```
/*************************************************************************
*                  Tyler Gumerson - Sannidhya Malpani                    *
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>
#include <libgen.h>
#include <sys/stat.h>
#include <ext2fs/ext2_fs.h>

#include "type.h"

MINODE minode[NMINODE];
MINODE *root;
PROC   proc[NPROC], *running;

char   gpath[256]; // global for tokenized components
char   *name[64];  // assume at most 64 components in pathname
int    n;          // number of component strings

int    fd, dev;
int    nblocks, ninodes, bmap, imap, inode_start;
char   line[256], cmd[32], pathname[256], newfile[256];

#include "util.c"
#include "cd_ls_pwd.c"
#include "mkdir.c"
#include "rmdir.c"
#include "link.c"
#include "unlink.c"
#include "misc.c"
#include "opened_file.c"
#include "read_file.c"
#include "write.c"

//This function serves to initialize all the instances of the two defined datatypes minode and proc
int init()
{
  int i, j;
  MINODE *mip;
  PROC   *p;

  printf("init()\n");

  //Initializes and freeing the inodes
  //NMINODE is 64
  //minode array definded above as minode
  for (i=0; i<NMINODE; i++){
    //set mip to dereferenced minode at this i
    mip = &minode[i];
    //Initializes all of minodes values - Note: dirty not initialized to 0
    mip->dev = mip->ino = 0;
    mip->refCount = 0;
    mip->mounted = 0;
    mip->mptr = 0;
```

```
  }

  //Initializes the process (2)
  //NPROC is 2
  //proc array defined above
  for (i=0; i<NPROC; i++){
    //access pointer to process in array
    p = &proc[i];
    //Initializing the base process values
    p->pid = i;
    p->uid = 0;
    p->cwd = 0;
    p->status = FREE;
    for (j=0; j<NFD; j++)
      p->fd[j] = 0;
  }
}

// load root INODE and set root pointer to it
int mount_root()
{
  printf("mount_root()\n");
  root = iget(dev, 2);
}

char *disk = "mydisk";

int main(int argc, char *argv[ ])
{
  int ino;
  char buf[BLKSIZE];
  if (argc > 1)
    disk = argv[1];

  //Checking Disk opens properly
  printf("checking EXT2 FS ....");
  if ((fd = open(disk, O_RDWR)) < 0){
    printf("open %s failed\n", disk);  exit(1);
  }

  //Dev is pointing to the disk we read in
  dev = fd;

  /********** read super block at 1024 ****************/
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;

  /* verify it's an ext2 file system *****************/
  if (sp->s_magic != 0xEF53){
      printf("magic = %x is not an ext2 filesystem\n", sp->s_magic);
      exit(1);
  }
  printf("OK\n");

  //Geting overall inode and block count from the super block
  ninodes = sp->s_inodes_count;
  nblocks = sp->s_blocks_count;
```

```
//Read in group descriptor (2nd block) blocks 3-7 also reserved for this
get_block(dev, 2, buf);
gp = (GD *)buf;

//Get the inode map and the block map
bmap = gp->bg_block_bitmap;
imap = gp->bg_inode_bitmap;

//Get the value where the inode starts
inode_start = gp->bg_inode_table;
printf("bmp=%d imap=%d inode_start = %d\n", bmap, imap, inode_start);

init();
mount_root();

//This should always print 1 after root is initialized
printf("root refCount = %d\n", root->refCount);


printf("creating P0 as running process\n");
//Accessing first proc in process array and preparing values
running = &proc[0];
running->status = READY;
//grabbing root that was just initialized setting as cwd for this process
running->cwd = iget(dev, 2);

//This should also print 1
printf("root refCount = %d\n", root->refCount);

//printf("hit a key to continue : "); getchar();
while(1){
  printf("input command : [ls|cd|pwd|mkdir|creat|rmdir|link|symlink|unlink|quit] ");
  fgets(line, 128, stdin);
  line[strlen(line)-1] = 0;
  if (line[0]==0)
    continue;
  pathname[0] = 0;
  cmd[0] = 0;

  sscanf(line, "%s %s %s", cmd, pathname, newfile);
  printf("cmd=%s pathname=%s newfile =%s\n", cmd, pathname, newfile);

  if (strcmp(cmd, "ls")==0)
     ls_dir();
  if (strcmp(cmd, "cd")==0)
     chdir();
  if (strcmp(cmd, "pwd")==0)
     pwd(running->cwd);
  if (strcmp(cmd, "mkdir")==0)
     make_dir();
  if (strcmp(cmd, "rmdir")==0)
     rmdir();
  if (strcmp(cmd, "creat")==0)
     creat_file();
  if (strcmp(cmd, "stat")==0)
     file_stat();
```

```
    if (strcmp(cmd, "link")==0)
      link();
    if (strcmp(cmd, "symlink")==0)
      symlink();
    if (strcmp(cmd, "unlink")==0)
      unlink();
    if (strcmp(cmd, "pfd")==0)
      mypfd();
    if (strcmp(cmd, "open")==0)
      open_file();
    if (strcmp(cmd, "cat")==0)
      mycat();
    if (strcmp(cmd, "cp")==0)
      mycp();
    if (strcmp(cmd, "close")==0){
      int fdnum;
      sscanf(pathname, "%d", &fdnum);
      close_file(fdnum);
    }
    if (strcmp(cmd, "lseek")==0){
      int fdnum, offset;
      sscanf(pathname, "%d", &fdnum);
      sscanf(newfile, "%d", &offset);
      mylseek(fdnum, offset);
    }
    if (strcmp(cmd, "quit")==0)
      quit();
  }
}

//This simply puts all of the MINODES back onto the disk to make sure its all updated and closes program
int quit()
{
  int i;
  MINODE *mip;
  for (i=0; i<NMINODE; i++){
    mip = &minode[i];
    if (mip->refCount > 0)
      iput(mip);
  }
  exit(0);
}

/************* cd_ls_pwd.c file **************/

/**** globals defined in main.c file ****/
extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256];

chdir(){
```

```
  //Temp is used to copy pathname and mip used to hold dir and change
  char temp[256];
  MINODE *mip;
  int dev, ino;

  //If a path was not provided use cwd
  if (pathname[0] == 0){
     iput(running->cwd);
     running->cwd = iget(root->dev, 2);
     return;
  }

  //Use roots device number if pathname is absolute
  if (pathname[0] == '/'){
    dev = root->dev;
  } else {
    //Use cwd's device number if pathname is relative
    dev = running->cwd->dev;
  }

  //Get the inumber of the provided pathname
  strcpy(temp, pathname);
  ino = getino(temp);

  //If the ino is 0 and not found than pathname is invalid
  if (!ino){
     printf("Invalid pathname\n");
     return(-1);
  }

  //Printing inode and dev number
  printf("dev=%d ino=%d\n", dev, ino);

  //Get the MINODE of the ino number
  mip = iget(dev, ino);

  //Make sure that the path is to a directory and not a file
  if (!S_ISDIR(mip->INODE.i_mode)){
     printf("Not a directory\n");
     iput(mip);
     return(-1);
  }

  //Write back to the disk and change cwd to change the directory
  iput(running->cwd);
  running->cwd = mip;
}

int ls_file(int ino, char *name){
  MINODE *mip;

  //Get the MINODE and inode of the current ino
  mip = iget(dev, ino);
  INODE *ip = &(mip->INODE);

  //Print the ino number other stats and the name
  printf("%4d  %4d  %4d  %4d  %s\n", ino, ip->i_mode, ip->i_uid, ip->i_size, name);
```

```
  return 0;
}

int ls_dir(){
		int ino;
		MINODE *mip;
  char buf[BLKSIZE], name[256], temp[256], wd[256], *cp;
  DIR *dp;

  //Getting the path to where to ls based on pathname
  strcpy(temp, pathname);
  strcpy(wd, dirname(temp));
  ino = getino(wd);

  //Check if inode is not found
  if(ino == 0){
    printf("Error - No path");
    return 0;
  }

  //Get MINODE based on the ino of dir of pathname
  mip = iget(dev,ino);

  //Get its first block
  get_block(mip->dev, mip->INODE.i_block[0], buf);
  dp = (DIR *)buf;
  cp = buf;

  //Increment through to each entry and call ls_file
  while (cp < buf + BLKSIZE){
    //Get the ino and name to pass to ls_file
    strncpy(name, dp->name, dp->name_len);
    name[dp->name_len] = 0;
    ls_file(dp->inode, name);
    //Increment character and directory pointer
    cp += dp->rec_len;
    dp = (DIR*)cp;
  }

  //return
  return 0;
}


// Recursively prints working directory
int rpwd(MINODE *wd)
{
  char buf[BLKSIZE], name[256], *cp;
  DIR   *dp;
  MINODE *parent;
  int ino, pino;

  //If working directory is the root then return
  if (wd == root)
      return;
```

```
  //Get the first block of dir and look at second entry, ..
