#include "type.h"

#include "source.h"

int main(int argc, char \*argv[])

{

int i, cmd;

char line[128], cname[64], parameter[64];

initialize();

mount\_root(argv[1]);

printf("Dev=%d, inodeBegin=%d, bmap=%d, imap=%d, ninodes=%d\n", dev, inodeBegin, bmap, imap, ninodes);

while(1)

{

strcpy(pathname, "");

strcpy(parameter, "");

strcpy(cname, "");

printf("P%d running: ", running->pid);

printf("input command : ");

fgets(line, 128, stdin);

//line[strlen(line)-1] = 0; // kill the \r char at end

if (line[0]==0) continue;

sscanf(line, "%s %s %s", cname, pathname, parameter);

printf("Pathname = %s, parameter = %s\n", pathname, parameter);

if(parameter[0] != 0)

{

strcat(pathname, " ");

strcat(pathname, parameter);

}

if(!strcmp(cname, "ls"))

{

ls(pathname);

}

else if(!strcmp(cname, "cd"))

{

cd(pathname);

}

else if(!strcmp(cname, "pwd"))

{

do\_pwd(pathname);

}

else if(!strcmp(cname, "quit"))

{

quit(pathname);

}

else if(!strcmp(cname, "mkdir"))

{

make\_dir(pathname);

}

else if(!strcmp(cname, "rmdir"))

{

my\_rmdir(pathname);

}

else if(!strcmp(cname, "creat"))

{

creat\_file(pathname);

}

else if(!strcmp(cname, "link"))

{

link(pathname);

}

else if(!strcmp(cname, "unlink"))

{

unlink(pathname);

}

else if(!strcmp(cname, "symlink"))

{

symlink(pathname);

}

}

}

#include "source.h"

// Initializes global variables

int initialize()

{

proc[0].uid = 0;

proc[0].cwd = 0;

proc[1].uid = 1;

proc[1].cwd = 0;

running = &(proc[0]);

readQueue = &(proc[1]);

int i = 0;

for(i = 0; i < 100; i++)

{

minode[i].refCount = 0;

minode[i].ino = 0;

}

for(i = 0; i < 10; i++) { MountTable[i].dev = 0;}

root = 0;

}

//Returns the requested block in buffer

int get\_block(int dev1, int blk, char \*buf)

{

lseek(dev1, (long)(blk\*BLKSIZE), 0);

read(dev1, buf, BLKSIZE);

}

int put\_block(int dev, int blk, char \*buf)

{

lseek(dev, (long)(blk\*BLKSIZE), 0);

write(dev, buf, BLKSIZE);

}

//Tokenizes given pathname into an array of strings terminated with a null string.

char\*\* tokenPath(char\* pathname)

{

int i = 0;

char\*\* name;

char\* tmp;

name = (char\*\*)malloc(sizeof(char\*)\*256);

name[0] = strtok(pathname, "/");

i = 1;

while ((name[i] = strtok(NULL, "/")) != NULL) { i++;}

name[i] = 0;

i = 0;

while(name[i])

{

tmp = (char\*)malloc(sizeof(char)\*strlen(name[i]));

strcpy(tmp, name[i]);

name[i] = tmp;

i++;

}

return name;

}

//searches through 12 blocks to look for str. If found returns ino number, else returns 0 (indicating filepath does not exist)

int search(int dev, char \*str, INODE \*ip)

{

int i;

char \*cp;

DIR \*dp;

char buf[BLKSIZE], temp[256];

for(i = 0; i < 12; i++)

{

if(ip->i\_block[i] == 0){break;}

get\_block(dev, ip->i\_block[i], buf);

dp = (DIR \*)buf;

cp = buf;

while(cp < buf+BLKSIZE)

{

memset(temp, 0, 256);

strncpy(temp, dp->name, dp->name\_len);

if(strcmp(str, temp) == 0){ return dp->inode;}

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

}

return 0;

}

int searchByIno(int dev, int ino, INODE \*ip, char\* temp)

{

int i;

char \*cp;

DIR \*dp;

char buf[BLKSIZE];

for(i = 0; i < 12; i++)

{

if(ip->i\_block[i] == 0){ break;}

get\_block(dev, ip->i\_block[i], buf);

dp = (DIR \*)buf;

cp = buf;

while(cp < buf+BLKSIZE)

{

if(ino == dp->inode) //Found the right inode

{

strncpy(temp, dp->name, dp->name\_len);

return 1;

}

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

}

return 0;

}

//Gets the ino number from a given pathname

unsigned int getino(int dev, char \*path)

{

int ino = 0, i = 0;

char \*\*tokens;

MINODE \*mip = NULL;

if(path && path[0])

{

tokens = tokenPath(path);

}

else //No pathname given so set ino to cwd

{

ino = running->cwd->ino;

return ino;

}

if(path[0]=='/') //start at root dir

{

ip = &(root->INODE);

ino = root->ino;

}

else //start at cwd

{

ip = &(running->cwd->INODE);

}

while(tokens[i])

{

ino = search(dev, tokens[i], ip);

if(0 >= ino)

{

if(mip){ iput(mip->dev, mip);}

return -1;

}

if(mip) { iput(mip->dev, mip);}

i++;

if(tokens[i])

{

mip = iget(dev, ino);

ip = &(mip->INODE);

}

}

i = 0;

while(tokens[i])

{

free(tokens[i]);

i++;

}

if(mip) { iput(mip->dev, mip);}

return ino;

}

MINODE \*iget(int dev1, unsigned int ino)

{

int i = 0, blk, offset;

char buf[BLKSIZE];

MINODE \*mip = NULL;

//search minode[100] to see if inode already exists in array

for(i = 0; i < 100; i++)

{

// If inode is already in array, set mip to point to MINODE in array, increment MINODE's refCount by 1.

if(minode[i].refCount > 0 && minode[i].ino == ino)

{

//printf("MINODE for inode %d already exists, just copying\n", minode[i].ino); //FOR TESTING

mip = &minode[i];

minode[i].refCount++;

return mip;

}

}

//If you have reached here then the inode does not currently exist in minode[100]. Put inode from disk into free MINODE in array's INODE field.

i = 0;

while(minode[i].refCount > 0 && i < 100) { i++;}

if(i == 100)

{

printf("Error: NO SPACE IN MINODE ARRAY\n");

return 0;

