Main.c

```
#include <ext2fs/ext2_fs.h>
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include <fcntl.h>
#include <libgen.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>

#include "type.h"

//global
int fd, dev;
int ninodes, nblocks;
int bmap, imap, inode_start, iblock;

char line[256], cmd[32], pathname[256];

char gpath[256];
char *name;
int n;

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

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

char *t1 = "xwrxwrxwr-------";
char *t2 = "----------------";

//typedefs
typedef void(*cmd_ptr)();

//function prototypes
int init();
int mount_root();
void ls(char *pathname);
void chdir(char *pathname);
void pwd(MINODE *wd);
int make_dir(char *pathname);
int creat_file(char *pathname);

void quit();

//int find_cmd(char *command);
void printCommands();

char *commands[32] = { "ls", "cd", "pwd", "quit", 0 };
cmd_ptr cmd_ptrs[32] = { &ls, &chdir, &pwd, &quit };
```

```
int main(int argc, char *argv[]) {
	init();
	mount_root();
	int i;
	while (1) {
		//print available commands
		printCommands();
		fgets(line, 256, stdin);
		i = sscanf(line, "%s %s", cmd, pathname);
		printf("cmd = %s, pathname = %s\n", cmd, pathname);
		//cmd_ptrs[find_cmd(cmd)](pathname);
		if (strcmp(cmd, "ls") == 0) {
			ls(pathname);
		}
		else if (strcmp(cmd, "cd") == 0) {
			chdir(pathname);
		}
		else if (strcmp(cmd, "pwd") == 0) {
			pwd(running->cwd);
		}
		else if (strcmp(cmd, "quit") == 0) {
			quit();
		}
		else if (strcmp(cmd, "mkdir") == 0) {
			make_dir(pathname);
		}
		else if (strcmp(cmd, "creat") == 0) {
			creat_file(pathname);
		}
		else {
			printf("%s is not an available command\n");
		}
	}
	return 0;
}

void printCommands() {
	printf("=============== commands ==============\n");
	printf("| ls | cd | pwd | mkdir | creat | quit |\n");
	printf("=======================================\n\n");
	printf("Enter a command: ");
}

int init() {
	int i, j;
	for (i = 0; i < NMINODE; ++i) {
		minode[i].refCount = 0;
	}
	for (i = 0; i < NPROC; ++i) {
		proc[i].status = READY;
		proc[i].pid = i;
		proc[i].uid = i;
		for (j = 0; j < NFD; ++j) {
			proc[i].fd[j] = 0;
		}
		proc[i].next = &proc[i + 1];
	}
	proc[NPROC - 1].next = &proc[0];
```

```
	running = &proc[0];
	root = 0;
}

int mount_root() {
	char buf[BLKSIZE];
	//SUPER *sp;
	//GD * gp;
	fd = open("disk", O_RDWR);
	if (fd < 0) {
		printf("Failed to open disk\n");
		exit(1);
	}
	get_block(fd, SUPERBLOCK, buf);
	sp = (SUPER *)buf;
	if (sp->s_magic != SUPER_MGAIC) {
		printf("diskImage is not ext2\n");
		exit(1);
	}
	ninodes = sp->s_inodes_count;
	nblocks = sp->s_blocks_count;
	printf("ninodes = %d nblocks = %d\n", ninodes, nblocks);
	get_block(fd, GDBLOCK, buf);
	gp = (GD *)buf;
	bmap = gp->bg_block_bitmap;
	imap = gp->bg_inode_bitmap;
	inode_start = iblock = gp->bg_inode_table;
	printf("bmp=%d imap=%d iblock = %d\n", bmap, imap, iblock);
	root = iget(fd, inode_start);
	proc[0].cwd = iget(fd, 2);
	proc[1].cwd = iget(fd, 2);
	running = &proc[0];
}

//helper functions

int ls_file(int ino, char *fname) {
	MINODE *mip = iget(fd, ino);
	ip = &mip->inode;
	char ftime[64], linkname[128];
	int i;
	//print file mode
	if ((ip->i_mode & 0xF000) == 0x8000) {
		printf("%c", '-');
	}
	if ((ip->i_mode & 0xF000) == 0x4000) {
		printf("%c", 'd');
	}
	if ((ip->i_mode & 0xF000) == 0xA000) {
		printf("%c", 'l');
	}
	for (i = 8; i >= 0; --i) {  //print file permissions
		if (ip->i_mode & (1 << i)) {
			printf("%c", t1[i]);
		}
		else {
			printf("%c", t2[i]);
		}
```

```
        }       //print file info
        printf("%4d", ip->i_links_count);
        printf("%4d", ip->i_gid);
        printf("%4d", ip->i_uid);
        printf("%8d", ip->i_size);
        strcpy(ftime, ctime(&ip->i_ctime));
        ftime[strlen(ftime) - 1] = 0;
        printf("%s", ftime);
        printf("%s", basename(fname));
