实验三: 文件系统的实现

基本信息

- 姓名: 尤永庭
- 学号: 09016314
- 日期: 2019年9月

实验内容

实现一个简单的文件系统naive。Naïve文件系统的基本规格如下:

- 块大小512B
- 文件系统的大小不超过2MB
- 文件/目录的最大长度不超过4KB，文件/目录名不超过30个字符
- 超级块中必须有一成员项，用于放置学号
- 只需要支持普通文件和目录的基本操作，特殊类型的文件不需考虑。

实验目的

- 熟悉OS中文件系统的结构

设计思路和流程图

1. **底层接口实现:**
   - 通常操作系统底层会有关于存储设备读写的API, 会首先配置硬盘, 然后提供以Block为单位读写的接口.
   - 本次实验实现了在linux用户态下简单的文件系统, 即建立文件作为虚拟磁盘设备, 这个设备提供以块为读写的基本功能, 然后在这个虚拟磁盘设备上实现文件系统的功能.
   - 按照要求, 块大小为512B, 文件系统2MB, 即需要4096个块, 使用dd命令建立虚拟磁盘:

     ```
     dd if=/dev/zero of=simulated_device bs=512 count=4096
     ```

2. **基本数据结构设计:**
   - SuperBlock: 存放有关文件系统的管理信息
   - BitMap: 使用位来管理某一个块是否被使用, 分为INode Bitmap, Data Bitmap, 分别代表某一个INode或者Data的block是被使用或者空闲.
   - INode: file或者directory的索引节点, 存放关于文件或者directory的metadata, 可以通过块号找到文件或directory的存放地址
   - DIR: directory struct, 包含一个DirEntry list
   - DirEntry: 目录下的每一项, 包含一个name以及inode号, 通过这个inode编号可以找到具体的文件或者目录.

- 由于只需要支持普通文件, 所以如果是file类型的话, inode直接指向一块数据区, 数据区里的内容不需要以struct形式来存放.

3. **文件系统布局**
   - BLOCK为0~4095号
   - #0: SuperBlock
   - #1: INode Bitmap
   - #2: Data Bitmap
   - #3~#100: INode Block
   - #101~#4095: Data Block
4. **文件系统初始化**
   - 创建设备
   - 构造一个superblock, 写入block 0#
   - 初始化bitmap, 全设置为0表示空闲, 然后按照superblock内规定写入相应block
   - 主要操作为新建一个root文件夹, 并且把这个root文件夹的INode挂到第一个INode Block
5. **文件表设计:**
   - 在打开操作之后, 系统会返回一个file discriptor(fd), 然后就可以根据这个fd对文件进行读写操作. 具体实现为系统保存一个int list, index为fd, list中内容为文件所在inode的编号
   - 按照一般操作系统的设计, 一般会有一个全局文件表, 然后每一个进程的文件表会按照实际情况指向全局文件表的一些项, 这个设计中只做了一个全局文件表
6. **基础操作设计**
   - 文件系统需要支持关于文件, 目录的基础操作
   - 头文件中关于操作的函数声明:

```
/* file API */
extern int my_open (const char * path);
extern int my_creat (const char * path);
extern int my_read (int fd, void * buf, int count);
extern int my_write (int fd, const void * buf, int count);
extern int my_close (int fd);

extern int my_remove (const char * path);
extern int my_rename (const char * old, const char * new);
extern int my_mkdir (const char * path);
extern int my_rmdir (const char * path);
```

主要数据结构及其说明

1. 底层磁盘操作接口: 包括磁盘初始化, 读出磁盘的一个BLOCK, 写入磁盘的一个BLOCK

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include "fs.h"

/* only open the file once */
static int fd = -1;
static int devsize = 0;
```

```
/* returns the device size (in blocks) if the operation is successful,
 * and -1 otherwise */
//dd if=/dev/zero of=simulated_device bs=512 count=4096
int dev_open ()
{
  struct stat st;

  if (fd < 0) {
    fd = open ("simulated_device", O_RDWR);
    if (fd < 0) {
      perror ("open");
      return -1;
    }
    if (fstat (fd, &st) < 0) {
      perror ("fstat");
      return -1;
    }
    devsize = st.st_size / BLOCKSIZE;
  }
  return devsize;
}

/* returns 0 if the operation is successful, and -1 otherwise */
int read_block (int block_num, char * block)
{
  if (block_num >= devsize) {
    printf ("block number requested %d, maximum %d", block_num, devsize - 1);
    return -1;
  }
  if (lseek (fd, block_num * BLOCKSIZE, SEEK_SET) < 0) {
    perror ("lseek");
    return -1;
  }
  if (read (fd, block, BLOCKSIZE) != BLOCKSIZE) {
    perror ("read");
    return -1;
  }
  return 0;
}