  get_block(dev, wd->INODE.i_block[0], buf);
  dp = (DIR *)buf;
  cp = buf;

  //Get ino using first entry
  ino = dp->inode;

  //Increment to second entry to get parent ino
  cp += dp->rec_len;
  dp = (DIR *)cp;
  pino = dp->inode;

  //Get the parent MINODE using inumber
  parent = iget(dev, pino);

  //Get the parent block to find the childs name (current wd)
  get_block(parent->dev, parent->INODE.i_block[0], buf);
  dp = (DIR *)buf;
  cp = buf;

  while (cp < buf + BLKSIZE){
  //Check to see if dirs ino is equal to current
  if(dp->inode == ino){
    strncpy(name, dp->name, dp->name_len);
    name[dp->name_len] = 0;
  }
  //Increment character and directory pointer
  cp += dp->rec_len;
  dp = (DIR*)cp;
  }

  //Recursive call on parent
  rpwd(parent);

  //Put the block back into memory and print its name
  iput(parent);
  printf("/%s", name);

  return 1;
}

int pwd(MINODE *wd){
  //If working directory is root print /
  if (wd == root){
    printf("/\n");
    return;
  }
  //If not root recursively print directories
  rpwd(wd);
  printf("\n");
}

/************* mkdir_creat.c file **************/

/**** globals defined in main.c file ****/
extern MINODE minode[NMINODE];
```

```
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256];

int make_dir(){

 //Global pathname holds the pathname to create the dir

 //Define start and parent MINDOE
 MINODE *start, *pip;
 int pino;

 char parent[64], child[64], temp[64];

 //Get parent path and new dir name
 strcpy(temp, pathname);
 //Parent directory is this part of path
 strcpy(parent, dirname(temp));
 strcpy(temp, pathname);
 //This would be the dir you are trying to make
 strcpy(child, basename(temp));

 //Determine if the path is relative or absolute
 //Define MINODE start as cwd or root, and use its correspending dev
 //This check really only matters if you are using multiple disks
 if(parent[0] == '/'){
  //Path is absolute
  start = root;
  dev = root->dev;
 }
 else{
  //Path is relative
  start = running->cwd;
  dev = running->cwd->dev;
 }

 //Getting MINODE of the parent with pino
 //Get inode of the parent (number in array minode)
 pino = getino(parent);
 //Get corresponding minode pointer to pino
 pip = iget(dev,pino);

 //Verify that parent is a directory
 if(!S_ISDIR(pip->INODE.i_mode))
 {
  printf("Error - not a directory\n");
  iput(pip);
  return;
 }

 //Check if inode for this pathname already exists (child)
 if(getino(pathname) != 0)
```

```
  {
    printf("Error - directory already exists\n");
    iput(pip);
    return;
  }

  //Call mymkdir on parent and child
  mymkdir(pip, child);

  //Incremnt links, atime and mark as dirty
  pip->INODE.i_links_count++;
  pip->INODE.i_atime = time(0L);
  pip->dirty = 1;

  //update pip on disk
  iput(pip);
}

//Takes the parent MINODE and the name of the child to be created
int mymkdir(MINODE *pip, char *name){

  //Initial Varaibales
  MINODE *mip;
  char buf[BLKSIZE];
  DIR *dp;
  char *cp;

  //Allocate an inode and disk block for the new directory
  int ino = ialloc(dev);
  int bno = balloc(dev);

  //Load inode of new directory
  //iget will create the new minode in the minode array
  mip = iget(dev,ino);
  INODE *ip = &mip->INODE;

  //Write contents to make it a dir inode
  ip->i_mode = 0x41ED;    // dir mode
  ip->i_uid  = running->uid;  // Owner uid
  //Group id is not in our process struct, this was provided
  //ip->i_gid  = running->gid;  // Group Id
  ip->i_size = BLKSIZE;   // Size in bytes
  ip->i_links_count = 2;      // Links count=2 because of . and ..
  ip->i_atime = time(0L);   // set to current time
  ip->i_ctime = time(0L);
  ip->i_mtime = time(0L);
  ip->i_blocks = 2;           // LINUX: Blocks count in 512-byte chunks
  ip->i_block[0] = bno;       // new DIR has one data block
  for(int i = 1; i < 15; i++)
    ip->i_block[i] = 0;

  //Write new inode out to disk
  mip->dirty = 1;
  iput(mip);

  // Now write write . and ..
  //dir starts at ., get its ino, name, and length
```

```
  dp = (DIR *)buf;
  dp->inode = ino;
  strncpy(dp->name, ".", 1);
  dp->name_len = 1;
  dp->rec_len = 12;

  //Set cp and increment to the next dir
  cp = buf;
  cp += dp->rec_len;

  //Now ready for next entry
  dp = (DIR *)cp;

  //Now at .. dir
  dp->inode = pip->ino;
  strncpy(dp->name, "..", 2);
  dp->name_len = 2;
  dp->rec_len = 1012;

  //Now use buf to write to disk block buf
  put_block(dev, bno, buf);

  //Enter name of the new dir into parent directory
  enter_name(pip, ino, name);
}

int creat_file(){
  //Define start and parent inode
  MINODE *start, *pip;
  int pino;

  char parent[64], child[64], temp[64];

  //Get parent path and new dir name
  strcpy(temp, pathname);
  //Parent directory is this part of path
  strcpy(parent, dirname(temp));
  strcpy(temp, pathname);
  //This would be the dir you are trying to make
  strcpy(child, basename(temp));

  //Determine if the path is relative or absolute
  //Define MINODE start as cwd or root, and declare it correspending dev
  if(parent[0] == '/'){
    //Path is absolute
    start = root;
    dev = root->dev;
  }
  else{
    //Path is relative
    start = running->cwd;
    dev = running->cwd->dev;
  }

  //Getting MINODE of the parent with pino
  //Get inode of the parent (number in array minode)
  pino = getino(parent);
```

```
  //Get corresponding minode pointer to pino
  pip = iget(dev,pino);

  //Verify that parent is a directory and no child exists
  if(!S_ISDIR(pip->INODE.i_mode))
  {
    printf("Error - not a directory\n");
    iput(pip);
    return;
  }

  if(getino(pathname) != 0)
  {
    printf("Error - directory already exists\n");
    iput(pip);
    return;
  }

  //Call create on parent and child
  my_creat(pip, child);

  //Incremnt atime and mark as dirty
  pip->INODE.i_atime = time(0L);
  pip->dirty = 1;

  //update pip on disk
  iput(pip);
}

int my_creat(MINODE *pip, char *name){

  MINODE *mip;
  char buf[BLKSIZE];
  DIR *dp;
  char *cp;
  int rec_length, myino;

  //Allocate an inode for the new file
  int ino = ialloc(dev);

  //Load inode of new directory
  mip = iget(dev,ino);
  INODE *ip = &mip->INODE;

  //Write contents to make it a file inode
  ip->i_mode = 0x81A4;    // dir mode
  ip->i_uid  = running->uid;  // Owner uid
  //ip->i_gid  = running->gid;  // Group Id
  ip->i_size = 0;   // Size in bytes
  ip->i_links_count = 1;      // Links count=1 because of ..