}

blk = (ino-1)/8 + inodeBegin;

offset = (ino-1)%8;

get\_block(dev1, blk, buf);

ip = (INODE \*)buf + offset;

memcpy(&(minode[i].INODE), ip, sizeof(INODE)); //Copy inode from disk into minode array

minode[i].dev = dev1;

minode[i].ino = ino;

minode[i].refCount = 1;

minode[i].dirty = 0;

minode[i].mounted = 0;

minode[i].mountptr = NULL;

return &minode[i];

}

//

int iput(int dev, MINODE \*mip)

{

char buf[BLKSIZE];

int blk, offset;

INODE \*tip;

mip->refCount--;

if(mip->refCount > 0) {return 1;}

if(mip->dirty == 0) {return 1;}

//Must write INODE back to disk

blk = (mip->ino-1)/8 + inodeBegin;

offset = (mip->ino-1)%8;

get\_block(dev, blk, buf);

tip = (INODE\*)buf + offset;

memcpy(tip, &(mip->INODE), sizeof(INODE));

put\_block(mip->dev, blk, buf);

return 1;

}

int mount\_root(char \*devName)

{

char buf[BLKSIZE];

dev = open(devName, O\_RDWR); //Open for read/write

if (dev < 0){ //Could not open device

printf("open %s failed\n", devName);

exit(1);

}

get\_super(dev, buf);

sp = (SUPER\*)buf;

if(is\_ext2(buf) <= 0) {exit(0);} //Check is ext2 filesystem, if not exit program

get\_inode\_table(dev); //Sets global variable inodeBegin to start of inode table

ninodes = sp->s\_inodes\_count;

root = iget(dev, ROOT\_INODE); //Set root inode

proc[0].cwd = iget(dev, ROOT\_INODE); // Set cwd for procedure 1 & 2 to root inode

proc[1].cwd = iget(dev, ROOT\_INODE);

MountTable[0].mounted\_inode = root;

MountTable[0].ninodes = ninodes;

MountTable[0].nblocks = sp->s\_blocks\_count;

MountTable[0].dev = dev;

strncpy(MountTable[0].name, devName, 256);

return dev;

}

// read SUPER block (block 1) and set contents to buf

int get\_super(int dev1, char \*buf)

{

get\_block(dev1, SUPERBLOCK, buf);

}

// Sets global variable inodeBegin to start of inode table for given device

void get\_inode\_table(int dev1)

{

char buf[BLKSIZE];

get\_gd(dev1, buf);

gp = (GD\*)buf;

inodeBegin = gp->bg\_inode\_table;

bmap = gp->bg\_block\_bitmap;

imap = gp->bg\_inode\_bitmap;

}

//Determines if device is an ext2 filesystem. If not exits program.

//Parameter is a buffer of BLKSIZE that contains the super block

int is\_ext2(char \*buf)

{

sp = (SUPER \*)buf;

if (SUPER\_MAGIC != sp->s\_magic)

{

printf("Error: Not an EXT2 file sytem\n");

return -1;

}

return 1;

}

// Reads group descriptor information in block 2

int get\_gd(int dev1, char \*buf)

{

get\_block(dev1, GDBLOCK, buf);

}

int cd(char\* pathname)

{

MINODE \*mip;

unsigned int ino;

if(!pathname || !pathname[0] || (pathname[0] == '/' && !pathname[1])){ ino = root->ino;}

else { ino = getino(dev, pathname);}

if(!ino)

{

printf("Error: Invalid Pathname\n");

return 0;

}

mip = iget(dev, ino);

//Verify inode is a dir

if(!S\_ISDIR(mip->INODE.i\_mode))

{

printf("Error: End of path is not a directory\n");

iput(dev, mip);

return 0;

}

iput(dev, running->cwd);

running->cwd = mip;

}

int do\_pwd(char \*pathname)

{

printf("cwd = ");

pwd(running->cwd);

printf("\n");

}

int pwd(MINODE \*wd)

{

int ino = 0;

MINODE \*next = NULL;

char temp[256];

if(wd == root)

{

printf("/");

return 1;

}

//Get parent's MINODE

ino = search(dev, "..", &(wd->INODE));

if(ino <= 0)

{

printf("ERROR: BAD INODE NUMBER\n");

return -1;

}

next = iget(dev, ino);

if(!next)

{

printf("ERROR: COULD NOT FIND INODE\n");

return -1;

}

pwd(next);

memset(temp, 0, 256);

searchByIno(next->dev, wd->ino, &(next->INODE), temp);

printf("%s/", temp);

iput(next->dev, next);

return;

}

int noPrintPwd(MINODE \*wd, char buf[256])

{

int ino = 0;

MINODE \*next = NULL;

char temp[256], temp2[256];

strncpy(temp2, buf, 256);

if(wd == root)

{

printf("/");

return 1;

}

//Get parent's MINODE

ino = search(dev, "..", &(wd->INODE));

if(ino <= 0)

{

printf("ERROR: BAD INODE NUMBER\n");

return -1;

}

next = iget(dev, ino);

if(!next)

{

printf("ERROR: COULD NOT FIND INODE\n");

return -1;

}

pwd(next);

memset(temp, 0, 256);

searchByIno(next->dev, wd->ino, &(next->INODE), temp);

sprintf(buf, "/%s%s", temp, temp2);

iput(next->dev, next);

return;

}

int ialloc(int dev1)

{

int i;

char buf[BLKSIZE]; // BLKSIZE=block size in bytes

// get inode Bitmap into buf[ ]

get\_block(dev1, imap, buf); // assume FD, bmap block# = 4

for (i=0; i < ninodes; i++){ // assume you know ninodes

if (tst\_bit(buf, i)==0){ // assume you have tst\_bit() function

set\_bit(buf, i); // assume you have set\_bit() function

put\_block(dev1, imap, buf); // write imap block back to disk

// update free inode count in SUPER and GD on dev

decFreeInodes(dev1); // assume you write this function

return (i+1);

}

}

return 0; // no more FREE inodes

}

int idalloc(int dev, int ino)

{

int i;

char buf[BLKSIZE];

get\_block(dev, IBITMAP, buf);

clr\_bit(buf, ino-1);

put\_block(dev, IBITMAP, buf);

incFreeInodes(dev);

}

int balloc(int dev1)

