Project 7

Design:

With each code addition modification, we included Design comments for what we did and why we did it. They are listed above each file name and directory.

Code Modification/Addition Listing:

Note: Changes are underlined

We set a flag for if the file is an immediate file

Usr/src/include/minix/const.h:

```
#define I_SYMBOLIC_LINK 0120000         /* file is a symbolic link */
#define I_REGULAR       0100000  /* regular file, not dir or special */
#define I_BLOCK_SPECIAL 0060000         /* block special file */
#define I_DIRECTORY     0040000/* file is a directory */

#define I_IMMEDIATE     0050000         /* immediate file */

#define I_CHAR_SPECIAL  0020000         /* character special file */
#define I_NAMED_PIPE    0010000         /* named pipe (FIFO) */
```

We set a flag for the creation of an immediate file

usr/src/include/fcntl.h:

```
/* Oflag values for open().  POSIX Table 6-4. */
#define O_CREAT       00100       /* creat file if it doesn't exist */
#define O_EXCL        00200       /* exclusive use flag */
#define O_NOCTTY      00400       /* do not assign a controlling terminal */
#define O_TRUNC       01000       /* truncate flag */

#define O_IMM         00030       /* immediate file flag */
```

This is for the ls command, so i will be displayed

usr/src/commands/fsck.mfs/fsck.h:

```
/* List the given inode. */
void list(ino_t ino, d_inode *ip)
{
  if (firstlist) {
        firstlist = 0;
        printf(" inode permission link   size name\n");
  }
  printf("%6u ", ino);
  switch (ip->i_mode & I_TYPE) {
    case I_REGULAR:               putchar('-');     break;
    case I_IMMEDIATE:             putchar('i');  break;
    case I_DIRECTORY:             putchar('d');     break;
    case I_CHAR_SPECIAL:          putchar('c');     break;
```

AND:

```
/* Check the mode and contents of an inode. */
int chkmode(ino_t ino, d_inode *ip)
{
 switch (ip->i_mode & I_TYPE) {
   case I_REGULAR:
        nregular++;
        return chkfile(ino, ip);
     case I_IMMEDIATE:
        nimmediate++;
        return chkfile(ino, ip);
   case I_DIRECTORY:
        ndirectory++;
```

We had to set the rules for when we are opening or creating an immediate file instead of a regular file. A lot of this involves setting a condition for if the file type is regular or immediate.

src/servers/vfs/Open.c:

```
/*===========================================================================*
 *                          common_open                              *
 *===========================================================================*/
PRIVATE int common_open(register int oflags, mode_t omode)
{
/* Common code from do_creat and do_open. */
  int b, r, exist = TRUE;
  dev_t dev;
  mode_t bits;
  struct filp *fil_ptr, *filp2;
  struct vnode *vp;
  struct vmnt *vmp;
  struct dmap *dp;

  /* Remap the bottom two bits of oflags. */
  bits = (mode_t) mode_map[oflags & O_ACCMODE];
  if (!bits) return(EINVAL);

  /* See if file descriptor and filp slots are available. */
  if ((r = get_fd(0, bits, &m_in.fd, &fil_ptr)) != OK) return(r);
  /* If O_CREATE is set, try to make the file. */
  if (oflags & O_CREAT && !(oflags & O_IMM)) {
        omode = I_REGULAR | (omode & ALL_MODES & fp->fp_umask);
      vp = new_node(oflags, omode);
      r = err_code;
      if (r == OK) exist = FALSE; /* We just created the file */
      else if (r != EEXIST) return(r);  /* other error */
      else exist = !(oflags & O_EXCL);  /* file exists, if the O_EXCL
                                   flag is set this is an error */
  }
  else if (oflags & O_CREAT && oflags & O_IMM) {
      omode = I_IMMEDIATE | (omode & ALL_MODES & fp->fp_umask);
      vp = new_node(oflags, omode);
      r = err_code;
      if (r == OK) exist = FALSE; /* We just created the file */
      else if (r != EEXIST) return(r);  /* other error */
  }
  else {
      /* Scan path name */
      if ((vp = eat_path(PATH_NOFLAGS, fp)) == NULL) return(err_code);
  }

  /* Claim the file descriptor and filp slot and fill them in. */
  fp->fp_filp[m_in.fd] = fil_ptr;
  FD_SET(m_in.fd, &fp->fp_filp_inuse);
  fil_ptr->filp_count = 1;
  fil_ptr->filp_vno = vp;
  fil_ptr->filp_flags = oflags;

  /* Only do the normal open code if we didn't just create the file. */
  if(exist) {
      /* Check protections. */
      if ((r = forbidden(vp, bits)) == OK) {
            /* Opening reg. files, directories, and special files
differ */
            switch (vp->v_mode & I_TYPE) {
               case I_IMMEDIATE:
               case I_REGULAR:
                  /* Truncate regular file if O_TRUNC. */
                  if (oflags & O_TRUNC) {
                        if ((r = forbidden(vp, W_BIT)) != OK)
                              break;
                        truncate_vnode(vp, 0);
                  }

               case I_DIRECTORY:
                  /* Directories may be read but not written. */
                  r = (bits & W_BIT ? EISDIR : OK);
                  break;
               case I_CHAR_SPECIAL:
```

