MAX_BLOCK_SIZE)**]] - bitmap block
  - direct and indirect inode blocks
- **cache.c** - FS has a buffer cache to reduce the number of disk accesses. File contains 9 procedures, few of them are listed below.
  - **get_block** - Fetch a block for reading/writing.

  - **put_block** - Return a block
  - **rw_block** - Transfer block between disk and cache
  - **free_zone** - If file is deleted, free the zone
  - ...

- **const.h** - Defines constants, like flags, table size that will be used in the file system. Few constants are: **IN_CLEAN, IN_DIRTY, ATIME, CTIME** etc
- **fs.h** - Master header for FS, includes all header files needed by the MFS source files.
- **glo.h** - Defines all the global variables. Few examples of global variables are **fs_m_in, fs_m_out, err_code, fs_dev, user_path[PATH_MAX]** etc.
- **inode.h** - Contains the stucucture for *inode* and the *inode_table* as **inode[NR_INODES]**
- **inode.c** - Contains functions which manages the inode table. The functions are **get_inode(), put_inode(), rw_inode(), alloc_inode()** etc
- **main.c** - Contains the main routiene of the file system. The main loop does three activities
  - **get_work(&fs_m_in)** - Gets a new work
  - **Processes** the work i.e. selects the function to be called using table of function pointers.
  - **reply(src, &fs_m_out)** - Sends a reply
- **open.c** - Contains the codes for six system calls: **open, close, mknode, mkdir, close, lseek**

- **proto.h** - Lists all function prototypes for all functions used in MFS
- **read.c** - All functions that are used for reading or writing are present in read.c. Some of the functions include, **fs_readwrite, rw_chunk, read_map** etc.
- **super.h** - Contains the superblock table. Super block holds information about inode bitmap, zone bitmaps, inodes etc.
- **super.c** - Handles the superblock table and other related data structures like zone bitmap, inode bitmap etc. Major functions in this file are: **alloc_bit(), free_bit(), get_super(), read_super()** etc.
- **table.c** - Contains the table that map system call numbers onto the routines.
- **write.c** - Contains files that are not in read.c but are necessary for writing in a file. Most important functions in **write.c** are **write_map, clear_zone, new_block** and **zero_block**

3.2 Design and Algorithm Immediate File System

There are two ways in which immediate files can be implemented in the minix operating system:

- **Static** - In this approach, the maximum file size is specified at the creation time i.e. user himself specifies whether the file will be immediate or regular and the file type can't be changed once it has been created. In case the immediate file size exceeds the specified size, it will report an error.
- **Dynamic** - A file is created as an immediate file and if size exceeds specified size (of immediate files), it becomes a regular file. The user doesnt have to bother about the size of the file

We have used **dynamic** approach in our project.

3.2.1 Detailed Algorithm to include Immediate files

Algorithm 1 Algorithm to include immediate files in the file system

```
1: procedure FS_READWRITE_IMMED
2:     is_immediate = 0 (0 if regular, 1 if immediate)
3:     mode = inode.i_mode (regualar or immediate)
4:     pos = req_msg.LSEEK_POS (lseek position)
5:     nrbytes = req_msg.NBYTES (number if bytes to be read or written)
6:     rw_flag = req_msg.m_type (READING or WRITING)
7:     f_size = inode.f_size
8:     immed_buff[] = "" (temporary array)
9:     if mode == I_IMMEDIATE then
10:        if rw_flag == WRITING then
11:            if (f_size + nrbytes) >MAX_IMMEDSIZE then
12:                if (pos + nrbytes) <MAX_IMMEDSIZE then
13:                    i_immediate = 1
14:                else
15:                    /** Shift from Immediate to regular **/
16:                    Copy the content of zone to immed_buff array
17:                    Mark all zones as empty_zone
18:                    Change inode.size to zero
19:                    Change update_time of inode
20:                    Mark the inode dirty
21:                    Request a new block, bp
22:                    Copy the immed_buff content to bp.data field
23:                    Mark the block, bp dirty
24:                    Update pos and f_size
25:                    inode.i_mode = REGULAR
26:                    is_immediate = 0 // file is no more immediate
27:                end if
28:            else
29:                is_immediate = 1
30:            end if
31:        else
32:            /** reading no change required **/
33:            is_immediate = 1
34:        end if
35:    end if
36:    if is_immediate == 1 then // the file is still immediate
37:        Calculate the zone_position in the disk
38:        Call system_read or system_write function with zone_pos as argument
39:    end if
40: end procedure
```

3.2.2 Implementation using dynamic approach

This section will give an overview of what our algorithm actually does.