  ip->i_atime = time(0L);   // set to current time
  ip->i_ctime = time(0L);
  ip->i_mtime = time(0L);
  ip->i_blocks = 2;           // LINUX: Blocks count in 512-byte chunks
  for(int i = 1; i < 15; i++)
    ip->i_block[i] = 0;
```

```
  //Write new inode out to disk
  mip->dirty = 1;
  iput(mip);

  //Enter name of the file into the directory inode
  enter_name(pip,ino,name);
}

/************* rmdir.c file **************/

/**** globals defined in main.c file ****/
extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256];

int rmdir(){

        //Used to hold inumber of pathname
        int ino;
        //Used for minode of the pathname
        MINODE *mip;
        //Used to hold dir names when iterating
        char temp[256];

        //Get inumber using pathname
        ino = getino(pathname);

        //Checking to make sure pathname grabbed ino correctly
        if(ino == 0){
                printf("wrong pathname!\n");
                return -1;
        }
        //Using inumber to get it MINODE
        mip = iget(dev,ino);
        INODE *ip = &mip->INODE;

        //Check Ownership, Super User ok or uid must match
        //If uid=0 then it is super
        /*
        if(running->uid != 0 && running->uid != ip->i_uid){
                printf("Permission not granted\n");
                iput(mip);
                return -1;
        }
        */

        //Check to make sure the indode is a directory
        if(!S_ISDIR(ip->i_mode)){
                printf("Invalid pathname, not a dir!\n");
                iput(mip);
                return -1;
```

```
}


//Used to hold the blocks while searching through
char buf[BLKSIZE];
//Used to hold pointer values in the block
char *cp;
DIR *dp;

//Check to make sure the directory is empty
if(ip->i_links_count <= 2){

        //Check to make sure no entries other than . and ..
        //If this is not 0 than it is not empty
           if(mip->INODE.i_block[0] != 0){

             //Get the block and iterate over to see contents
             get_block(dev, mip->INODE.i_block[0], buf);
             //Set char and dir pointer to iterate over block
             cp = buf;
             dp = (DIR*)buf;

             while (cp < buf + BLKSIZE){
               //Copy name into temp from dir pointer
               strncpy(temp, dp->name, dp->name_len);
               temp[dp->name_len] = 0;

               //For each new dir pointer check if name is not . or ..
               if (strcmp(temp, ".") != 0 && strcmp(temp, "..") != 0){
                 //If temp is not them it is a file and dir is not empty
                 printf("Directory is not empty\n");
                 iput(mip);
                 return -1;
               }
               //Increment character and directory pointer
               cp += dp->rec_len;
               dp = (DIR*)cp;
             }
           }
} else {
        //More than 2 link count so dir is not empty
        printf("Directory is not empty\n");
        iput(mip);
        return -1;
}
//Checks Completed - Ready to remove

//Deallocate direct blocks that are not empty
for (int i=0; i<12; i++){
    if (mip->INODE.i_block[i]==0)
       continue;
    bdalloc(mip->dev, mip->INODE.i_block[i]);
}
//Deallocate the inode
idalloc(mip->dev, mip->ino);
iput(mip); //(which clears mip->refCount = 0);
```

```
    //Used to hold the pathname of parent and child in provided path
            char parent[256], child[256];
            //MINODE for parent
            MINODE *pip;

    //Set the parent and child paths from pathname, to get their inumbers
            strcpy(temp, pathname);
            strcpy(parent, dirname(temp));
            strcpy(temp, pathname);
            strcpy(child, basename(temp));

            //Getting parent ino number and MINODE
            int pino = getino(parent);
            pip = iget(mip->dev,pino);
            rm_child(pip, child);

            //Decrement link count, touch atime and mtime, mark dirty
            pip->INODE.i_links_count--;
            pip->dirty = 1;
            pip->INODE.i_atime = time(0L);   // set to current time
            pip->INODE.i_mtime = time(0L);
            iput(pip);

            //0 mean success!
            return 0;
}

// rm_child(): remove the entry [INO rlen nlen name] from parent's data block.
int rm_child(MINODE *parent, char *name){

            //Used to hold the blocks and dir names while searching through

    char buf2[BLKSIZE], temp2[256];
    //Used to hold pointer values in the block
    char *cp;
    char *prevcp;
    DIR *dp;

    int curr_rlen, prev_rlen;
            //Search parent inodes data blocks for the entry of name
            for(int i = 0; i < 12; i ++){

                    //Check to make sure no entries other than . and ..