{

int i;

char buf[BLKSIZE]; // BLKSIZE=block size in bytes

get\_block(dev1, bmap, buf);

for (i=0; i < BLKSIZE; i++){ // assume you know ninodes

if (tst\_bit(buf, i)==0){ // assume you have tst\_bit() function

set\_bit(buf, i); // assume you have set\_bit() function

put\_block(dev1, bmap, buf); // write bmap block back to disk

// update free inode count in SUPER and GD on dev

decFreeBlocks(dev1); // assume you write this function

memset(buf, 0, BLKSIZE);

put\_block(dev1, i+1, buf);

return (i+1);

}

}

return 0; // no more FREE inodes

}

int bdalloc(int dev, int ino)

{

int i;

char buf[BLKSIZE];

get\_block(dev, BBITMAP, buf);

clr\_bit(buf, ino-1);

put\_block(dev, BBITMAP, buf);

incFreeBlocks(dev);

}

//Quit program iputs all dirty MINODEs

int quit(char\* pathname)

{

int i = 0;

char str[256];

for(i = 0; i < 10; i++)

{

if(running->fd[i] != NULL)

{

snprintf(str, 10, "%d", i);

close\_file(str);

}

}

for(i = 0; i < 100; i++)

{

if(minode[i].refCount > 0)

{

if(minode[i].dirty != 0)

{

minode[i].refCount = 1;

iput(dev, &minode[i]);

}

}

}

printf("Exiting Program\n");

exit(0);

}

int tst\_bit(char\* buf, int i)

{

int byt, offset;

byt = i/8;

offset = i%8;

return (((\*(buf+byt))>>offset)&1);

}

int set\_bit(char\* buf, int i)

{

int byt, offset;

char temp;

char \*tempBuf;

byt = i/8;

offset = i%8;

tempBuf = (buf+byt);

temp = \*tempBuf;

temp |= (1<<offset);

\*tempBuf = temp;

return 1;

}

int clr\_bit(char\* buf, int i)

{

int byt, offset;

char temp;

char \*tempBuf;

byt = i/8;

offset = i%8;

tempBuf = (buf+byt);

temp = \*tempBuf;

temp &= (~(1<<offset));

\*tempBuf = temp;

return 1;

}

int decFreeInodes(int dev)

{

char buf[BLKSIZE];

get\_super(dev, buf);

sp = (SUPER\*)buf;

sp->s\_free\_inodes\_count -= 1;

put\_block(dev, SUPERBLOCK, buf);

get\_gd(dev, buf);

gp = (GD\*)buf;

gp->bg\_free\_inodes\_count -=1;

put\_block(dev, GDBLOCK, buf);

return 1;

}

int incFreeInodes(int dev)

{

char buf[BLKSIZE];

get\_super(dev, buf);

sp = (SUPER\*)buf;

sp->s\_free\_inodes\_count += 1;

put\_block(dev, SUPERBLOCK, buf);

get\_gd(dev, buf);

gp = (GD\*)buf;

gp->bg\_free\_inodes\_count +=1;

put\_block(dev, GDBLOCK, buf);

return 1;

}

int decFreeBlocks(int dev1)

{

char buf[BLKSIZE];

get\_super(dev, buf);

sp = (SUPER\*)buf;

sp->s\_free\_blocks\_count -= 1;

put\_block(dev1, SUPERBLOCK, buf);

get\_gd(dev1, buf);

gp = (GD\*)buf;

gp->bg\_free\_blocks\_count -=1;

put\_block(dev1, GDBLOCK, buf);

return 1;

}

int incFreeBlocks(int dev1)

{

char buf[BLKSIZE];

get\_super(dev, buf);

sp = (SUPER\*)buf;

sp->s\_free\_blocks\_count += 1;

put\_block(dev1, SUPERBLOCK, buf);

get\_gd(dev1, buf);

gp = (GD\*)buf;

gp->bg\_free\_blocks\_count +=1;

put\_block(dev1, GDBLOCK, buf);

return 1;

}

int ls(char\* path)

{

int ino;

MINODE \*mip = NULL;

if(!path || !pathname[0]) { ino = running->cwd->ino;}

else if(pathname[0] == '/' && !pathname[1]) { ino = root->ino;}

else { ino = getino(dev, path);}

if(0 >= ino)

{

printf("Invalid pathname\n");

return -1;

}

mip = iget(dev, ino);

findBlocks(&(mip->INODE), 1);

iput(mip->dev, mip);

return 1;

}

//finds all the data blocks from a pointer to that inode and prints the dir names in those data blocks.

int findBlocks(INODE \*ip, int printStat)

{

int i, j , k;

unsigned int buf[256], buf2[256];

//Print dirs in direct blocks

for(i = 0; i < 12; i++)

{

if(ip->i\_block[i] != 0)

{

printDirs(ip->i\_block[i], printStat);

}

}

//Print dirs in indirect blocks

if(ip->i\_block[12]) //Indirect block exists

{

get\_block(dev, ip->i\_block[12], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i]) { printDirs(buf[i], printStat); }

}

}

//Print dirs in double indirect blocks

if(ip->i\_block[13]) //Double indirect block exists

{

get\_block(dev, ip->i\_block[13], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i])

{

get\_block(dev, buf[i], (char\*)buf2);

for(j = 0; j < 256; j++)

{

if(buf2[j]) { printDirs(buf2[j], printStat);}

}

}

}

}

}

int printDirs(int block, int printStat)

{

int i;

char \*cp;

DIR \*dp;

char buf[BLKSIZE], temp[256];

get\_block(dev, block, buf);

dp = (DIR \*)buf;

cp = buf;

while(cp < buf+BLKSIZE)

{

if(printStat) { printStat1(dp);}

else

{

memset(temp, 0, 256);

strncpy(temp, dp->name, dp->name\_len);

printf("%4d %4d %4d %s\n", dp->inode, dp->rec\_len, dp->name\_len, temp);

}

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

return 0;

}

int printStat1(DIR\* dp)

{

struct stat s;

char temp[256], lnk[256];

MINODE \*mip;

//Stat File

myStat(dp, &s);