This handles the immediate file reading and writing. This is also where we set the maximum size of the immediate file, so we aren't overwriting the inode and causing a crash.

usr/src/servers/mfs/read.c:

```
FORWARD _PROTOTYPE( int rw_imm, (struct inode* rip, unsigned off,
size_t chunk,
                int rw_flag, cp_grant_id_t gid, unsigned buf_off));

-----------------------

PUBLIC int fs_readwrite(void)
{
  int r, rw_flag, block_spec;
  int regular, immediate;
  cp_grant_id_t gid;
  off_t position, f_size, bytes_left;
  unsigned int off, cum_io, block_size, chunk;
  mode_t mode_word;

-----------------------  mode_word = rip->i_mode & I_TYPE;
  regular = (mode_word == I_REGULAR || mode_word == I_NAMED_PIPE);
  immediate = (mode_word == I_IMMEDIATE || mode_word == I_NAMED_PIPE);
  block_spec = (mode_word == I_BLOCK_SPECIAL ? 1 : 0);

  /* Determine blocksize */
  if (block_spec) {
      block_size = get_block_size( (dev_t) rip->i_zone[0]);
      f_size = MAX_FILE_POS;
  } else {
      block_size = rip->i_sp->s_block_size;
      f_size = rip->i_size;
  }

  /* Get the values from the request message */
  rw_flag = (fs_m_in.m_type == REQ_READ ? READING : WRITING);
  gid = (cp_grant_id_t) fs_m_in.REQ_GRANT;
  position = (off_t) fs_m_in.REQ_SEEK_POS_LO;
  nrbytes = (size_t) fs_m_in.REQ_NBYTES;

  rdwt_err = OK;        /* set to EIO if disk error occurs */

  if (rw_flag == WRITING && !block_spec && (rip->i_mode & I_TYPE) !=
I_IMMEDIATE) {
        /* Check in advance to see if file will grow too big. */
        if (position > (off_t) (rip->i_sp->s_max_size - nrbytes))
              return(EFBIG);

        /* Clear the zone containing present EOF if hole about
         * to be created.  This is necessary because all unwritten
         * blocks prior to the EOF must read as zeros.
         */
        if(position > f_size) clear_zone(rip, f_size, 0);
  }

  cum_io = 0;
  if ((rip->i_mode & I_TYPE) == I_IMMEDIATE)
  {
      if (rw_flag == WRITING)
      {
            if ((position * 4 + nrbytes) > 32)
            {
                  printf("Unable to write data, too large\n");
                  exit(-1);
            }

            r = rw_imm(rip, position, nrbytes, rw_flag, gid, cum_io);
            if (r == OK)
            {
                  cum_io += nrbytes;
                  position += ceil(nrbytes/4.0);
                  nrbytes = 0;
            }
      }
      else
      {
            /*bytes_left = f_size - position;
            if (bytes_left > 0 && nrbytes > bytes_left)
            {
                  nrbytes = bytes_left;
            }*/

            r = rw_imm(rip, 0, position * 4, rw_flag, gid, cum_io);
      }
/* On write, update file size and access time. */
  if (rw_flag == WRITING) {
        if (regular || immediate || mode_word == I_DIRECTORY) {
              if (position > f_size) rip->i_size = position;
        }
  }

  /* Check to see if read-ahead is called for, and if so, set it up. */
  if(rw_flag == READING && rip->i_seek == NO_SEEK &&
     (unsigned int) position % block_size == 0 &&
     (regular || immediate || mode_word == I_DIRECTORY)) {
        rdahed_inode = rip;
        rdahedpos = position;
  }
```

