```c
#include "type.h"

void get_block(int fd, int blk, char buf[BLKSIZE])
{
    lseek(fd, (long)(blk*BLKSIZE), 0);
    read(fd, buf, BLKSIZE);
}

void put_block(int fd, int blk, char buf[BLKSIZE])
{
    lseek(fd, (long)(blk*BLKSIZE), 0);
    write(fd, buf, BLKSIZE);
}

/*
Mailman's algorithm
(ino - 1) / 8 = n
(ino - 1) % 8 = m
n blocks, m inodes over for needed ino
*/

MINODE * iget(int dev, int ino)
{
    int ipos = 0;
    int i = 0;
    int offset = 0;
    char buf[1024];
    INODE *ip = NULL;
    MINODE *mip = malloc(sizeof(MINODE));//memory inode pointer


    for (i = 0; i < NMINODE; i++)//looks for the needed memory inode
    {
        mip = &minode[i];

        if(mip->dev == dev && mip->ino == ino)//checks to see if we are at the
correct inode in memory
        {
            mip->refCount++;//increment ref count of memory inode
            return mip;//returns pointer to it
        }
    }

    //mailman's
    ipos = (ino - 1)/8 + INODE_START_POS;
    offset = (ino - 1) % 8;

    //get block
    get_block(dev, ipos, buf);
    ip = (INODE *)buf + offset;

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

        if (mip->refCount == 0)//refCount is zero this it is a free memory inode
        {
            mip->INODE = *ip;//copys inode into memory inode
            mip->dev = dev;
            mip->ino = ino;
            mip->refCount++;
            return mip;//return memory inode pointer
        }
    }
}

void tokenize(char pathname[256])//tokens pathname into individual strings
{
    char path_copy[256] = "";
    char *temp;
    int i = 0;

    for(i = 0; i < 64; i++)
        strcpy(names[i], "");

    i = 0;


    strcpy(path_copy, pathname);
    strcat(path_copy, "/");
    temp = strtok(path_copy, "/");

    while(temp != NULL)
    {
        strcpy(names[i], temp);
        temp = strtok(NULL, "/");//breaks down pathname and stores it into name array
        i++;
    }
}

int get_Inode(MINODE *mip, char pathname[64])
{
    int ino = 0, i = 0, n = 0;
    int inumber, offset;
    char path[64];
    char name[64][64];
    char *temp;
    char buf[1024];

    //root
    if(!strcmp(pathname, "/"))
        return 2;

    //starts at root
    if(pathname[0] == '/')
        mip = root;

    //parse pathname
    if(pathname)
    {
        strcat(pathname, "/");
        temp = strtok(pathname, "/");

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

    for(i = 0; i < n; i++)
```

```c
    {
        ino = search(mip, name[i]);

        if(ino == 0) //no node exists
            return 0;

        mip = iget(dev, ino);//gets the memory inode
    }

    return ino;
}

int search(MINODE *mip, char *name)
{
    int i;
    char buf[BLKSIZE], *cp;
    char dir_name[64];
    DIR *dir_p;

    if(!S_ISDIR(mip->INODE.i_mode))//checks to see if we are actually in a dir
    {
        printf("ERROR: Not a directory!\n");
        return 0;
    }
    for(i = 0; i < 12; i++)
    {
        if(mip->INODE.i_block[i])//if the pointer points at an inode block with is
not empty continue
        {
            get_block(dev, mip->INODE.i_block[i], buf);

            dir_p = (DIR *)buf;
            cp = buf;

            while(cp < buf + BLKSIZE)
            {

                if(dir_p->name_len < 64)//name less than 64
                {
                    strncpy(dir_name, dir_p->name, dir_p->name_len);//copy name into
dir_name for length name_len
                    dir_name[dir_p->name_len] = 0;
                }
                else
                {
                    strncpy(dir_name, dir_p->name, 64);//large dir name, copy first
64 characters into dirname
                    dir_name[63] = 0;
                }
                if(strcmp(name, dir_name) == 0)//if the dir name is the same as what
we are looking for then return the dir inode
                    return dir_p->inode;

                cp += dir_p->rec_len;//increment and keep checking
                dir_p = (DIR *)cp;
            }
        }
    }
    return 0;//return zero when the search did not find what it wanted to
}
```

```
void iput(MINODE *mip)//disposes of a used memory inode by using it's pointer
{
    int ino = 0;
    int offset, ipos;
    char buf[1024];

    ino = mip->ino;//set inode number to memory inode pointer number

    mip->refCount--;//decrement refcount of memory inode


    if (mip->refCount == 0 && mip->dirty == 0)//ref count is zero thus it is empty
return, or if it is dirty return
        return;

    //mailman's
    ipos = (ino - 1) / 8 + INODE_START_POS;
    offset = (ino -1) % 8;

    //read
    get_block(mip->dev, ipos, buf);//gets the block that correctly corresuper_ponds
to the inode and stores it in buf

    ip = (INODE*)buf + offset;//updates global inode pointer with current buf+offset
for its memory address
    *ip = mip->INODE;//sets that memory address to the inode that is currently in our
memory pointer

    //write
    put_block(mip->dev, ipos, buf);
    mip->dirty = 0;
}