//Print info

if (S\_ISLNK(s.st\_mode)) { printf("%s ", "LNK"); }

else if (S\_ISREG(s.st\_mode)) { printf("%s ", "REG"); }

else if(S\_ISDIR(s.st\_mode)) { printf("%s ", "DIR"); }

else { printf("%s ", "N/A"); }

printf("%hu ", s.st\_nlink);

if (test\_perm(&s, USER\_READ) != 0) {printf("r");}

else {printf("-");}

if (test\_perm(&s, USER\_WRITE) != 0) {printf("w");}

else {printf("-");}

if (test\_perm(&s, USER\_EXEC) != 0) {printf("x ");}

else {printf("- ");}

if (test\_perm(&s, GROUP\_READ) != 0) {printf("r");}

else {printf("-");}

if (test\_perm(&s, GROUP\_WRITE) != 0) {printf("w");}

else {printf("-");}

if (test\_perm(&s, GROUP\_EXEC) != 0) {printf("x ");}

else {printf("- ");}

if (test\_perm(&s, OTHER\_READ) != 0) {printf("r");}

else {printf("-");}

if (test\_perm(&s, OTHER\_WRITE) != 0) {printf("w");}

else {printf("-");}

if (test\_perm(&s, OTHER\_EXEC) != 0) {printf("x ");}

else {printf("- ");}

// Print UID

printf("%hu ", (unsigned short)s.st\_uid);

// Print size

printf("%lu ", (unsigned long)s.st\_size);

// Print ctime in calendar format

strncpy(temp,(char\*)ctime(&(s.st\_ctime)), 256);

temp[strlen(temp)-1] = 0;

printf("%s ", temp);

//Print filename

memset(temp, 0, 256);

strncpy(temp, dp->name, dp->name\_len);

printf("%s ", temp);

if(S\_ISLNK(s.st\_mode))

{

mip = iget(dev, dp->inode);

printf("-> %s", (char\*)mip->INODE.i\_block);

iput(mip->dev, mip);

}

printf("\n");

}

int myStat(DIR \*dp, struct stat \*s)

{

int ino;

MINODE \*mip;

if(!dp)

{

printf("Error: No directory entry\n");

return -1;

}

//Get inode

ino = dp->inode;

mip = iget(dev, ino);

ip = &(mip->INODE);

s->st\_dev = dev;

s->st\_ino = ino;

s->st\_mode = ip->i\_mode;

s->st\_nlink = ip->i\_links\_count;

s->st\_uid = ip->i\_uid;

s->st\_gid = ip->i\_gid;

s->st\_size = ip->i\_size;

s->st\_blksize = BLKSIZE;

s->st\_blocks = ip->i\_blocks;

s->st\_atime = ip->i\_atime;

s->st\_mtime = ip->i\_mtime;

s->st\_ctime = ip->i\_ctime;

iput(mip->dev, mip);

return 1;

}

int test\_mode(struct stat \*input, enum stat\_mode mode)

{

if (((input->st\_mode >> 12) & mode) == mode) { return 1; }

return 0;

}

int test\_perm(struct stat \*input, enum perm\_mode mode)

{

if (((input->st\_mode) & mode) != 0) { return 1; }

return 0;

}

int myDirname(char \*pathname, char buf[256])

{

int i = 0;

memset(buf, 0, 256);

strcpy(buf, pathname);

while(buf[i]) { i++; }

while(i >= 0)

{

if(buf[i] == '/')

{

buf[i+1] = 0;

//printf("parent = %s\n", buf); //FOR TESTING

return 1;

}

i--;

}

buf[0] = 0;

return 1;

}

int myBasename(char \*pathname, char \*buf)

{

int i = 0, j = 0;

if(!pathname[0]) {return -1;}

i = strlen(pathname);

while(i >= 0 && pathname[i] != '/') { i--; }

if(pathname[i] == '/')

{

i++;

while(pathname[i]) { buf[j++] = pathname[i++];}

buf[j] = 0;

return 1;

}

else { strncpy(buf, pathname, 256);}

return 1;

}

int make\_dir(char \*pathname)

{

int dev1, ino, r;

char parent[256], child[256], origPathname[512];

MINODE \*mip;

memset(parent, 0, 256);

memset(child, 0, 256);

memset(origPathname, 0, 512);

strcpy(origPathname, pathname);

if(pathname[0] == '/') { dev1 = root->dev; }

else { dev1 = running->cwd->dev; }

myDirname(pathname, parent);

myBasename(origPathname, child);

ino = getino(dev1, parent);

if(ino <= 0)

{

printf("ERROR: INVALID PATHNAME\n");

return -1;

}

mip = iget(dev1, ino);

if(!S\_ISDIR(mip->INODE.i\_mode))

{

printf("ERROR: NOT A DIRECTORY\n");

iput(dev1, mip);

return -1;

}

ino = search(dev1, child, &(mip->INODE));

if(ino > 0)

{

printf("ERROR: DIRECTORY ALREADY EXISTS");

iput(mip->dev, mip);

return -1;

}

//printf("Going into the mkdir function\n"); //FOR TESTING

r = my\_mkdir(mip, child);

iput(mip->dev, mip);

return r;

}

int my\_mkdir(MINODE \*pip, char child[256])

{

int inumber, bnumber, idealLen, needLen, newRec, i, j;

MINODE \*mip;

char \*cp;

DIR \*dpPrev;

char buf[BLKSIZE];

char buf2[BLKSIZE];

int blk[256];

inumber = ialloc(pip->dev);

bnumber = balloc(pip->dev);

mip = iget(pip->dev, inumber);

//Write contents into inode for Directory entry

mip->INODE.i\_mode = 0x41ED;

mip->INODE.i\_uid = running->uid;

mip->INODE.i\_gid = running->gid;

mip->INODE.i\_size = BLKSIZE;

mip->INODE.i\_links\_count = 2;

mip->INODE.i\_atime = mip->INODE.i\_ctime = mip->INODE.i\_mtime = time(0L);

mip->INODE.i\_blocks = 2;

mip->dirty = 1;

for(i = 0; i <15; i++) { mip->INODE.i\_block[i] = 0; }

mip->INODE.i\_block[0] = bnumber;

iput(mip->dev, mip);

//Write . and .. entries

dp = (DIR\*)buf;

dp->inode = inumber;

strncpy(dp->name, ".", 1);

dp->name\_len = 1;

dp->rec\_len = 12;

//printf("Wrote \".\" entry\n"); //FOR TESTING

cp = buf + 12;

dp = (DIR\*)cp;

dp->inode = pip->ino;

dp->name\_len = 2;

strncpy(dp->name, "..", 2);

dp->rec\_len = BLKSIZE - 12;

put\_block(pip->dev, bnumber, buf);