/* returns 0 if the operation is successful, and -1 otherwise */
int write_block (int block_num, char * block)
{
  if (block_num >= devsize) {
    printf ("block number requested %d, maximum %d", block_num, devsize - 1);
    return -1;
  }
  if (lseek (fd, block_num * BLOCKSIZE, SEEK_SET) < 0) {
    perror ("lseek");
    return -1;
  }
  if (write (fd, block, BLOCKSIZE) != BLOCKSIZE) {
    perror ("write");
    return -1;
  }
  if (fsync (fd) < 0)
    perror ("fsync");   /* but return success anyway */
  return 0;
}
```

2. **文件系统数据结构**

- SuperBlock:

  主要存放这个文件系统的具体布局, 包括inode从哪一个磁盘块开始, data从哪一个磁盘块开始, 剩余的磁盘空间, 方便新放入文件时做出判断.

  按照试验要求, 放入一个学号的成员项.

  ```
  typedef struct SuperBlock{
      int inode_loc; //starting block of I-nodes, 1st inode -> root dir
      int data_loc; //starting block of data region
      int free_space; //how many remaining blocks are empty
  ```

```
    char student_num[9];
} SUPERBLOCK;
```

- INode:

  希望每一个INode都放进一个block内, 所以file ptr的数量需要有限制, 使得这个struct不要大过512B, 否则需要其他设计进行扩展

  同时, 实验要求说明文件/目录的最大长度不超过4KB, 那么只需要8个直接指针就足够了; 如果希望支持更大的文件, 那么可以使用二级, 三级指针

```
typedef struct INode{
    int file_type; //0: file, 1: Dir, other: unvalid
    int file_length; //how many bytes
    int file_ptr[INODE_ENTRIES]; //block number of the file(at most 8)
} INODE;
```

- 目录与目录文件项目:

  同样, 设计Dir时为了考虑让Dir能放入一个block, 对entry数目也有限制

  按照试验要求, 规定了文件/目录名不超过30个字符

```
//size: 36B
typedef struct DirEntry{
    char name[MAX_NAME_LENGTH];
    int inode;
} DIRENTRY;