Data Structures Involved All data strucutres which are related to our project are listed below

- **inode** - Structure of an inode in a disk is given in Figure2.2

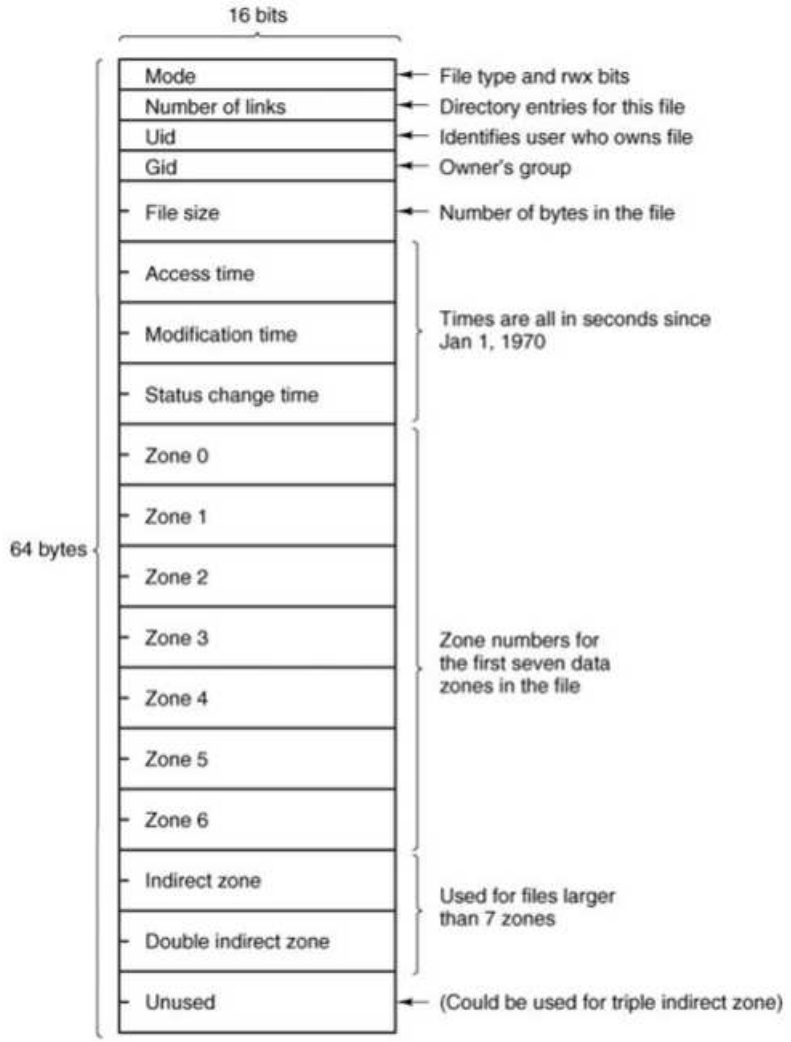

Figure 3.1: Inode Structure

In Figure 3.1. we can see that there are 7 zones of 4 bytes each, an indirect

zone of size 4 bytes, a double indirect node of size 4 bytes and an unused space of 4 bytes. Each zone points to a disk block where actual data gets stored. According to definition of immediate files we had to find a space in the inode where we can store the immediate files. These zones are apt place to store the immediate files because no data which is critical to the file is being affected. We calcualted **maximum size** of immediate files as **40 bytes** by adding up sizes of all zones, indirect zones and unused space.