// Put name into parents directory

memset(buf, 0, BLKSIZE);

needLen = 4\*((8+strlen(child)+3)/4);

bnumber = findLastBlock(pip);

//Check if rooom in last block in parents directory

get\_block(pip->dev, bnumber, buf);

cp = buf;

dp = (DIR\*)cp;

while((dp->rec\_len + cp) < buf+BLKSIZE)

{

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

idealLen = 4\*((8+dp->name\_len+3)/4);

if(dp->rec\_len - idealLen >= needLen) //There is room in this block

{

//printf("There is room in this block\n"); //FOR TESTING

newRec = dp->rec\_len - idealLen;

dp->rec\_len = idealLen;

cp += dp->rec\_len;

dp = (DIR\*)cp;

dp->inode = inumber;

dp->name\_len = strlen(child);

strncpy(dp->name, child, dp->name\_len);

dp->rec\_len = newRec;

}

else // Allocate new data block

{

bnumber = balloc(pip->dev);

dp = (DIR\*)buf;

dp->inode = inumber;

dp->name\_len = strlen(child);

strncpy(dp->name, child, dp->name\_len);

dp->rec\_len = BLKSIZE;

addLastBlock(pip, bnumber);

}

put\_block(pip->dev, bnumber, buf);

pip->dirty = 1;

pip->INODE.i\_links\_count++;

memset(buf, 0, BLKSIZE);

//printf("Finding parent's ino\n"); //FOR TESTING

searchByIno(pip->dev, pip->ino, &running->cwd->INODE, buf);

touch(buf);

return 1;

}

int findLastBlock(MINODE \*pip)

{

int buf[256];

int buf2[256];

int bnumber, i, j;

//Find last used block in parents directory

if(pip->INODE.i\_block[0] == 0) {return 0;}

for(i = 0; i < 12; i++) //Check direct blocks

{

if(pip->INODE.i\_block[i] == 0)

{

return (pip->INODE.i\_block[i-1]);

}

}

if(pip->INODE.i\_block[12] == 0) {return pip->INODE.i\_block[i-1];}

get\_block(dev, pip->INODE.i\_block[12], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] == 0) {return buf[i-1];}

}

if(pip->INODE.i\_block[13] == 0) {return buf[i-1];}

//Print dirs in double indirect blocks

memset(buf, 0, 256);

get\_block(pip->dev, pip->INODE.i\_block[13], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] == 0) {return buf2[j-1];}

if(buf[i])

{

get\_block(pip->dev, buf[i], (char\*)buf2);

for(j = 0; j < 256; j++)

{

if(buf2[j] == 0) {return buf2[j-1];}

}

}

}

}

int addLastBlock(MINODE \*pip, int bnumber)

{

int buf[256];

int buf2[256];

int i, j, newBlk, newBlk2;

//Find last used block in parents directory

for(i = 0; i < 12; i++) //Check direct blocks

{

if(pip->INODE.i\_block[i] == 0) {pip->INODE.i\_block[i] = bnumber; return 1;}

}

if(pip->INODE.i\_block[12] == 0) //Have to make indirect block

{

newBlk = balloc(pip->dev);

pip->INODE.i\_block[12] = newBlk;

memset(buf, 0, 256);

get\_block(pip->dev, newBlk, (char\*)buf);

buf[0] = bnumber;

put\_block(pip->dev, newBlk, (char\*)buf);

return 1;

}

memset(buf, 0, 256);

get\_block(pip->dev, pip->INODE.i\_block[12], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] == 0) {buf[i] = bnumber; return 1;}

}

if(pip->INODE.i\_block[13] == 0) //Make double indirect block

{

newBlk = balloc(pip->dev);

pip->INODE.i\_block[13] = newBlk;

memset(buf, 0, 256);

get\_block(pip->dev, newBlk, (char\*)buf);

newBlk2 = balloc(pip->dev);

buf[0] = newBlk2;

put\_block(pip->dev, newBlk, (char\*)buf);

memset(buf2, 0, 256);

get\_block(pip->dev, newBlk2, (char\*)buf2);

buf2[0] = bnumber;

put\_block(pip->dev, newBlk2, (char\*)buf2);

return 1;

}

memset(buf, 0, 256);

get\_block(pip->dev, pip->INODE.i\_block[13], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] == 0)

{

newBlk2 = balloc(pip->dev);

buf[i] = newBlk2;

put\_block(pip->dev, pip->INODE.i\_block[13], (char\*)buf);

memset(buf2, 0, 256);

get\_block(pip->dev, newBlk2, (char\*)buf2);

buf2[0] = bnumber;

put\_block(pip->dev, newBlk2, (char\*)buf2);

return 1;

}

memset(buf2, 0, 256);

get\_block(pip->dev, buf[i], (char\*)buf2);

for(j = 0; j < 256; j++)

{

if(buf2[j] == 0) {buf2[j] = bnumber; return 1;}

}

}

printf("ERROR: COULD NOT ADD BLOCK TO INODE\n");

return -1;

}

int touch (char\* name)

{

char buf[1024];

int ino;

MINODE \*mip;

ino = getino(dev, name);

if(ino <= 0)

{

creat\_file(name);

return 1;

}

mip = iget(dev, ino);

mip->INODE.i\_atime = mip->INODE.i\_mtime = mip->INODE.i\_ctime = time(0L);

mip->dirty = 1;

iput(mip->dev, mip);

return 1;

}

int my\_chmod(char\* pathname)

{

char buf[1024];

char nMode[256];

char path[256];

int ino, newMode, i;

MINODE\* mip;

if(split\_paths(pathname, nMode, path) <= 0) { return -1; }

newMode = strtoul(nMode, NULL, 8);

ino = getino(dev, path);

if(ino <= 0)

{

printf("ERROR: INVALID PATHNAME\n");

return -1;

}

mip = iget(dev, ino);

i = ~0x1FF;

mip->INODE.i\_mode &= i;

mip->INODE.i\_mode |= newMode;

mip->dirty = 1;

iput(dev, mip);

return 1;

}

int creat\_file(char\* pathname)

{

int dev1, ino, r;

char parent[256];

char child[256];

MINODE \*mip;

memset(parent, 0, 256);

memset(child, 0, 256);

if(pathname[0] == '/') { dev1 = root->dev;}

else { dev1 = running->cwd->dev; }

myDirname(pathname, parent);

myBasename(pathname, child);

ino = getino(dev1, parent);

if(ino <= 0)