                    //If this is not 0 than it is not empty
               if(parent->INODE.i_block[i] != 0){

                 //Get the block and iterate over to see contents
                 get_block(dev, parent->INODE.i_block[i], buf2);
                 //Set char and dir pointer to iterate over block
                 cp = buf2;
                 prevcp = buf2;
                 dp = (DIR*)buf2;
                 while (cp < buf2 + BLKSIZE){
                   //Copy name into temp from dir pointer
                   strncpy(temp2, dp->name, dp->name_len);
                   temp2[dp->name_len] = 0;
```

```
        //Check to see if dp is pointing to correct child
if(strcmp(name,temp2) == 0){
        //Set curr_rlen so you can add to previous dp
        curr_rlen = dp->rec_len;

        //We know dir pointer is the correct child
        //Check to see if this is the last dir in the block
        if(cp + dp->rec_len >= buf2 + BLKSIZE){
                //Back up to the previous dp
                cp -= prev_rlen;
                dp = (DIR *)cp;
                //Add the current to previous rlen
                dp->rec_len += curr_rlen;

                //Write the parent black to the disk and mark dirty
                put_block(parent->dev, parent->INODE.i_block[i], buf2);
                parent->dirty = 1;
                return 0;
        }

        //Check to see if this is the first dir in the block
        if(dp->rec_len == BLKSIZE){
                //deallocate the current block
                bdalloc(parent->dev, parent->INODE.i_block[i]);
                //Set the iblock that held its name to zero and mark as dirty
                parent->INODE.i_block[i] = 0;
                parent->dirty = 1;

                //Shift nonempty blocks upward so there are no holes
                for(int j = i; j < 11; j++){
                        //Checking to see if next inode is empty
                        if(parent->INODE.i_block[j+1] != 0){
                                //If it is not empty move block up and increment
                                parent->INODE.i_block[j] = parent->INODE.i_block[j+1];
                                //Set this block to empty after moving
                                parent->INODE.i_block[j+1] = 0;
                        } else {
                                break;
                        }
                }
                return 0;
        }

        //We now know that the dir is in the middle of the block
        //Hold length of current dir to remove
        curr_rlen = dp->rec_len;

        //Iterate through the other directories shifting to the left
        while(cp < buf2 + BLKSIZE){
                        //Increment directory and cp
                        cp += dp->rec_len;
                        dp = (DIR *)cp;

                        //Use memove to shift current dp to the left
                        memmove(prevcp, cp, dp->rec_len);

                        //check if this is last dir
```

```
                                        if(cp + dp->rec_len >= BLKSIZE){
                                                dp = (DIR *)prevcp;
                                                dp->rec_len += curr_rlen;
                                                break;
                                        }

                                        //Increment prevcp
                                        prevcp += dp->rec_len;
                                }
                        //Write the parent block to the disk and mark dirty
                        put_block(parent->dev, parent->INODE.i_block[i],buf2);
                        parent->dirty = 1;
                        return 0;
                    }
                    //Update the previous rec len
                    prev_rlen = dp->rec_len;
                    //Increment character and directory pointer
                    cp += dp->rec_len;
                    prevcp += dp->rec_len;
                    dp = (DIR*)cp;
                  }
                } else {
                        break;
                }
            }
}

/************ link.c file **************/

/**** globals defined in link.c file ****/
extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256], newfile[256];


int link(){
  char parent[128], temp[128], child[128];

  //Setting paths for parent and new file
  strcpy(temp, newfile);
  strcpy(parent, dirname(temp));
  strcpy(temp, newfile);
  strcpy(child, basename(temp));

  //MINODES for old file and parent dir
  MINODE *mip, *pmip;

  //Get the ino for file we want to link to
  //get inumber of the pathname
  int ino = getino(pathname);
```

```
  //Check that this found a valid ino
  if(ino == 0){
    printf("Invalid Path\n");
    return -1;
  }

  //Get MINODE for this files ino
  mip = iget(dev, ino);
  INODE *ip= &mip->INODE;

  //Check file is not a dir
  if(S_ISDIR(ip->i_mode)){
    printf("Cannot Link to Directory\n");
    //Put minode back and return
    iput(mip);
    return -1;
  }

  //Get inode for dir where link will be created
  int pino = getino(parent);

  //Check that this is a valid directory
  if(pino == 0){
    printf("Invalid Path\n");
    iput(mip);
    return -1;
  }

  //Get MINODE for the parent dir
  pmip = iget(dev, pino);

  //Make sure the new file path does not exist
  if (search(pmip, child) != 0){
    printf("File already exists\n");
    iput(mip);
    iput(pmip);
    return -1;
  }

  //Create the new file link
  enter_name(pmip, ino, child);

  //Parent MINODE touch time and make dirty
  pmip->INODE.i_atime = time(0L);
  pmip->dirty = 1;

  //Put parent MINODE back
  iput(pmip);

  //increment links count by 1 for the old file
  ip->i_links_count++;

  //Put back old file MINODE
  iput(mip);
}

int symlink(){
```

```
MINODE *mip, *pmip;
int len;
char parent[128], temp[128], child[128];

//Setting pathname for child and parent
strcpy(temp, newfile);
strcpy(parent, dirname(temp));
strcpy(temp, newfile);
strcpy(child, basename(temp));

//Get the ino for file we want to link to
//get inumber of the pathname
int ino = getino(pathname);

//Check that this found a valid ino
if(ino == 0){
  printf("Invalid Path\n");
  return -1;
}

//Get MINODE for this files ino
mip = iget(dev, ino);

//Check file is a dir or reg file
if(!S_ISDIR(mip->INODE.i_mode) && !S_ISREG(mip->INODE.i_mode)){
  printf("Needs to be a directory or file\n");
  //Put minode back and return
  iput(mip);
  return -1;
}

//Put old file mip back
iput(mip);

//Get parent inode of newfile
int pino = getino(parent);
pmip = iget(dev, pino);
INODE *pip = &pmip->INODE;

//Make sure parent path for new file is a dir
if(!S_ISDIR(pip->i_mode)){
  printf("Invalid Path\n");
  iput(pmip);
  return -1;
}

//Make sure a file does not already exist
if(search(pmip, child) != 0){
  printf("File already exists\n");
  iput(pmip);
  return -1;
}

//Create new inode for file
//create(pmip, child);
ino = ialloc(dev);
```

```
  //Get MINODE of the new file
  mip = iget(dev, ino);
  INODE *ip = &mip->INODE;
  len = strlen(pathname);

  //Write contents to make it a link inode
  ip->i_mode = 0120000;    // link mode
  ip->i_uid  = running->uid;  // Owner uid
  //ip->i_gid  = running->gid;  // Group Id
  ip->i_size = len;   // Size in bytes
  ip->i_links_count = 1;      // Links count=1 because of ..
  ip->i_atime = time(0L);   // set to current time
  ip->i_ctime = time(0L);
  ip->i_mtime = time(0L);
  ip->i_blocks = 2;
  //Copy pathname into inode i_block
  strncpy((char *)mip->INODE.i_block, pathname, len);
  for(int i = 1; i < 15; i++)
    ip->i_block[i] = 0;

  //Write new inode out to disk
  mip->dirty = 1;
  iput(mip);

  //Enter name of the file into the directory inode
  enter_name(pmip,ino,newfile);

  //Touch the parent inode and put it back
  pip->i_atime = time(0L);
  pmip->dirty = 1;
  iput(pmip);

  //write back to disk
  iput(mip);
}

/************* link.c file **************/

/**** globals defined in link.c file ****/
extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256];

int unlink(){

  MINODE *mip, *pip;
  char parent[64], child[32], temp[64];

  //Check to make sure pathname is provided
  if(strcmp(pathname, "") == 0){
```

```
  printf("Error - No Pathname\n");
  return -1;
 }

 //Get the MINODE of the provided pathname
 int ino = getino(pathname);

 //Check to make sure the ino is valid
 if(ino == 0){
  printf("Invalid Pathname\n");
  return -1;
 }

 //Get the MINODE and inode of the pathname
 mip = iget(dev, ino);

 //Check to make sure the inode is a file
 if(S_ISDIR(mip->INODE.i_mode))
 {
  printf("Error - Cannot unlink directory\n");
  iput(mip);
  return -1;
 }

 //Ready to unlink the file
 //Decrement link count of the pathname inode
 mip->INODE.i_links_count--;

 //If link count is 0 free the inode and its blocks
 if(mip->INODE.i_links_count == 0){
  truncate(mip);
  idalloc(mip->dev,mip->ino);
 }

 //Put mip back to the blocks
 iput(mip);

 //Get the parent and child path based on the pathname
 strcpy(temp, pathname);
 strcpy(parent, dirname(temp));
 strcpy(temp, pathname);
 strcpy(child, basename(temp));

 printf("\nparent: %s child: %s\n",parent,child);

 //Get the parent MINODE of the file we are unlinking
 int pino = getino(parent);
 pip = iget(dev, pino);

 //Call rm_child on this dir for this child
 rm_child(pip, child);

 //Put pip back to the blocks
 iput(pip);
}

extern MINODE minode[NMINODE];
```

```
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char   line[256], cmd[32], pathname[256], newfile[256];

void file_stat(){

        struct stat myst;

   //get INODE of filename into memory:
   int ino = getino(pathname);
   MINODE *mip = iget(dev, ino);
   myst.st_dev = mip->dev;
        myst.st_ino = ino;

        //Add the stats of the inode
        INODE *ip = &mip->INODE;
        //Write contents to make it a dir inode
        myst.st_mode = ip->i_mode;              // dir mode
        myst.st_uid = ip->i_uid;        // Owner uid
        //ip->i_gid  = running->gid;    // Group Id
        myst.st_size = ip->i_size;              // Size in bytes
        //myst.st_links_count = ip->i_links_count;          // Links count=1 because of ..