        if ((ip->i_mode & 0xF000) == 0xA000) { //print symbolic link
                readlink(fname, linkname, 128);
                printf(" - > %s", linkname);
        }
        printf("\n");
}

int ls_dir(char *dname) {
        int ino = getino(pathname);
        MINODE *mip = iget(fd, ino);
        char buf[BLKSIZE], *cp;
        //DIR *dp;
        int i = 0;
        get_block(fd, mip->inode.i_block[0], buf);
        dp = (DIR *)buf;
        while (cp < buf + BLKSIZE) {
                mip = iget(fd, dp->inode);
                ls_file(mip->ino, dp->name);
                dp = (DIR *)buf;
                cp = buf;
        }
}

//command functions
void ls(char *pathname) {
        char *name, buf;
        //DIR *dp;
        if (strcmp(pathname, "") == 0) {   //ls for cwd
                get_block(fd, running->cwd->inode.i_block[0], buf);
                dp = (DIR *)buf;
                name = dp->name;
                ls_dir(name);
        }
}

void chdir(char *pathname) {
        MINODE *mip;
        if (strcmp(pathname, "") == 0) {
                running->cwd = root;
        }
        else {
                int ino = getino(pathname);
                if (ino == 0) {
                        printf("Error: failed to get ino\n");
                        return;
                }
                mip = iget(fd, ino);
                if (mip->inode.i_mode != FILE_MODE) {
                        printf("pathname is not a DIR\n");
```

```
                        return;
                }
                iput(running->cwd);
                running->cwd = mip;
        }
}

void rpwd(MINODE *wd) {
        int ino, pino;
        INODE *pip;
        char buf[BLKSIZE], myname[128];

        if (wd == root) {
                return;
        }
        pino = findino(wd, &ino);
        pip = iget(fd, pino);
        findmyname(pip, ino, myname);
        rpwd(pip);
        printf("/%s", myname);
}

void pwd(MINODE *wd) {
        if (wd == root) {
                printf("/\n");
        }
        else {
                rpwd(wd);
        }
}

void quit() {
        //write minodes to disk
        int i;
        MINODE *mip;
        for (i - 0; i < NMINODE; ++i) {
                mip = &minode[i];
                if (mip->refCount && mip->dirty) {
                        mip->refCount = 1;
                        iput(mip);
                }
        }
        //exit
        exit(0);
}

int enter_name(MINODE *pip, int myino, char *myname) {
        int i, remain, ideal_len, bno;
        char buf[BLKSIZE], *cp;

        for (i = 0; i < 12; ++i) {
                if (pip->inode.i_block[i] == 0) {
                        break;
                }
        }
        get_block(fd, pip->inode.i_block[i], buf);

        dp = (DIR *)buf;
```

```
	cp = buf;
	printf("step to LAST entry in data block %d\n", pip->inode.i_block[i]);
	while (cp + dp->rec_len < buf + BLKSIZE) {
		printf("ino = %d, name = %s \n", dp->inode, dp->name);
		cp += dp->rec_len;
		dp = (DIR *)cp;
	}
	ideal_len = 4 * ((8 + dp->name_len + 3) / 4);
	remain = dp->rec_len - ideal_len;
	if (remain < dp->rec_len) {
		dp->rec_len = ideal_len;
		cp += dp->rec_len;
		dp = (DIR *)cp;
		dp->inode = myino;
		dp->rec_len = remain;
		strcpy(dp->name, myname);
		dp->name_len = strlen(myname);
		put_block(fd, pip->inode.i_block[i], buf);
	}
	else {
		bno = balloc(fd);
		ip = &pip->inode;
		ip->i_size += BLKSIZE;
		ip->i_block[i + 1] = bno;
		enter_name(pip, myino, myname);
	}
}

int mymkdir(MINODE *pip, char *name) {
	MINODE *mip;
	int ino, bno, i;
	ino = ialloc(fd);
	bno = balloc(fd);
	printf("ino = %d, bno = %d\n", ino, bno);

	mip = iget(fd, ino);
	ip = &mip->inode;
	ip->i_mode = DIR_MODE;
	ip->i_uid = running->uid;
	ip->i_gid = running->gid;
	ip->i_size = BLKSIZE;
	ip->i_links_count = 2;
	ip->i_atime = ip->i_ctime = ip->i_mtime = time(0L);
	ip->i_blocks = 2;
	ip->i_block[0] = bno;
	for (i = 1; i < 14; ++i) {