{

printf("ERROR: INVALID PATHNAME\n");

return -1;

}

mip = iget(dev1, ino);

if(!S\_ISDIR(mip->INODE.i\_mode))

{

printf("ERROR: NOT A DIRECTORY\n");

iput(dev1, mip);

return -1;

}

ino = search(dev1, child, &(mip->INODE));

if(ino > 0)

{

printf("ERROR: DIRECTORY ALREADY EXISTS");

iput(mip->dev, mip);

return -1;

}

r = my\_creat(mip, child);

iput(mip->dev, mip);

return r;

}

int my\_creat(MINODE \*pip, char child[256])

{

int inumber, bnumber, idealLen, needLen, newRec, i, j;

MINODE \*mip;

char \*cp;

DIR \*dpPrev;

char buf[BLKSIZE];

char buf2[BLKSIZE];

int blk[256];

inumber = ialloc(pip->dev);

mip = iget(pip->dev, inumber);

//Write contents into inode for Directory entry

mip->INODE.i\_mode = 0x81A4;

mip->INODE.i\_uid = running->uid;

mip->INODE.i\_gid = running->gid;

mip->INODE.i\_size = 0;

mip->INODE.i\_links\_count = 1;

mip->INODE.i\_atime = mip->INODE.i\_ctime = mip->INODE.i\_mtime = time(0L);

mip->INODE.i\_blocks = 0;

mip->dirty = 1;

for(i = 0; i <15; i++)

{

mip->INODE.i\_block[i] = 0;

}

iput(mip->dev, mip);

// Put name into parents directory

memset(buf, 0, BLKSIZE);

needLen = 4\*((8+strlen(child)+3)/4);

bnumber = findLastBlock(pip);

//Check if rooom in last block in parents directory

get\_block(pip->dev, bnumber, buf);

dp = (DIR\*)buf;

cp = buf;

while((dp->rec\_len + cp) < buf+BLKSIZE)

{

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

idealLen = 4\*((8+dp->name\_len+3)/4);

if(dp->rec\_len - idealLen >= needLen) //There is room in this block

{

newRec = dp->rec\_len - idealLen;

dp->rec\_len = idealLen;

cp += dp->rec\_len;

dp = (DIR\*)cp;

dp->inode = inumber;

dp->name\_len = strlen(child);

strncpy(dp->name, child, dp->name\_len);

dp->rec\_len = newRec;

}

else // Allocate new data block

{

bnumber = balloc(pip->dev);

dp = (DIR\*)buf;

dp->inode = inumber;

dp->name\_len = strlen(child);

strncpy(dp->name, child, dp->name\_len);

dp->rec\_len = BLKSIZE;

addLastBlock(pip, bnumber);

}

//printf("Putting parent block back\n"); //FOR TESTING

put\_block(pip->dev, bnumber, buf);

pip->dirty = 1;

memset(buf, 0, BLKSIZE);

searchByIno(pip->dev, pip->ino, &running->cwd->INODE, buf);

touch(buf);

return 1;

}

int my\_rmdir(char \*pathname)

{

int ino, i;

char parent[256], child[256], origPathname[512];

MINODE \*pip = NULL;

MINODE \*mip = NULL;

strcpy(origPathname, pathname);

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

if(!pathname || !pathname[0])

{

printf("ERROR: NO DIRECTORY GIVEN\n");

}

else

{

ino = getino(dev, pathname);

}

if(0 >= ino)

{

printf("Invalid pathname\n");

return -1;

}

mip = iget(dev, ino);

if(!S\_ISDIR(mip->INODE.i\_mode))

{

printf("ERROR: NOT A DIRECTORY\n");

iput(mip->dev, mip);

return -1;

}

if(mip->refCount > 1)

{

printf("ERROR: DIRECTORY IS IN USE\n");

return -1;

}

//Check if empty

if(mip->INODE.i\_links\_count > 2)

{

printf("ERROR: DIRECTORY NOT EMPTY\n");

iput(mip->dev, mip);

return -1;

}

//Check if files exist in directory by checking its data blocks

if(is\_empty(mip) != 0)

{

printf("ERROR: DIRECTORY NOT EMPTY\n");

iput(mip->dev, mip);

return -1;

}

for(i = 0; i < 12; i++)

{

if(mip->INODE.i\_block[i] != 0)

{

bdalloc(mip->dev, mip->INODE.i\_block[i]);

}

}

idalloc(mip->dev, mip->ino);

myDirname(origPathname, parent);

myBasename(origPathname, child);

//printf("dirname = %s, basename = %s\n", parent, child);

ino = getino(mip->dev, parent);

pip = iget(mip->dev, ino);

iput(mip->dev, mip);

//printf("Going to remove child now!\n"); //FOR TESTING

rm\_child(pip, child);

pip->INODE.i\_links\_count--;

//printf("pip->links = %d\n", pip->INODE.i\_links\_count);

touch(parent);

pip->dirty = 1;

iput(pip->dev, pip);

return 1;

}

int rm\_child(MINODE \*pip, char \*child)

{

int i, size, found = 0;

char \*cp, \*cp2;

DIR \*dp, \*dp2, \*dpPrev;

char buf[BLKSIZE], buf2[BLKSIZE], temp[256];

memset(buf2, 0, BLKSIZE);

for(i = 0; i < 12; i++)

{

if(pip->INODE.i\_block[i] == 0) { return 0; }

get\_block(pip->dev, pip->INODE.i\_block[i], buf);

dp = (DIR \*)buf;

dp2 = (DIR \*)buf;

dpPrev = (DIR \*)buf;

cp = buf;

cp2 = buf;

while(cp < buf+BLKSIZE && !found)

{

memset(temp, 0, 256);

strncpy(temp, dp->name, dp->name\_len);

if(strcmp(child, temp) == 0)

{

//if child is only entry in block

if(cp == buf && dp->rec\_len == BLKSIZE)

{

bdalloc(pip->dev, pip->INODE.i\_block[i]);

pip->INODE.i\_block[i] = 0;

pip->INODE.i\_blocks--;

found = 1;

}

//else delete child and move entries over left

else

{

while((dp2->rec\_len + cp2) < buf+BLKSIZE)

{

dpPrev = dp2;

cp2 += dp2->rec\_len;

dp2 = (DIR\*)cp2;