//size: 364B
typedef struct Dir{
    int count; //entry count in this directory
    DIRENTRY list[MAX_DIR_ENTRY];
} DIR;
```

源程序与注释(部分)

实现逻辑:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <math.h>
#include "fs.h"
#include "bitmap.c"
#include "dir.c"

FILETABLE file_table;


/* input: inode number; return: fd */
int register_filetable(int inode){
  for(int i=0; i<FILETABLE_SIZE; i++){
    //ft entry is not in use or used by same file
    if(file_table.inode_ptr[i] == -1 || (inode == file_table.inode_ptr[i])){
      (file_table.inode_ptr)[i] = inode;
      return i;
    }
  }
  return -1;
}

void display_filetable(){
  for(int i=0; i<FILETABLE_SIZE; i++){
    printf("%d, %d\n", i, file_table.inode_ptr[i]);
```

```
  }
}

int parse_path(char *file_path){
  char tmp[MAX_FILE_NAME_LENGTH];
  memcpy(tmp, file_path, MAX_FILE_NAME_LENGTH);
  int count = 0;
  char *delim = "/";
    char *p;
    strcpy(PATH[count++], strtok(tmp, delim));
    while((p = strtok(NULL, delim)))
      strcpy(PATH[count++], p);
    return count;
}

//return block number of directory
int navigate_path(int level){
  //首先得到根目录的 DIR
  int dir_block = -1;
  char *result_buf = malloc(BLOCKSIZE);
  int result  = read_block(101, result_buf);
  DIR *dir = (DIR*)(result_buf);
  //根据path层数前进
  for(int i=0; i<level-1; i++){
    char *current_dir_name = PATH[i];
    for(int j = 0; j<MAX_DIR_ENTRY; j++){
      //遍历每一个entry
      DIRENTRY entry = (dir->list)[i];
      //如果entry name和下一步路径名相同, 则进行跳转操作
      if(entry.inode != -1 && entry.name == current_dir_name){
        int new_inode = entry.inode;
        char *buf = malloc(BLOCKSIZE);
        read_block(new_inode, buf);
        INODE *current_node = (INODE*)buf;
        if(current_node->file_type == 1){
          //hack: 暂时不考虑占有多个block的dir
          dir_block = (current_node->file_ptr)[0];
          read_block(dir_block, buf);
          dir = (DIR*)buf;
        }
      }
    }
  }
  return dir_block;
}
/* open an exisiting file for reading or writing */
//如果成功返回fd, 否则-1
int my_open (const char * path)
{
  int level = parse_path(path);
  int current_inode_num = 3;
  char *buf = malloc(BLOCKSIZE);
  for(int i=0; i<level - 1; i++){
    read_block(current_inode_num, buf);//buf中读入一个Inode
    INODE *current_inode = (INODE*)buf;
    int dir_block = current_inode->file_ptr[0];
    read_block(dir_block, buf);
    DIR *current_dir = (DIR*)buf;
    DIRENTRY *entry_lst = current_dir->list;
    for(int j=0; j<MAX_DIR_ENTRY; j++){
      if(strcmp(entry_lst[i].name, PATH[i]) == 0){
        current_inode_num = entry_lst[i].inode;
        break;
      }
    }
  }
  read_block(current_inode_num, buf);
  INODE *current_inode = (INODE*)buf;
  int current_dir_block =   current_inode->file_ptr[0];
  char *filename = PATH[level-1];
  read_block(current_dir_block, buf);
  DIR *current_dir = (DIR*)buf; //current_dir is root
  int entry_count = current_dir->count;
  DIRENTRY *entry_lst = current_dir->list;
  //遍历目录下所有entry, 找到目标文件然后打开
  for(int i=0; i<entry_count; i++){
    char *n = entry_lst[i].name;
```

```
    if(strcmp(n, filename) == 0){
      int file_inode = entry_lst[i].inode;
      return register_filetable(file_inode);
    }
  }
  return -1;
}

/* open a new file for writing only */
int my_creat (const char * path)
{
  int level = parse_path(path);
  int current_inode_num = 3;
  for(int i=0; i<level - 1; i++){
    char *buf = malloc(BLOCKSIZE);
    read_block(current_inode_num, buf);//buf中读入一个Inode
    INODE *current_inode = (INODE*)buf;
    int dir_block = current_inode->file_ptr[0];
    read_block(dir_block, buf);
    DIR *current_dir = (DIR*)buf;
    DIRENTRY *entry_lst = current_dir->list;
    for(int j=0; j<MAX_DIR_ENTRY; j++){
      if(strcmp(entry_lst[i].name, PATH[i]) == 0){
        current_inode_num = entry_lst[i].inode;
        break;
      }
    }
  }
  //只支持root
  //create inode
  char *name = PATH[level -1 ];

  DIRENTRY newfile;
  strcpy(newfile.name, name);
  newfile.inode = make_file();
  //connect to root
  add_entry_inode(current_inode_num, newfile);
  return 1;

}

/* sequentially read from a file */
int my_read (int fd, void * buf, int count)
{
  int block_num = ceil((float)count/BLOCKSIZE);//文件一共占多少个block
  int inode_num = file_table.inode_ptr[fd];//通过fd找到inode号
  char *tmpbuf = malloc(BLOCKSIZE);
  read_block(inode_num, tmpbuf);
  INODE *file_inode = (INODE*)tmpbuf;
  //每一个block分别读出
  for(int i=0; i<block_num; i++){
    int current_data_block = file_inode->file_ptr[i];
    char *tmpblock = malloc(BLOCKSIZE);
    read_block(current_data_block, tmpblock);
    memcpy(buf + i*BLOCKSIZE, tmpblock, BLOCKSIZE);
  }
  return -1;
}

/* sequentially write to a file */
int my_write (int fd, const void * buf, int count)
{
  int block_num = ceil((float)count/BLOCKSIZE);//文件一共占多少个block
  int inode_num = file_table.inode_ptr[fd];//通过fd找到inode号
  char *tmpbuf = malloc(BLOCKSIZE);
  read_block(inode_num, tmpbuf);
  INODE *file_inode = (INODE*)tmpbuf;
  //每一个block分别写入
  for(int i=0; i<block_num; i++){
    int new_data_block = take_data();
    file_inode->file_ptr[i] = new_data_block;
    char *tmpblock = malloc(BLOCKSIZE);
    memcpy(tmpblock, buf + i*BLOCKSIZE, BLOCKSIZE);
    write_block(new_data_block, tmpblock);
  }
  write_block(inode_num, file_inode);
}
```

```
/* only works if all but the last component of the path already exists */
int my_mkdir (const char * path)
{
  //hack: 先只支持一级
  char *dirname = path;
  int father_inode = 3;
  int dir_inode = make_dir(father_inode);
  DIRENTRY entry;
  entry.inode = dir_inode;
  strcpy(entry.name, dirname);
  add_entry_inode(father_inode, entry);
}



/* check to see if the device already has a file system on it,
 * and if not, create one. */
void init_device(){
  /* initialize device */
  //dd if=/dev/zero of=simulated_device bs=512 count=4096
  int devsize = dev_open();
  printf("Device initialized, size = %d\n", devsize);
}

void init_superblock(){
  //init superblock
  SUPERBLOCK init_super_block = {.inode_loc = 3, .data_loc = 101, .free_space = 3000, .student_num = "09016314"};
  write_block(0, (char*)(&init_super_block)); //write into block 0#
  /*
  //some test
  char* result = malloc(BLOCKSIZE);
  read_block(0, result);
  SUPERBLOCK *sb = (SUPERBLOCK*) result;
  printf("student num233: %s\n", sb->student_num);
  */
}

void init_root_dir(){
  int root_inode = 3;
  make_dir(3);
  int file_inode = make_file();

  char *buf = malloc(BLOCKSIZE);
  read_block(root_inode, buf);
  INODE *root = (INODE*)buf;
  int root_block = root->file_ptr[0];
  read_block(root_block, buf);
  DIR *rootdir = (DIR*)buf;


  //make testfile
  DIRENTRY testfile;
  strcpy(testfile.name,  "testfile");
  testfile.inode = make_file();
  add_entry(rootdir, testfile);
  write_block(root_block, rootdir);


  //make testfolder
  DIRENTRY testfolder;
  strcpy(testfolder.name, "testfolder");
  testfolder.inode = make_dir(root_inode);
  add_entry(rootdir, testfolder);
  write_block(root_block, rootdir);


  //make myfile
  DIRENTRY myfile;
  strcpy(myfile.name, "myfile");
  myfile.inode = make_file();
  //已知testfolder的inode，获取testfolder的DIR
  read_block(testfolder.inode, buf);
  INODE *testfolder_inode = (INODE*)buf;
  int testfolder_data = testfolder_inode->file_ptr[0];
  read_block(testfolder_data, buf);
```

```
  DIR *testfolder_dir = (DIR*)buf;
  add_entry(testfolder_dir, myfile);
  write_block(testfolder_data, testfolder_dir );


}

void init_filetable(){
  for(int i=0; i<FILETABLE_SIZE; i++){
    file_table.inode_ptr[i] = -1;
  }
}
void my_mkfs ()
{
  init_device();
  init_superblock();
  init_bitmap();
  init_root_dir();
  init_filetable();

  printf ("my_mkfs finished\n");
}
```