        myst.st_atime = ip->i_atime;    // set to current time
        myst.st_ctime = ip->i_ctime;
        myst.st_mtime = ip->i_mtime;
        myst.st_blocks = ip->i_blocks;

        //Put mip back
   iput(mip);

   //print the stats to console
   printf("dev: %d\n", myst.st_dev);
   printf("ino: %d\n", myst.st_ino);
   printf("mode: %d\n", myst.st_mode);
   printf("uid: %d\n", myst.st_uid);
   printf("Size: %d\n", myst.st_size);
   printf("i_atime: %d\n", myst.st_atime);
   printf("i_ctime: %d\n", myst.st_ctime);
   printf("i_mtime: %d\n", myst.st_mtime);
   printf("i_blocks: %d\n", myst.st_blocks);
}

/********** util.c file ***************/

/**** globals defined in main.c file ****/
extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;

extern char   gpath[256];
extern char   *name[64];
extern int    n;
```

```
extern int    fd, dev;
extern int    nblocks, ninodes, bmap, imap, inode_start;
extern char   line[256], cmd[32], pathname[256];

//Used to grab block of data from the device
int get_block(int dev, int blk, char *buf){
  //Get to the specified block
  lseek(dev, (long)blk*BLKSIZE, 0);
  //Read the specified block into buf
  read(dev, buf, BLKSIZE);
}

int put_block(int dev, int blk, char *buf){
  //Same as above but write to the provided block using buf
  lseek(dev, (long)blk*BLKSIZE, 0);
  write(dev, buf, BLKSIZE);
}

//Tokenizing pathname into name and setting n
int tokenize(char *line)
{
  // tokenize pathname in GLOBAL gpath[]; pointer by name[i]; n tokens
    int i = 0;
    char *token = strtok(line, "/");
    while(token != NULL){
        name[i] = token;
        token = strtok(NULL, "/");
        i++;
    }
    name[i] = 0;
    n = i;
    return 0;
}

//Return minode pointer to loaded INODE
MINODE *iget(int dev, int ino){
  int i;
  MINODE *mip;
  char buf[BLKSIZE];
  int blk, disp;
  INODE *ip;

  for (i=0; i<NMINODE; i++){
    mip = &minode[i];

    //This will not be true when usinig function for root or if inode to get does not exist (dev & ino)
    if (mip->refCount && mip->dev == dev && mip->ino == ino){
       mip->refCount++;
       //printf("found [%d %d] as minode[%d] in core\n", dev, ino, i);
       return mip;
    }
  }

  //This for the case of createing root or when an inode with dev & ino passed does not exist
  for (i=0; i<NMINODE; i++){
    mip = &minode[i];
```

```
    //Checking that this is an inode that has actually been initialized
    if (mip->refCount == 0){
      //printf("allocating NEW minode[%d] for [%d %d]\n", i, dev, ino);
      //Initializing ref count of inode to 1 and initializing dev and ino based on values passed
       mip->refCount = 1;
       mip->dev = dev;
       mip->ino = ino;

       //Properly set block and displacement for the ino (ex. 2 would be 2nd inode)
       // get INODE of ino to buf
       blk  = (ino-1) / 8 + inode_start;
       disp = (ino-1) % 8;

       //printf("iget: ino=%d blk=%d disp=%d\n", ino, blk, disp);

       //Reading inode pointer from dev and initializing it inside of minode
       get_block(dev, blk, buf);
       ip = (INODE *)buf + disp;
       // copy INODE to mp->INODE
       mip->INODE = *ip;

       return mip;
    }
  }
  printf("PANIC: no more free minodes\n");
  return 0;
}

//Does this just update MINODE based on the inode pointer?
int iput(MINODE *mip){

  int i, block, offset;
  char buf[BLKSIZE];
  INODE *ip;

  //If the mip is 0 then just return as cannot add to array of minodes
  if (mip==0)
   return;

  //Why do we decrease ref count here? - to show this user done with it
  mip->refCount--;

  if (mip->refCount > 0) return;
  if (!mip->dirty)       return;

  /* write back */
  printf("iput: dev=%d ino=%d\n", mip->dev, mip->ino);

  block =  ((mip->ino - 1) / 8) + inode_start;
  offset =  (mip->ino - 1) % 8;

  /* first get the block containing this inode */
  get_block(mip->dev, block, buf);

  ip = (INODE *)buf + offset;
  *ip = mip->INODE;
```

```
  put_block(mip->dev, block, buf);
}

//Takes MINODE to see if the name is a child in the directory, return 0 if not
int search(MINODE *mip, char *name)
{
  //Used to hold the names and test them
  char sbuf[1024], temp[256];
  char *cp;
  //Set ip as the actual inode pointer associated with MINODE
  INODE *ip = &(mip->INODE);
  fprintf(stderr,"name %s\n",name);

  for (int i=0; i < 12; i++){  // assume DIR at most 12 direct blocks
    printf("In for loop\n");
    //mip = iget(dev, ino);

    //checking if this block is free, end loop no more children
    if (ip->i_block[i] == 0){
      break;
    }
    // YOU SHOULD print i_block[i] number here
    printf("In Search, IP block number:%d \n",i);

    //Read the current block into buf if it is not 0
    get_block(dev,ip->i_block[i], sbuf);

    //Create a directory and char pointer based off what was read into buf(from inode block)
    dp = (DIR *)sbuf;
    cp = sbuf;

    //Using char pointer to search the current direct block of this dir inode
    while(cp < sbuf + 1024){

      //Copy the name and its length from the directory pointer
      strncpy(temp, dp->name, dp->name_len);
      //Set the end of the name to 0 or null
      temp[dp->name_len] = 0;

      //If this directory name equals the child then it is found
      if(strcmp(temp,name)==0){
        fprintf(stderr,"%s found! \n",temp);
        //Return the inode of this directory pointer
        return dp->inode;
      }

      //This is incrmenting cp to the next record of this directory pointer
      cp += dp->rec_len;
      //Update the directory pointer based of the new cp
      dp = (DIR *)cp;
    }
  }

  //The child was not found in any of the direct blocks of this inode so return 0
  fprintf(stderr,"%s not found! \n",temp);
  return 0;
```

```
}

//Returning ino number (one of the 64 inodes) for this pathname, return 0 if does not exist
int getino(char *pathname)
{
  int i, ino, blk, disp;
  INODE *ip;
  MINODE *mip;

  printf("getino: pathname=%s\n", pathname);

  //If the case is we are searching for root just return root, root always = ino 2
  if (strcmp(pathname, "/")==0)
      return 2;

  //Now we check if this is an absolute path
  if (pathname[0]=='/')
    //If it is absolute set mip equal to the root MINODE
    mip = iget(dev, 2);
  else
    //Otherwise set mip equal to the cwd MINODE
    mip = iget(running->cwd->dev, running->cwd->ino);

  //This tokenizes the pathname with / and stores the tokens in name, n is global number of tokens
  tokenize(pathname);

  //Looping through the tokens
  for (i=0; i<n; i++){
      printf("===========================================\n");

      //Gets the inode number for the next inode in the path
      ino = search(mip, name[i]);

      //Case when path is invalid one of the names do not exist
      if (ino==0){
        iput(mip);
        printf("name %s does not exist\n", name[i]);
        return 0;
      }

      //if ino is valid update mip to this child
      iput(mip);
      mip = iget(dev, ino);

      //WHY do we use iput in both of these cases? - to update disk