}

if(dp2 == dp) //Child is last entry

{

//printf("Child is last entry\n"); //FOR TESTING

dpPrev->rec\_len += dp->rec\_len;

found = 1;

}

else

{

//printf("Child is not the last entry\n"); //FOR TESTING

size = ((buf + BLKSIZE) - (cp + dp->rec\_len));

printf("Size to end = %d\n", size);

dp2->rec\_len += dp->rec\_len;

printf("dp2 len = %d\n", dp2->rec\_len);

memmove(cp, (cp + dp->rec\_len), size);

dpPrev = (DIR\*)cp;

memset(temp, 0, 256);

strncpy(temp, dpPrev->name, dpPrev->name\_len);

printf("new dp name = %s\n", temp);

found = 1;

}

}

}

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

if(found)

{

put\_block(pip->dev, pip->INODE.i\_block[i], buf);

return 1;

}

}

printf("ERROR: CHILD NOT FOUND\n");

return -1;

}

int is\_empty(MINODE \*mip)

{

int i;

char \*cp;

DIR \*dp;

char buf[BLKSIZE], temp[256];

for(i = 0; i < 12; i++)

{

if(ip->i\_block[i] == 0) { return 0; }

get\_block(dev, ip->i\_block[i], buf);

dp = (DIR \*)buf;

cp = buf;

while(cp < buf+BLKSIZE)

{

memset(temp, 0, 256);

strncpy(temp, dp->name, dp->name\_len);

if(strncmp(".", temp, 1) != 0 && strncmp("..", temp, 2) != 0) { return 1;}

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

}

}

int split\_paths(char \*original, char \*path1, char \*path2)

{

char \*temp;

//printf("Original = %s\n", original);

temp = strtok(original, " ");

strcpy(path1, temp);

//printf("Path1 = %s\n", path1);

temp = strtok(NULL, " ");

if(temp == NULL)

{

printf("ERROR: NO SECOND PATH GIVEN\n");

return -1;

}

strcpy(path2, temp);

return 1;

}

int link(char\* pathname)

{

char oldFile[256], newFile[256], parent[256], child[256], buf[BLKSIZE];

int ino, ino2, bnumber, needLen, idealLen, newRec;

MINODE \*mip, \*mip2;

char \*cp;

DIR \*dp;

if(0 >= split\_paths(pathname, oldFile, newFile)) { return -1; }

ino = getino(dev, oldFile);

if(ino <= 0) { return -1; }

mip = iget(dev, ino);

if(!S\_ISREG(mip->INODE.i\_mode))

{

printf("ERROR: PATH IS NOT A REGULAR FILE\n");

iput(mip->dev, mip);

return -1;

}

myDirname(newFile, parent);

myBasename(newFile, child);

if(0 >= (ino2 = getino(mip->dev, parent)))

{

iput(mip->dev, mip);

return -1;

}

mip2 = iget(mip->dev, ino2);

if(!S\_ISDIR(mip2->INODE.i\_mode))

{

printf("ERROR: NOT A DIRECTORY\n");

iput(mip->dev, mip);

iput(mip2->dev, mip2);

return -1;

}

//printf("Child = %s\n", child);

ino2 = search(mip2->dev, child, &(mip2->INODE));

if(ino2 > 0)

{

printf("ERROR: FILE ALREADY EXISTS\n");

iput(mip->dev, mip);

iput(mip2->dev, mip2);

return -1;

}

//Put name in parents block

memset(buf, 0, BLKSIZE);

needLen = 4\*((8+strlen(child)+3)/4);

bnumber = findLastBlock(mip2);

//Check if rooom in last block in parents directory

get\_block(mip2->dev, bnumber, buf);

dp = (DIR\*)buf;

cp = buf;

while((dp->rec\_len + cp) < buf+BLKSIZE)

{

cp += dp->rec\_len;

dp = (DIR\*)cp;

}

idealLen = 4\*((8+dp->name\_len+3)/4);

if(dp->rec\_len - idealLen >= needLen) //There is room in this block

{

newRec = dp->rec\_len - idealLen;

dp->rec\_len = idealLen;

cp += dp->rec\_len;

dp = (DIR\*)cp;

dp->inode = ino;

dp->name\_len = strlen(child);

strncpy(dp->name, child, dp->name\_len);

dp->rec\_len = newRec;

}

else // Allocate new data block

{

bnumber = balloc(mip2->dev);

dp = (DIR\*)buf;

dp->inode = ino;

dp->name\_len = strlen(child);

strncpy(dp->name, child, dp->name\_len);

dp->rec\_len = BLKSIZE;

addLastBlock(mip2, bnumber);

}

//printf("Putting parent block back\n"); //FOR TESTING

put\_block(mip2->dev, bnumber, buf);

mip->dirty = 1;

mip->INODE.i\_links\_count++;

memset(buf, 0, BLKSIZE);

searchByIno(mip2->dev, mip2->ino, &running->cwd->INODE, buf);

iput(mip->dev, mip);

iput(mip2->dev, mip2);

return 1;

}

int unlink(char \*pathname)

{

char oldFile[256], parent[256], child[256], buf[BLKSIZE];

int ino, ino2, bnumber, needLen, idealLen, newRec;

MINODE \*mip, \*mip2;

char \*cp;

DIR \*dp;

ino = getino(dev, pathname);

if(ino <= 0)

{

printf("ERROR: FILE DOES NOT EXIST\n");

return -1;

}

mip = iget(dev, ino);

if(!S\_ISREG(mip->INODE.i\_mode) && !S\_ISLNK(mip->INODE.i\_mode))

{

printf("ERROR: NOT A FILE\n");

iput(mip->dev, mip);

return -1;

}

mip->INODE.i\_links\_count--;

if(mip->INODE.i\_links\_count <= 0)

{

rm(mip);

}

myDirname(pathname, parent);

myBasename(pathname, child);

ino2 = getino(mip->dev, parent);

if(ino2 <= 0) {return -1;}

mip2 = iget(mip->dev, ino2);

rm\_child(mip2, child);

iput(mip->dev, mip);

iput(mip->dev, mip2);

}

int rm(MINODE \*mip)

{

int i, j;

int buf[256], buf2[256];

if(!S\_ISLNK(mip->INODE.i\_mode))

{

for(i = 0; i < 12; i++)

{

if(mip->INODE.i\_block[i] != 0)

{

bdalloc(mip->dev, mip->INODE.i\_block[i]);