程序运行时初值和运行结果

- 使用一个简单的testcase来验证创建目录, 打开文件, 并且写入读出文件, 具体为:
  - 初始化文件系统
  - 新建了1024B的buffer, 跨度为2个block, 然后第一个block的最后两个字节和第二个block前两个字节设置为char '6'.
  - 然后在root下新建/testfolder/t文件, 然后打开t文件返回一个fd
  - 使用fd打开文件, 尝试写入buffer
  - 从同样的目录读出这个buffer, 打印出第一个block最后两个字节, 第二个block前两个字节, 验证是否都为'6'
  - 测试文件:

```
int main(int argc, char ** argv)
{
  //文件系统初始化
  my_mkfs();
  //根目录下新建文件夹testfolder
  my_mkdir("testfolder");
  //testfolder路径下新建文件t
  my_creat("/testfolder/t");
  //打开文件/testfolder/t
  int fd = my_open("/testfolder/t");
  printf("fd of file t = %d\n", fd);

  //新建长度为2个block的buffer
  //0-1023B, 内容随机 BLOCK1: 0-511, BLOCK2: 512-1023
  //两个block交界处写入4个6， buffer 510B-513B == '6'
  int length = BLOCKSIZE * 2;
  char *testinput = malloc(length);
  for(int i=0; i<4; i++){
    testinput[BLOCKSIZE-2+i] = '6';
  }
  my_write(fd, testinput, length);
  char *testoutput = malloc(length);
  my_read(fd, testoutput, length);
    for(int i=0; i<4; i++){
    printf("%d#: %c\n",BLOCKSIZE-2+i, testoutput[BLOCKSIZE-2+i]);
  }


}
```

- 运行

```
parallels@parallels-Parallels-Virtual-Platform:/media/psf/Home/Desktop/files/csapp/Labs/myfilesystem$ ./output
Device initialized, size = 4096
my_mkfs finished
fd of file t = 0
510#: 6
511#: 6
512#: 6
513#: 6
```

实验体会

1. 上课听file system implementation一章其实有很多不理解, 通过这次课程设计对原本一些superblock, inode等抽象概念有了更加具体的理解, 也大致了解了OS文件系统的基本构成
2. 系统级编程使用的C语言非常接近底层, 给了程序员极大的自由去通过指针访问内存空间, 因此非常容易出错; 而现代编程语言经过了很多层封装, 在语言设计上就避免了内存相关错误的可能性. 因此使用C写大规模的项目是非常有难度的, 需要严格遵守编码规范来避免错误. 比如: 调用任何通过返回值显示执行状态错误与否的函数, 一定要检查返回值, 为了代码的简洁性可以自己封装一层wrapper来做错误检查, 提升简洁性.