    }
    iput(mip);
    //return the inode number of the last dir it ended on correctly
    return ino;
}

int tst_bit(char *buf, int bit)
{
  int i, j;
  i = bit/8; j=bit%8;
  if (buf[i] & (1 << j))
      return 1;
```

```
  return 0;
}

int set_bit(char *buf, int bit)
{
  int i, j;
  i = bit/8; j=bit%8;
  buf[i] |= (1 << j);
}

int clr_bit(char *buf, int bit)
{
  int i, j;
  i = bit/8; j=bit%8;
  buf[i] &= ~(1 << j);
}


int incFreeInodes(int dev)
{
  char buf[BLKSIZE];

  // inc free inodes count in SUPER and GD
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  sp->s_free_inodes_count++;
  put_block(dev, 1, buf);

  get_block(dev, 2, buf);
  gp = (GD *)buf;
  gp->bg_free_inodes_count++;
  put_block(dev, 2, buf);
}

int decFreeInodes(int dev)
{
  char buf[BLKSIZE];
  // dec free inodes count by 1 in SUPER and GD
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  sp->s_free_inodes_count--;
  put_block(dev, 1, buf);

  get_block(dev, 2, buf);
  gp = (GD *)buf;
  gp->bg_free_inodes_count--;
  put_block(dev, 2, buf);
}

int decFreeBlocks(int dev)
{
  char buf[BLKSIZE];
  // dec free inodes count by 1 in SUPER and GD
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  sp->s_free_blocks_count--;
  put_block(dev, 1, buf);
```

```
  get_block(dev, 2, buf);
  gp = (GD *)buf;
  gp->bg_free_blocks_count--;
  put_block(dev, 2, buf);
}

int incFreeBlocks(int dev)
{
  char buf[BLKSIZE];

  // dec free inodes count in SUPER and GD
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  sp->s_free_blocks_count++;
  put_block(dev, 1, buf);

  get_block(dev, 2, buf);
  gp = (GD *)buf;
  gp->bg_free_blocks_count++;
  put_block(dev, 2, buf);
}

int ialloc(int dev)  // allocate an inode number
{
  int  i;
  char buf[BLKSIZE];

  // read inode_bitmap block
  get_block(dev, imap, buf);

  for (i=0; i < ninodes; i++){
    if (tst_bit(buf, i)==0){
       set_bit(buf,i);
       put_block(dev, imap, buf);
       decFreeInodes(dev);
       return i+1;
    }
  }
  return 0;
}

// SAME AS IALLOC BUT FOR BLOCKS
int balloc(int dev)
{
    int i;
    char buf[BLKSIZE];
    // read block bitmap into buf
    get_block(dev, bmap, buf);

    for(i = 0; i < nblocks; i++)
    {
        // allocate a free block
        if(tst_bit(buf, i)==0)
        {
            set_bit(buf, i);
            decFreeBlocks(dev);
```

```
            put_block(dev, bmap, buf);
            // return i which is its block number
            return i;
        }
    }
    return 0;
}

int idalloc(int dev, int ino)  // deallocate an ino number
{
  int i;
  char buf[BLKSIZE];

  if (ino > ninodes){
    printf("inumber %d out of range\n", ino);
    return;
  }

  // get inode bitmap block
  get_block(dev, bmap, buf);
  clr_bit(buf, ino-1);

  // write buf back
  put_block(dev, imap, buf);

  // update free inode count in SUPER and GD
  incFreeInodes(dev);
}

int bdalloc(int dev, int blk) // deallocate a blk number
{
  int i;
  char buf[BLKSIZE];

  if (blk > nblocks){
    printf("blknumber %d out of range\n", blk);
    return;
  }

  // get inode bitmap block
  get_block(dev, bmap, buf);
  clr_bit(buf, blk-1);

  // write buf back
  put_block(dev, bmap, buf);

  // update free block count in SUPER and GD
  incFreeBlocks(dev);
}

//Add the file or directory to the first available block with space, create new block in parent dir if necessary
int enter_name(MINODE *pip, int myino, char *myname){
  int ideal_len, remain;
  char buf[BLKSIZE];
  DIR *dp;
  char *cp;
```

```
//Calculate length of the name
MINODE *parent = pip;
int n_len = strlen(myname);

//Calculate the lenght we need based on name
int need_length = 4 * ((8 + n_len + 3) / 4); // a multiple of 4

//Search through blocks of the inode
for(int i = 0; i < 12; i++){

  //If block is not allocated stop
  if(parent->INODE.i_block[i] == 0){
    return -1;
  }

  // parents ith block
  get_block(parent->dev, parent->INODE.i_block[i], buf);

  //Set the dp and cp to look through this blocks
  dp = (DIR *)buf;
  cp = buf;

  //Increment through the directories in this block
  while(cp + dp->rec_len < buf + BLKSIZE)
  {
    cp+= dp->rec_len;
    dp = (DIR *)cp;
  }

  //Set ideal length based on final dir
  ideal_len = 4 * ((8 + dp->name_len + 3) / 4);

  //Get remaining length in the block
  remain = dp->rec_len - ideal_len;

  //If enough room is left in the block the dir can be added
  if(remain >= need_length){
    //enter the new entry as the LAST entry and trim the previous entry
    //to its IDEAL_LENGTH;

    //Trim previous dir, current dp to ideal length
    dp->rec_len = ideal_len;

    //Increment dp and cp for the new dir
    cp += dp->rec_len;
    dp = (DIR *)cp;

    //Add the new dir or file
    dp->rec_len = remain;
    dp->inode = myino;
    dp->name_len = n_len;
    strncpy(dp->name, myname, n_len);
  }
  else{
    //Need to create a new block to store dir or file

    //Allocate a new data block
```

```
      int bno = balloc(parent->dev);

      //Increase inode size of the parentquit
      parent->INODE.i_size += 1024;

      //Set the next block to this new block, if i is 12 no more direct blocks
      if(i < 11){
        parent->INODE.i_block[i+1] = bno;
      }

      //Get the new block into buf
      get_block(parent->dev, parent->INODE.i_block[i+1], buf);
      dp = (DIR *)buf;

      //Add the new dir or file
      dp->rec_len = BLKSIZE;
      dp->inode = myino;
      dp->name_len = n_len;
      strncpy(dp->name, myname, n_len);
    }

    // write block back to disk
    put_block(parent->dev, parent->INODE.i_block[i], buf);
  }
}

int truncate(MINODE *mip){
  //Go through the direct blocks and deallocate
  for (int i = 0; i < 12; i++){
    // If block is already free, go on to next iteration
    if (mip->INODE.i_block[i] == 0) {
      continue;
    }
    //Deallocate this block
    bdalloc(mip->dev, mip->INODE.i_block[i]);
  }
  //Deallocate the indirect blocks if necessary
  if(mip->INODE.i_block[12] != 0){
    int ibuf[256];
    get_block(mip->dev, mip->INODE.i_block[12], ibuf);
    // indirect has 256 blocks
    for(int i = 0; i < 256; i++) {
      //If block is empty continue
      if(ibuf[i] == 0)
        continue;
      //If it is not empty deallocate a block
      bdalloc(mip->dev, ibuf[i]);
    }
  }
  //Deallocate double indirect blocks if necessary
  if(mip->INODE.i_block[13] != 0) {
    int ibuf[256];
    int tempBuf[256];
    get_block(mip->dev, mip->INODE.i_block[13], ibuf);
    //256 indirect blocks
    for(int i = 0; i < 256; i++) {
      if(ibuf[i] != 0){
```

```
        get_block(mip->dev, ibuf[i], tempBuf);
        //Each 256 indirect has 256 double indirect blocks
        for(int j = 0; j < 256; j++) {
          if(tempBuf[j] == 0)
            continue;
          bdalloc(mip->dev, tempBuf[j]);
        }
        bdalloc(mip->dev, ibuf[i]);
      }
    }
  }
  // touch mips time and set it dirty
  mip->INODE.i_atime = time(0L);
  mip->INODE.i_mtime = time(0L);
  mip->INODE.i_ctime = time(0L);
  mip->INODE.i_size = 0;