}

}

if(mip->INODE.i\_block[12] != 0)

{

memset(buf, 0, 256);

get\_block(mip->dev, mip->INODE.i\_block[12], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] != 0) {bdalloc(mip->dev, buf[i]);}

}

bdalloc(mip->dev, mip->INODE.i\_block[12]);

}

if(mip->INODE.i\_block[13] != 0)

{

memset(buf, 0, 256);

get\_block(mip->dev, mip->INODE.i\_block[13], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] != 0)

{

memset(buf2, 0, 256);

get\_block(mip->dev, buf[i], (char\*)buf2);

for(j = 0; j < 256; j++)

{

if(buf2[j] != 0) {bdalloc(mip->dev, buf2[j]);}

}

bdalloc(mip->dev, buf[i]);

}

}

bdalloc(mip->dev, mip->INODE.i\_block[13]);

}

}

idalloc(mip->dev, mip->ino);

return 1;

}

int symlink(char \*pathname)

{

char oldname[256], newname[256];

int ino;

MINODE \*mip;

if(split\_paths(pathname, oldname, newname) <= 0) {return -1;}

if(0 >= (ino = getino(dev, oldname)))

{

printf("ERROR: FILE DOES NOT EXIST\n");

return -1;

}

creat\_file(newname);

if(0 >= (ino = getino(dev, newname)))

{

printf("ERROR: COULD NOT CREATE FILE\n");

return -1;

}

mip = iget(dev, ino);

mip->INODE.i\_mode &= ~0770000;

mip->INODE.i\_mode |= 0120000;

mip->dirty = 1;

strcpy((char\*)mip->INODE.i\_block, oldname);

iput(mip->dev, mip);

}

int readlink(char \*pathname, char \*linkStr)

{

int ino;

MINODE \*mip;

if(0 >= (ino = getino(dev, pathname)))

{

printf("ERROR: INVALID PATH\n");

return -1;

}

mip = iget(dev, ino);

if(!S\_ISLNK(mip->INODE.i\_mode))

{

printf("ERROR: NOT A LINK\n");

iput(mip->dev, mip);

return -1;

}

strcpy(linkStr, (char\*)mip->INODE.i\_block);

iput(mip->dev, mip);

return 1;

}

int open\_file(char \*pathname)

{

char filePath[256], cMode[256];

int mode, ino, i;

MINODE \*mip;

OFT \*oftp;

if(split\_paths(pathname, filePath, cMode) <= 0) {return -1;}

mode = atoi(cMode);

if(0 >= (ino = getino(dev, pathname)))

{

printf("ERROR: INVALID PATH\n");

return -2;

}

mip = iget(dev, ino);

if(!S\_ISREG(mip->INODE.i\_mode))

{

printf("ERROR: NOT A REGULAR FILE\n");

iput(mip->dev, mip);

return -1;

}

//TODO: Check permissions

for(i = 0; i < 10; i++)

{

if (running->fd[i] != NULL)

{

if(running->fd[i]->inodeptr == mip)

{

if(running->fd[i]->mode != 0 || mode != 0)

{

printf("ERROR: FILE IS IN USE\n");

iput(mip->dev, mip);

return -1;

}

}

}

}

oftp = (OFT\*)malloc(sizeof(OFT));

oftp->mode = mode;

oftp->refCount = 1;

oftp->inodeptr = mip;

switch(mode)

{

case 0: oftp->offset = 0;

break;

case 1: truncate(mip);

oftp->offset = 0;

break;

case 2: oftp->offset = 0;

break;

case 3: oftp->offset = mip->INODE.i\_size;

break;

default: printf("ERROR: INVALID MODE\n");

iput(mip->dev, mip);

free(oftp);

return -1;

break;

}

i = 0;

while(running->fd[i] != NULL && i < 10) { i++; }

if(i == 10)

{

printf("ERROR: NO ROOM TO OPEN FILE\n");

iput(mip->dev, mip);

free(oftp);

return -1;

}

running->fd[i] = oftp;

touch(pathname);

if(mode != 0) { mip->dirty = 1; }

return i;

}

int truncate(MINODE \*mip)

{

int buf[256];

int buf2[256];

int bnumber, i, j;

if(mip == NULL)

{

printf("ERROR: NO FILE\n");

return -1;

}

// Deallocate all used blocks

for(i = 0; i < 12; i++) //Check direct blocks

{

if(mip->INODE.i\_block[i] != 0)

{

bdalloc(mip->dev, mip->INODE.i\_block[i]);

}

}

//Indirect blocks

if(mip->INODE.i\_block[12] == 0) {return 1;}

get\_block(dev, mip->INODE.i\_block[12], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i] != 0) {bdalloc(mip->dev, buf[i]);}

}

bdalloc(mip->dev, mip->INODE.i\_block[12]);

if(mip->INODE.i\_block[13] == 0) {return 1;}

//deallocate all double indirect blocks

memset(buf, 0, 256);

get\_block(mip->dev, mip->INODE.i\_block[13], (char\*)buf);

for(i = 0; i < 256; i++)

{

if(buf[i])

{

get\_block(mip->dev, buf[i], (char\*)buf2);

for(j = 0; j < 256; j++)

{

if(buf2[j] != 0) {bdalloc(mip->dev, buf2[j]);}

}

bdalloc(mip->dev, buf[i]);

}

}

bdalloc(mip->dev, mip->INODE.i\_block[13]);

mip->INODE.i\_atime = mip->INODE.i\_mtime = time(0L);

mip->INODE.i\_size = 0;

mip->dirty = 1;

return 1;

}

int close\_file(char \*pathname)

{

OFT \*oftp;

MINODE \*mip;

int fd;

if(!pathname[0])

{

printf("ERROR: NO FILE DESCRIPTOR GIVEN\n");

return -1;

}

fd = atoi(pathname);

if (fd > 9 || fd < 0)

{

printf("ERROR: FILE DESCRIPTOR OUT OF RANGE\n");

return -1;

}

if(running->fd[fd] == NULL)

{

printf("ERROR: FILE DESCRIPTOR NOT FOUND\n");

return -1;

}

oftp = running->fd[fd];

running->fd[fd] = 0;

oftp->refCount--;

if(oftp->refCount > 0) {return 1;}

mip = oftp->inodeptr;

iput(mip->dev, mip);

free(oftp);

return 1;

}