- **buffer or block cache** - This is a union of different types of blocks in the disk. Eg. normal data block, directory block, inode block, bitmap block etc. The design of buffer or block cache is given below,

```
union fsdata_u {
  /* ordinary user data */
  char b__data[_MAX_BLOCK_SIZE];
  /* 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)];
  /* bit map block */
  bitchunk_t b__bitmap[FS_BITMAP_CHUNKS(_MAX_BLOCK_SIZE)];
};
```

b__data array is used to cache the data which is stored in the disk block, all the modifications by the user are done here and then the data is written back to the disk. **b_data(b)** is a macro which returns the pointer to the

first byte of **b__data** array.

- message - message structure is defined as follows

```
typedef struct {
  int m_source;         // message source
  int m_type;           // message type
  union {
        mess_1 m_m1;
        mess_2 m_m2;
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_7 m_m7;
        mess_8 m_m8;
  } m_u;
} message;
```

mess_1, mess_2, mess_3, ... are different message types. In MFS, global variables, **fs_m_in** and **fs_m_out**, of type **message** are used to send and recieve messages from various servers like VFS. Following system calls are used for message passing: **echo**, **notify**, **sendrec**, **receive**, **send**.

Files involved List of all files in which codes are added or deleted.

- **/src/include/const.h** - A new flag **I_IMMEDIATE** is created to support immediate files.
- **src/sys/lib/libsa/minixfs3.h** - A new flag **I_IMMEDIATE** is created to support immediate files.
- **/src/servers/vfs/open.c** - When **O_CREATE** flag is set. Set the file mode as immediate instead of regular. It will have size zero.

```
  /* In function common_open */
if (oflags & O_CREAT) {
// we have removed I_REGULAR mode
      omode = I_IMMEDIATE | (omode & ALLPERMS & fp->fp_umask)
  ;
vp = new_node(&resolve, oflags, omode);
r = err_code;
if (r == OK) exist = FALSE; /* file created */
else if (r != EEXIST) {   /* other error */
  if (vp) unlock_vnode(vp);
  unlock_filp(filp);
  return(r);
}

```

- **src/sys/sys/stat.h** - Following additions are done in this file:
  * **_S_IFIMMED** - macro was defined, similar to regular files

```
  #define _S_IFIMMED 0130000      /* Immediate files */

```

  * **S_IFIMMED** - macro redifined for easier usage, similar to regular files.

```
#define S_IFIMMED _S_IFIMMED     /* Immediate files */

```

* **S_ISIMMED(m)** - macro defined which checks whether a files is immediate or not (similar to regular files)

```
    /* Immediate */
#define S_ISIMMED(m) (((m) & _S_IFMT) == _S_IFIMMED)