  mip->dirty = 1;
}

extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256], newfile[256];

int open_file(){
    int mode, ino;
    MINODE *mip;
    OFT *oft;

    //extracting the mode
    sscanf(newfile,"%d", &mode);
    sscanf("newfile :%d", &mode);

    //set the dev
    //if path starts from root
    if(pathname[0] == '/'){
        dev = root->dev;
    }
    //if path starts from relative
    else{
        dev = running->cwd->dev;
    }

    //get ino number for the path
    ino= getino(pathname);

    //return if invalid path
    if(ino==0){
        printf("Invalid pathname \n");
        return -1;
    }
```

```
//Get the minode of the path
mip = iget(dev, ino);
INODE *ip = &mip->INODE;

//Check to make sure it is a regular file
if(!S_ISREG(ip->i_mode)){
    printf("Not a regular file \n");
    iput(mip);
    return -1;
}

//Check if file is already opened in incompatible mode
for(int i = 0; i < NFD; i++){
    //Check for this mptr to see if its already opened
    if(running->fd[i]!=NULL && running->fd[i]->mptr == mip){
        //Check to see if its opened in a mode other than read
        if(running->fd[i]->mode > 0 || mode > 0){
            printf("file already opened for writing \n");
            iput(mip);
            return -1;
        }
        //Check to see if both are to be opened for read
        if(running->fd[i]->mode == 0 || mode == 0){
            //Increment refcount and return
            running->fd[i]->refCount++;
            iput(mip);
            return i;
        }
    }
}

//Allocate a new open file table
oft = malloc(sizeof(OFT));
//Set the initial values
oft->mode = mode;
oft->refCount = 1;
oft->mptr = mip;


switch(mode){
    //read
    case 0:
        oft->offset = 0;
        break;
    //write
    case 1:
        truncate(mip);
        oft->offset = 0;
        break;
    //read and write
    case 2:
        oft->offset = 0;
        break;
    //append
    case 3:
        oft->offset = ip->i_size;
        break;
```

```
        default:
            printf("Invalid mode! \n");
            iput(mip);
            return -1;
    }

    //Look for an empty process slot
    int i = 0;
    for(i = 0 ; i < NFD; i++){
        if(running->fd[i] == NULL){
            break;
        }
    }

    //Put the oft into the empty slot
    running->fd[i] = oft;

    //if mode is read, then just change the access time
    if(oft->mode == 0){
        ip->i_atime = time(0L);
    }

    //if mode is other than read mode
    else if(oft->mode > 0){
        ip->i_atime = time(0L);
        ip->i_mtime = time(0L);
    }

    //mark the file dirty
    mip->dirty = 1;

    //put minode back on disk
    iput(mip);
    printf("file opened");
    mypfd();
    return i;
}

int mypfd(){
  printf("     OPENED FILES:     \n");
  printf("fd  mode    offset  INODE     refcounts\n");
  //Search through fd to find any OFT's
  for(int i = 0; i < NFD; i++){
    char mode[8];

    //Check OFT exist and convert mode from int
    if(running->fd[i] != NULL) {
      if(running->fd[i]->mode == 0)
        strcpy(mode, "READ");
      else if(running->fd[i]->mode == 1)
        strcpy(mode, "WRITE");
      else if(running->fd[i]->mode == 2)
        strcpy(mode, "R/W");
      else if(running->fd[i]->mode == 3)
        strcpy(mode, "APPEND");

      //Print all the info for the OFT
```

```
      printf("%d   %s    %d      [%d, %d]    %d\n", i, mode, running->fd[i]->offset,
        running->fd[i]->mptr->dev, running->fd[i]->mptr->ino, running->fd[i]->refCount);
    }
  }
}

int close_file(int fdnum){
    MINODE *mip;

    //Check if fd is in the valid range and exists
    if(fdnum < 0 || fdnum > NFD || running->fd[fdnum] == NULL){
        printf("fd out of range or does not exist \n");
        return -1;
    }

    //Check if this is the last reference to OFT
    if(running->fd[fdnum]->refCount == 1){
        //Last one so remove OFT
        running->fd[fdnum]->refCount--;
        mip = running->fd[fdnum]->mptr;
        iput(mip);
        running->fd[fdnum] = 0;

    } else {
        //Decrement refCounts
        running->fd[fdnum]->refCount--;
        return 0;
    }

    mypfd();
}

int mylseek(int fdnum, int position){

    int op;

    //Check to make sure OFT exist and is valid
    if(running->fd[fdnum] != NULL && fdnum < 8){

        //Check to make sure it does not overrun end of file
        if(running->fd[fdnum]->mptr->INODE.i_size < position){
            printf("The offset is greater than file size\n");
            return -1;
        }

        //Save the original offset and set it to position
        op = running->fd[fdnum]->offset;
        running->fd[fdnum]->offset = position;

    } else {
        printf("Invalid lseek call \n");
        return -1;
    }

    return op;
}
```

```
extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256], newfile[256];

int myread(int fd, char *buf, int nbytes){

 int count = 0,blk;
 int remain = 0;
 OFT *oftp;
 char readbuf[BLKSIZE];
 oftp = running->fd[fd];
 MINODE *mip = oftp->mptr;
 INODE *ip = &mip->INODE;


 int fileSize = ip->i_size;
 //number of bytes still available for read
 int avil = fileSize - running->fd[fd]->offset;
 //cq points at buf[ ]
 char *cq = buf;



 while (nbytes && avil){
  //Compute LOGICAL BLOCK number lbk and startByte in that block from offset;
  int lbk       = oftp->offset / BLKSIZE;
  int startByte = oftp->offset % BLKSIZE;

  // I only show how to read DIRECT BLOCKS. YOU do INDIRECT and D_INDIRECT

  if (lbk < 12){                    // lbk is a direct block
    blk = ip->i_block[lbk]; // map LOGICAL lbk to PHYSICAL blk
  }
  else if (lbk >= 12 && lbk < 256 + 12) {
    //indirect blocks
    //Load indirect blocks into ibuf
    int ibuf[256];
    get_block(mip->dev, ip->i_block[12], ibuf);

    //Grab indirect based on lbk
    blk = ibuf[lbk-12];
  }
  else{
    //double indirect blocks
    //Load double indirect blocks into buf
    int ibuf[256], dibuf[256];
    get_block(mip->dev, ip->i_block[13], dibuf);

    //Load corresponding indirect block, diblk (double indirect we are on) - iblk (indirect block we are on)
    int dilbk = (lbk-12-256) / 256;
    int ilbk = (lbk-12-256) % 256;
```

```
      //load in ibuf
      get_block(mip->dev, dibuf[dilbk], ibuf);

      //set blk value
      blk = ibuf[ilbk];
    }

    /* get the data block into readbuf[BLKSIZE] */
    get_block(mip->dev, blk, readbuf);

    //printf("\nreadbuf: %s", readbuf);

    /* copy from startByte to buf[ ], at most remain bytes in this block */
    char *cp = readbuf + startByte;
    remain = BLKSIZE - startByte;   // number of bytes remain in readbuf[]

    while (remain > 0){
      *cq++ = *cp++;             // copy byte from readbuf[] into buf[]
      oftp->offset++;           // advance offset
      count++;                  // inc count as number of bytes read
      avil--; nbytes--;  remain--;
      if (nbytes <= 0 || avil <= 0)
        break;
    }

    // if one data block is not enough, loop back to OUTER while for more ...