int findmyname(MINODE *parent, int myino, char *myname)
{

    int i;
    INODE *ip;
    char buf[BLKSIZE];
    char *cp;
    DIR *dir_p;

    if(myino == root->ino)
    {
        strcpy(myname, "/");
        return 0;
    }

    if(!parent)
    {
        printf("No parent.\n");
        return 1;
    }

    ip = &parent->INODE;

    if(!S_ISDIR(ip->i_mode))
    {
        printf("Not a directory.\n");
        return 1;
    }
    for(i = 0; i < 12; i++)//walk through direct blocks that the parent inode contains
    {
        if(ip->i_block[i])//if there is an inode then check it,
        {
            get_block(dev, ip->i_block[i], buf);
            dir_p = (DIR*)buf;
            cp = buf;

            while(cp < buf + BLKSIZE)
            {
                if(dir_p->inode == myino)//checks whether or not the inode we are
looking for is here
                {
                    strncpy(myname, dir_p->name, dir_p->name_len);//copies the name
into the array
                    myname[dir_p->name_len] = 0;//sets the last index to null term
                    return 0;
                }
                else
                {
                    cp += dir_p->rec_len;//increments the current pointer to next one
                    dir_p = (DIR*)cp;
                }
            }
        }
    }
    return 1;
}

int findino(MINODE *mip, int *myino, int *parentino)
{
    INODE *ip;
    char buf[1024];
    char *cp;
    DIR *dir_p;

    if(!mip)//check if memeory pointer exists

    {
        printf("Error, the Inode doesn't exist.\n");
        return 1;
    }

    ip = &mip->INODE;//set inode pointer to our memory inode pointer

    if(!S_ISDIR(ip->i_mode))//checks to see if the inode is a directory
    {
        printf("Error, the Inode is not a directory.\n");
        return 1;
    }

    get_block(dev, ip->i_block[0], buf);
    dir_p = (DIR*)buf;
    cp = buf;

    //.
    *myino = dir_p->inode;

    cp += dir_p->rec_len;//increments the current pointer to get to the parent
    dir_p = (DIR*)cp;//casts it to a directory pointer

    //..
    *parentino = dir_p->inode;//sets the parent inode to the previously changed
directory pointer

    return 0;
}

int clearBlocks(MINODE *mip)
{
    int Dev = mip->dev;
    int *in_block, *double_block;
    char indirect_buf[1024], double_buf[1024];

    for(int i =0; i < 12; i++)
    {
        if(mip->INODE.i_block[i] != 0)
            break;
        bdealloc(Dev, mip->INODE.i_block[i]);
        mip->INODE.i_block[i] = 0;
    }

    //indirect blocks
        if(mip->INODE.i_block[12] != 0)
        {
                get_block(Dev, mip->INODE.i_block[12], indirect_buf);
                in_block = (int *)indirect_buf;

                for(int i = 0; i < 256; i++)
                {
                        if(in_block[i] == 0)
                        {
                                continue;
                        }
                        bdealloc(Dev, in_block[i]);
                }
                bdealloc(Dev, mip->INODE.i_block[12]);
                mip->INODE.i_block[12] = 0;
        }

        memset(indirect_buf, 0, BLOCK_SIZE);
        memset(double_buf, 0, BLOCK_SIZE);

        //doubly indirect blocks
        if(mip->INODE.i_block[13] != 0)
        {
                get_block(Dev, mip->INODE.i_block[13], double_buf);
                double_block = (int *)double_buf;

                for(int i = 0; i < 256; i++)
                {
                        if(double_block[i] == 0)
                        {
                                continue;
                        }

                        get_block(Dev, double_block[i], indirect_buf);
                        in_block = (int *)indirect_buf;

                        for(int j = 0; j < 256; j++)
                        {
                                if(in_block[j] == 0)
                                {
                                        continue;
                                }
                                bdealloc(Dev, in_block[j]);
                        }

                        memset(indirect_buf, 0, BLOCK_SIZE);

                }
                bdealloc(Dev, mip->INODE.i_block[13]);
                mip->INODE.i_block[13] = 0;
        }

    mip->dirty = 1;
    mip->INODE.i_blocks = 0;
    mip->INODE.i_size = 0;
    mip->INODE.i_ctime = time(0L);
    mip->INODE.i_mtime = time(0L);
    return 0;
}

int checkDuplicates(MINODE * mip)
{
    int i;
        OFT *oftp;

        for(int i = 0; i < NOFT; i++)
        {
                oftp = &OpenFileTable[i];


                if(oftp->refCount > 0)
                {
                        if(oftp->inodeptr->dev == mip->dev && oftp->inodeptr->ino ==
mip->ino)
                        {
                                if(oftp->mode != 0)
                                {
                                        return 1;
                                }
                        }
                }
        }
        return 0;


}
```