This is the actual function for when an immediate file is being read from or written to.

```
/*=====================================================================
======*
 *                        rw_imm                                    *
```

```
*====================================================================
=====*/
PRIVATE int rw_imm(rip, off, chunk, rw_flag, gid, buf_off)
register struct inode* rip;
unsigned off;
unsigned int chunk;
int rw_flag;
cp_grant_id_t gid;
unsigned buf_off;
{
      int r = OK;


      printf("Inside rw_imm\n");


      if (rw_flag == READING)
      {
            r = sys_safecopyto(VFS_PROC_NR, gid, (vir_bytes) buf_off,
                  (vir_bytes)(rip->i_zone+off), (size_t)chunk, D);
      }
      else
      {
            r = sys_safecopyfrom(VFS_PROC_NR, gid, (vir_bytes) buf_off,
                  (vir_bytes)(rip->i_zone+off), (size_t)chunk, D);
            rip->i_dirt = DIRTY;
      }

      return (r);
}
```

```
-------------------------------------

  if ((rip->i_mode & I_TYPE) == I_IMMEDIATE)
      return(NO_BLOCK);
#define S_IFIMM  0050000  /* immediate */
#define S_ISIMM(m)      (((m) & S_IFMT) == S_IFIMM)   /* is a imm file
*/
```

This is where we actually set the i for ls to be desplayed

usr/src/include/sys/stat.h:

```
#define S_IFIMM  0050000  /* immediate */
```

And

```
#define S_ISIMM(m)(((m) & S_IFMT) == S_IFIMM)   /* is a imm file */
```

usr/src/commands/ls/ls.c:

```
char l_ifmt[] = "0pcCdibB-?l?s???";
```

Man Page:

Name

Immediate files – adds the ability to write to an inode if the file is small enough

Synopsis

file = open("./testfile.txt", O_CREAT + O_RDWR + O_IMM, 0755);

Options

Examples

Description

Immediate files can be used when a file is less than 40 bytes in size. An immediate file is written to an inode instead of the disk or memory like a regular file. An immediate file saves significant space with a large amount of small files. To open a file as an immediate file, sent in O_IMM as a parameter in the open command. You can do anything that you would do with a regular file in the MINIX system to an immediate file. If you try to write more than the maximum size of 40 bytes, the system will printout an error message and exits.

Tests:

Our two test files:

Test.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char **argv)
{
        int file, err;
        int iSend1 = 1234;
        int iSend2 = 5678;
        int iRcv[5];

        int iBigSend[] = {9012, 3456, 7890};
```

```
		printf("Creating immediate file\n");
		file = open("./testfile.txt", O_CREAT + O_RDWR + O_IMM, 0755);
		if (file < 0)
		{
			printf("Didn't work\n");
			exit(1);
		}

		printf("Writing %d to file", iSend1);
		err = write(file, &iSend1, 4);
		if (err == -1)
		{
			printf("Error writing\n");
		}

		err = read(file, iRcv, 4);
		if (err == -1)
		{
			printf("Error reading\n");
		}

		printf("Current data from file: %d\n", iRcv[0]);

		printf("Appending %d to file\n", iSend2);
		err = write(file, &iSend2, 4);
		if (err == -1)
		{
			printf("Error writing\n");
		}

		err = read(file, iRcv, 8);
		if (err == -1)
		{
			printf("Error reading\n");
		}

		printf("Current data from file: %d, %d\n", iRcv[0], iRcv[1]);

		printf("Appending %d, %d, %d to file\n", iBigSend[0],
iBigSend[1], iBigSend[2]);
		err = write(file, iBigSend, 12);
		if (err == -1)
		{
			printf("Error writing\n");
		}

		err = read(file, iRcv, 20);
		if (err == -1)
		{
			printf("Error reading\n");
		}

		printf("Current data from file: %d, %d, %d, %d, %d\n", iRcv[0],
iRcv[1], iRcv[2], iRcv[3], iRcv[4]);
```

```
        close (file);
}
```

Testfail.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char **argv)
{
        int file, err;
        int iBigSend[] = {9012, 3456, 7890, 1234, 1234, 1234, 1234, 1234, 1234};

        file = open("./testfile.txt", O_CREAT + O_RDWR + O_IMM, 0755);
        if (file < 0)
        {
                printf("Didn't work\n");
                exit(1);
        }

        err = write(file, iBigSend, 36);
        if (err == -1)
        {
                printf("Error writing\n");
        }

        close (file);
}
```

which generates the two output files, respectively:

output.txt:

```
Creating immediate file
Writing 1234 to file
Current data from file: 1234
Appending 5678 to file
Current data from file: 1234, 5678
Appending 9012, 3456, 7890 to file
Current data from file: 1234, 5678, 9012, 3456, 7890
```

outputfail.txt:

```
Unable to write data, too large
```