  }

  //printf("myread: read %d char from file descriptor %d\n", count, fd);

  return count;   // count is the actual number of bytes read
}

int mycat(){

  char mybuf[1024], dummy = 0;  // a null char at end of mybuf[ ]
  int n;

  //Set newfile to read for the open function
  strcpy(newfile,"0");

  //open the file and return the index in fd for proc
  int fdnum = open_file();

  //file doesnt exist
  if (fdnum == -1){
     printf("Invalid path, no such files exist \n");
     return -1;
  }

  printf("\nStart of file: \n");

  //reading
  int i = 0;
  n = myread(fdnum, mybuf, BLKSIZE);
  while(n > 0){
```

```
      mybuf[n] = 0;
      while(i<n){
         printf("%c", mybuf[i]);
         i++;
      }
      i = 0;
      n = myread(fdnum, mybuf, BLKSIZE);
   }

   printf("\nEnd of file \n");

   //Close the file
   close_file(fdnum);
}

extern MINODE minode[NMINODE];
extern MINODE *root;
extern PROC   proc[NPROC], *running;
extern char   gpath[256];
extern char   *name[64];
extern int    n;
extern int fd, dev;
extern int nblocks, ninodes, bmap, imap, inode_start;
extern char line[256], cmd[32], pathname[256], newfile[256];

int mycp(){
    char buf[BLKSIZE];

    //get source files inode number
    int src_ino = getino(pathname);
    //get destination's inode number
    int dest_ino = getino(newfile);

    //check that destination does not exist
    char temp_holder[256];
    if (dest_ino == 0){
        // save the original file, since creat uses pathname
        strcpy(temp_holder, pathname);
        // copy destination file's path
        strcpy(pathname,newfile);
        // create the empty file
        creat_file();
        // reset path
        strcpy(pathname, temp_holder);
    }

    //open source file
    //save the destination file pathname
    strcpy(temp_holder,newfile);
    //set this to read mode for open call
    strcpy(newfile,"0");
    //open source file to read
    int fdnum = open_file();
    //reset newfile
    strcpy(newfile,temp_holder);

    //open destination file to write
```

```
    strcpy(temp_holder,pathname);
    strcpy(pathname,newfile);
    //set to write mode for the open call
    strcpy(newfile,"2");
    int gd = open_file();
    strcpy(newfile,pathname);
    strcpy(pathname,temp_holder);

    //read in n bytes to buf from old file, then write it to newfile
    while (n = myread(fdnum, buf, BLKSIZE)){
        mywrite(gd, buf, n);
    }

    //close the files
    close_file(fdnum);
    close_file(gd);

}

int mywrite(int fd, char *buf, int nbytes){

  int count = 0,blk;
  int remain = 0;
  OFT *oftp;
  char wbuf[BLKSIZE];
  oftp = running->fd[fd];
  MINODE *mip = oftp->mptr;
  INODE *ip = &mip->INODE;
  char *cq = buf;


    while(nbytes > 0){
        //compute LOGICAL BLOCK (lbk) and the startByte in that lbk:
        int lbk       = oftp->offset / BLKSIZE;
        int startByte = oftp->offset % BLKSIZE;

        // writing in direct blocks
        if (lbk < 12){
            if (ip->i_block[lbk] == 0){
                ip->i_block[lbk] = balloc(mip->dev);
            }
            blk = ip->i_block[lbk];
        }

        // writing to indirect block
        else if (lbk >= 12 && lbk < 256 + 12){
            //indirect blocks
            //Load indirect blocks into ibuf
            int ibuf[256];

            //Allocate a new block if necessary
            if (ip->i_block[12] == 0){
                ip->i_block[12] = balloc(mip->dev);
            }

            get_block(mip->dev, ip->i_block[12], ibuf);
```

```
    //Allocate a new block if necessary
    if (ibuf[lbk-12] == 0){
        ibuf[lbk-12] = balloc(mip->dev);
    }

    //Grab indirect based on lbk
    blk = ibuf[lbk-12];
}

// double indirect blocks
else{
  //double indirect blocks
  //Load double indirect blocks into buf
  int ibuf[256], dibuf[256];

  //Allocate a new block if necessary
  if (ip->i_block[13] == 0){
    ip->i_block[13] = balloc(mip->dev);
  }

  get_block(mip->dev, ip->i_block[13], dibuf);

  //Load corresponding indirect block, diblk (double indirect we are on) - iblk (indirect block we are on)
  int dilbk = (lbk-12-256) / 256;
  int ilbk = (lbk-12-256) % 256;

  //Allocate a new block if necessary
  if (dibuf[dilbk] == 0){
    dibuf[dilbk] = balloc(mip->dev);
  }

  //load in ibuf
  get_block(mip->dev, dibuf[dilbk], ibuf);

  //Allocate a new block if necessary
  if (ibuf[ilbk] == 0){
    ibuf[ilbk] = balloc(mip->dev);
  }

  //set blk value
  blk = ibuf[ilbk];
}

get_block(mip->dev, blk, wbuf); // read disk block into wbuf[ ]
char* cp = wbuf + startByte; // cp points at startByte in wbuf[]
remain = BLKSIZE - startByte; // number of BYTEs remain in this block

while (remain > 0){               // write as much as remain allows
    count++;
    *cp++ = *cq++;              // cq points at buf[ ]
    nbytes--; remain--;         // dec counts
    oftp->offset++;             // advance offset
    if (oftp->offset > mip->INODE.i_size)  // especially for RW|APPEND mode
        mip->INODE.i_size++;    // inc file size (if offset > fileSize)
    if (nbytes <= 0) break;     // if already nbytes, break
}
```

```
        put_block(mip->dev, blk, wbuf);   // write wbuf[ ] to disk
    }

    mip->dirty = 1;
    printf("wrote %d chars into fd = %d \n", count , fd);
    return count;
}
/************** type.h file ************************/
typedef unsigned char  u8;
typedef unsigned short u16;
typedef unsigned int   u32;

typedef struct ext2_super_block SUPER;
typedef struct ext2_group_desc  GD;
typedef struct ext2_inode       INODE;
typedef struct ext2_dir_entry_2 DIR;

SUPER *sp;
GD    *gp;
INODE *ip;
DIR   *dp;

#define FREE        0
#define READY       1

#define BLKSIZE  1024
#define NMINODE    64
#define NFD         8
#define NPROC       2

typedef struct minode{
  INODE INODE;
  int dev, ino;
  int refCount;
  int dirty;
  // for level-3
  int mounted;
  struct mntable *mptr;
}MINODE;


typedef struct oft{ // for level-2
  int  mode;
  int  refCount;
  MINODE *mptr;
  int  offset;
}OFT;

typedef struct proc{
  struct proc *next;
  int         pid;
  int         uid;
  int         status;
  MINODE      *cwd;
  OFT         *fd[NFD];
}PROC;
```