```

- There are **114** more files where we have located **S_ISREG(m)** and added **S_ISIMMED(m)** also in the code, because regular files and immediate files have same functionalities except that immediate files are stored in inode and regular files are stored in disk blocks pointed by zones in the inode. Few of the files are listed below:
  * **/servers/vfs/select.c**
  * **/servers/vfs/link.c**
  * **/servers/vfs/read.c**
  * **/commands/grep/mmfile.c** ...etc
- **/src/severs/mfs/read.c** - Major changes/addition for implementation of immediate file system is done in this file, in the **fs_readwrite()** function.

```
/** in File read.c **/
/** start **/
cum_io = 0;
char immed_buff[41];
if ((rip->i_mode & I_TYPE) == I_IMMEDIATE) {
  int is_immediate;
  int i;
  if (rw_flag == WRITING) {
   if ((f_size + nrbytes) > 40) {
```

```
10      if (position == 0 && nrbytes <= 40) {
11        is_immediate = 1;
12      } else {
13        register struct buf *bp;
14        for (i = 0; i < f_size; i++) {
15          immed_buff[i] = *(((char *) rip->i_zone)+i);
16        }
17
18        for (i = 0; i < V2_NR_TZONES; i++) {
19          rip->i_zone[i] = NO_ZONE;
20        }
21        rip->i_size = 0;
22        rip->i_update = ATIME | CTIME | MTIME;
23        IN_MARKDIRTY(rip);
24
25        bp = new_block(rip, (off_t) 0);
26
27        if (bp == NULL)
28          panic("error");
29
30        for (i = 0; i < f_size; i++) {
31          b_data(bp)[i] = immed_buff[i];
32        }
33
34        MARKDIRTY(bp);
35        put_block(bp, PARTIAL_DATA_BLOCK);
36
37        // same as after rw_chunk is called
38        position += f_size;
39        f_size = rip->i_size;
40        rip->i_mode = I_REGULAR;
41        is_immediate = 0;
42      }
```

```
43    } else {
44      is_immediate = 1;
45    }
46  }
47   if (is_immediate == 1) {
48     if (rw_flag == READING) {
49       r = sys_safecopyto(VFS_PROC_NR, gid,
50       (vir_bytes) cum_io,
51       (vir_bytes)(rip->i_zone + position),
52       (size_t) nrbytes);
53     } else {
54       r = sys_safecopyfrom(VFS_PROC_NR, gid,
55       (vir_bytes) cum_io,
56       (vir_bytes)(rip->i_zone + position),
57       (size_t) nrbytes);
58       IN_MARKDIRTY(rip);
59     }
60
61     if (r == OK) {
62       cum_io += nrbytes;
63       position += nrbytes;
64       nrbytes = 0;
65     }
66     for (int i = 0; i < f_size; i++) {
67       immed_buff[i] = *(((char *) rip->i_zone)+i);
68     }
69     printf("immedbuf: %s\n", immed_buff);
70   }
71 }
72 /** end **/
```

Chapter 4

Further Work

4.1 About Website

We have created a website - `minixnitc.github.io` so that all the further projects done on minix can be compiled here. All that we learned while working on the project is posted here.

4.1.1 Guide to Minix

This section consists of notes from the textbook -Design and Implementation of Operating Systems by Andrew Tanenbaum. Since this book is based on earlier versions of minix, we have included our own understanding of the code wherever we found that the system deviates from the test. Because on inclusion of Virtual File System in MINIX 3.2, there is a considerable difference in File System and System Call implementation.

4.1.2 Hands-on Tutorials on Minix OS

- `www.cs.ucsb.edu/~ravenben/classes/170/html/projects.html`
- `web.fe.up.pt/~pfs/aulas/lcom2014/labs/doc/MinixMacVB.pdf`
- `www.cis.syr.edu/~wedu/seed/Documentation/Minix3/How_to_add_system_call.pdf`

- cise.ufl.edu/class/cop4600sp14/Minix-Syscall_Tutorialv2.pdf
- www.phien.org/ucdavis/ta/ecs150-f03/syscall.html
- wiki.minix3.org/doku.php?id=developersguide:newkernelcall

4.1.3 Implementation

This section consists of detailed explanation of the functions in file system and VFS which were changed while implementing Immediate files. The relevant code in implementation has also been explained in detail.

4.2 Possible Projects

Minix is an ever growing system and there are many projects that can be done based on it at b.tech level, since minix was basically developed for education purposes. Here are the links to Google Summer Of Code page for Minix

- GSOC 2011 - http://wiki.minix3.org/doku.php?id=soc:2011:start
- GSOC 2012 - http://wiki.minix3.org/doku.php?id=soc:2012:start
- GSOC 2013 - http://wiki.minix3.org/doku.php?id=soc:2012:start
- One can further explore the system for more ideas

Bibliography

[1] Design and implementation of the minix virtual file system, http://www.minix3.org/theses/gerofi-minix-vfs.pdf.

[2] Minix3 developers page, http://wiki.minix3.org.

[3] Andrew S. Tanenbaum, S. J. M. Immediate files, http://dare.ubvu.vu.nl/bitstream/handle/1871/2604/11033.pdf.

[4] Dr Muralikrishnan, Sharath Hari N, S. A. C. Experiments with minix operating system - http://sudevambadi.me/minixmajor/.

[5] Tanenbaum, A. S. Book: Operating System, Design and Implementation, Pearson Prentice Hall, 3rd Edition. 1987.