		ip->i_block[i] = 0;
	}
	mip->dirty = 1;
	iput(mip);
	enter_name(mip, ino, ".");
	enter_name(mip, pip->ino, "..");
	enter_name(pip, ino, name);
	return 0;
}

int make_dir(char *pathname) {
	MINODE *start;
```

```
	int dev, pino;
	char *parent, *child, temp[256];
	MINODE *pmip;

	printf("%s\n", pathname);
	strcpy(temp, pathname);
	printf("pathname copied\n");
	if (temp[0] == '/') {
		//printf("path is relative to root\n");
		start = root;
		dev = root->dev;
		//printf("start = root, dev = %d\n", dev);
	}
	else {
		//printf("path is relative to cwd\n");
		start = running->cwd;
		//printf("Debug: assign start\n");
		dev = running->cwd->dev;
		//printf("Debug: assign dev\n");
		//printf("start = %d, dev = %d\n", start->ino, dev);
	}
	parent = dirname(temp);
	child = basename(temp);
	printf("parent = %s, child = %s\n", parent, child);
	pino = getino(parent);
	pmip = iget(dev, pino);
	printf("pino = %d, pmip = %d\n", pino, pmip->ino);
	if (pmip->inode.i_mode != DIR_MODE) {
		printf("Error: %s is not a valid pathname\n", pathname);
		return -1;
	}
	if (search(pmip, child) == 0) {
		printf("Error: %s already exists\n", pathname);
	}
	mymkdir(pmip, child);
	//update i_links_count
	pmip->inode.i_links_count += 1;
	//touch atime
	pmip->inode.i_atime = time(0L);
	pmip->dirty = 1;
	iput(pmip);
	return 0;
}

int my_creat(MINODE *pip, char *name) {
	MINODE *mip;
	int ino, bno, i;
	ino = ialloc(fd);
	//bno = balloc(fd);
	printf("ino = %d, bno = %d\n", ino, bno);
	mip = iget(fd, ino);
	ip = &mip->inode;
	ip->i_mode = FILE_MODE;
	ip->i_uid = running->uid;
	ip->i_gid = running->gid;
	ip->i_size = 0;
	ip->i_links_count = 1;
	ip->i_atime = ip->i_ctime = ip->i_mtime = time(0L);
```

```
	ip->i_blocks = 2;
	//ip->i_block[0] = bno;
	for (i = 0; i < 14; ++i) {
		ip->i_block[i] = 0;
	}
	mip->dirty = 1;
	iput(mip);
	enter_name(pip, ino, name);
	return 0;
}

int creat_file(char *pathname) {
	MINODE *start;
	int dev, pino;
	char *parent, *child, temp[256];
	MINODE *pmip;
	strcpy(temp, pathname);

	if (pathname[0] == '/') {
		start = root;
		dev = root->dev;
	}
	else {
		start = running->cwd;
		dev = running->cwd->dev;
	}
	parent = dirname(temp);
	child = basename(temp);
	pino = getino(parent);
	pmip = iget(dev, pino);
	if (pmip->inode.i_mode != DIR_MODE) {
		printf("Error: %s is not a valid pathname\n", pathname);
		return -1;
	}
	if (search(pmip, child) == 0) {
		printf("Error: %s already exists\n", pathname);
	}
	my_creat(pmip, child);
	pmip->inode.i_atime = time(0L);
	pmip->dirty = 1;
	iput(pmip);
	return 0;
}
```

Type.h

```
#include <ext2fs/ext2_fs.h>

//constants
#define SUPERBLOCK 1
#define GDBLOCK 2
#define ROOT_INODE 2

#define DIR_MODE 0x41ED
#define FILE_MODE 0x81AE
#define SUPER_MGAIC 0xEF53

#define SUPER_USER 0
#define FREE 0
#define BUSY 1
#define READY 2

#define NMINODE 64
#define NMTABLE 10
#define NPROC 2
#define NFD 10
#define NOFT 40

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

#define BLKSIZE 1024

typedef struct oft {
	int mode;
	int refCount;
	struct minode *minodePtr;
	int offset;
}OFT;

typedef struct proc {
	struct proc *next;
	int pid;
	int uid;
	int gid;
	int ppid;
	int status;
	struct minode *cwd;
	OFT *fd[NFD];
}PROC;

typedef struct minode {
	INODE inode;
	int dev, ino;
	int refCount;
	int dirty;
	int mounted;
	struct mount *mntPtr;

}MINODE;
```

```
typedef struct mtable {
	int dev;
	int ninodes;
	int nblocks;
	int free_blocks;
	int free_inodes;
	int bmap;
	int imap;
	int iblock;
	MINODE *mntDirPtr;
	char devName[64];
	char mntName[64];
};
```

Util.c

```
/*********** util.c file ****************/
#include "type.h"
#include <string.h>
#include <stdint.h>

/**** 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];

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

int get_block(int dev, int blk, char *buf)
{
	lseek(dev, (long)blk*BLKSIZE, 0);
	read(dev, buf, BLKSIZE);
}

int put_block(int dev, int blk, char *buf)
{
	lseek(dev, (long)blk*BLKSIZE, 0);
	write(dev, buf, BLKSIZE);
}

int tokenize(char *pathname)
{
	char *s;
	strcpy(gpath, pathname);
	// YOUR tokenize() code: strtok(gpath)
```

```
	n = 0;
	s = strtok(gpath, "/");
	while (s) {
		name[n++] = s;
		s = strtok(NULL, "/");
	}
}

// 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];
		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;
		}
	}
	for (i = 0; i < NMINODE; i++) {
		mip = &minode[i];
		if (mip->refCount == 0) {
			//printf("allocating NEW minode[%d] for [%d %d]\n", i, dev, ino);
			mip->refCount = 1;
			mip->dev = dev;
			mip->ino = ino;
			// 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);
			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;
}

int iput(MINODE *mip)
{
	int i, block, offset;
	char buf[BLKSIZE];
	INODE *ip;
	if (mip == 0)
		return;
	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);
}

int search(MINODE *mip, char *name)
{
	// YOUR search function: return ino if found name; else return 0;
	int i;
	char *cp, temp[256], sbuf[BLKSIZE];
	DIR *dp;
	for (i = 0; i < 12; ++i) {
		if (mip->inode.i_block[i] == 0) {
			return 0;
		}
		get_block(mip->dev, mip->inode.i_block[i], sbuf);
		dp = (DIR *)sbuf;
		cp = sbuf;
		while (cp < sbuf + BLKSIZE) {
			strncpy(temp, dp->name, dp->name_len);
			temp[dp->name_len] = 0;
			printf("%8d%8d%8u %s", dp->inode, dp->rec_len, dp->name_len, temp);
			if (strcmp(name, temp) == 0) {
				printf("found %s : inum = %d\n", name, dp->inode);
				cp += dp->rec_len;
				dp = (DIR *)cp;
			}
		}
	}
}

int getino(char *pathname)
{
	int i, ino, blk, disp;
	INODE *ip;
	MINODE *mip;
	printf("getino: pathname=%s\n", pathname);
	if (strcmp(pathname, "/") == 0)
		return 2;
	if (pathname[0] == '/')
		mip = iget(dev, 2);
	else
		mip = iget(running->cwd->dev, running->cwd->ino);
	tokenize(pathname);
	for (i = 0; i < n; i++) {
		printf("==========================================\n");
		ino = search(mip, name[i]);
		if (ino == 0) {
			iput(mip);
			printf("name %s does not exist\n", name[i]);
			return 0;
		}
		iput(mip);
		mip = iget(dev, ino);
```

```
        }
        iput(mip);
        return ino;
}

int findmyname(MINODE *parent, uint32_t myino, char myname[])
{
        char buf[BLKSIZE];
        // find mynio in parent data block; copy name string to myname[ ];
        //get_block(fd, parent->inode.i_block[0], buf);
        INODE *ip = &parent->inode;
        int i;
        DIR *dp;
        char *cp;
        for (i = 0; i < 12; ++i) {
                if (ip->i_block[i] == 0) {
                        break;
                }
        }
        get_block(fd, ip->i_block[i], buf);
        dp = (DIR *)buf;
        cp = buf;
        while (cp < buf + BLKSIZE) {
                if (dp->inode == myino) {
                        strncpy(myname, dp->name, dp->name_len);
                        return 0;
                }
                cp += dp->rec_len;
                dp = (DIR *)cp;
        }
}

int findino(MINODE *mip, uint32_t *myino) // return ino of parent and myino of .
{
        char buf[BLKSIZE], *cp;
        DIR *dp;
        get_block(mip->dev, mip->inode.i_block[0], buf);
        cp = buf;
        dp = (DIR *)buf;
        *myino = dp->inode;
        cp += dp->rec_len;
        dp = (DIR *)cp;
        return dp->inode;
}

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 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 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(fd);
			return i + 1;
		}
	}
	return 0;
}

int balloc(int dev)  // allocate an inode number
{
	int  i;
	char buf[BLKSIZE];
	// read inode_bitmap block
	get_block(dev, bmap, buf);
	for (i = 0; i < nblocks; i++) {
		if (tst_bit(buf, i) == 0) {
			set_bit(buf, i);
			put_block(dev, bmap, buf);
			decFreeInodes(fd);
			return i + 1;
		}
	}
	return 0;
}
```