

内存文件系统

定义

mem 定义

```
#define blocknr 16384
static void *mem[ blocknr ];
```

这是一个指针数组，用的时候用如下代码映射到内存：


```
mem[blocknum] = mmap(NULL, blocksize, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

blocksize 定义

```
int blocksize = 32768; //一页32kb
```

定义为一个变量，是因为宏定义它的话，会无法通过gcc编译（原因找不到呀QAQ）。

(like this)



```
myfs.c:4:19: error: expected ';', ',', or ')' before numeric constant
#define blocksize 32768
                  ^
```

filename 的定义

使用链表储存，链表加长采用头插法。

```
#define maxblock 16000
struct filename //文件节点存储为链表
{
    char filename[64]; //文件名
    int file_block; //文件节点使用的页号
    int block_num; //文件总共拥有页数
    unsigned short content[maxblock]; //content记录页号
    struct stat st; //st是文件信息
    struct filename *next; //采用链表储存，头插法
};
```

一个 `filename` 差不多刚好用完一页，`content` 是一个数组，记录的是对应文件的用到的页在 `mem` 数组中的下标。支持随机访问，所以读写速度稳定。

之前我尝试用

$$maxblock = 32768 - sizeof(char * 64) - \dots$$

$$maxblock = blocksize - sizeof(char * 64) - \dots$$

来定义maxblock，但是结果也是无法通过gcc编译（error说是定义数组时不能使用变量定义），所以无奈之下只能用16000这个整数来定义。

因为页大小的限制，一个文件最大只有16000页，也就是

$$16000 * 32KB = 500MB$$

整个文件系统大小为512MB（0.5GB）。

root指针的映射

mem[0] 映射了一个内存空间放置指向root指针的指针 root_for_begin：

```
static struct filenode **root_for_begin;
```

init 函数中将 mem[0] 分配给 root_for_begin。

```
static void *oshfs_init(struct fuse_conn_info *conn)
//初始化mem栈
{
    int i = 0;
    for (i = 0; i < blocknr; i++) {
        mem[i] = NULL;
        mstack[i] = blocknr - i - 1;
    }
    mstack_pointer = blocknr - 1;
    int z;
    z = get_mem();
    root_for_begin = (struct filenode**)mem[z];//将root_for_begin映射到mem[0]中去;
    *root_for_begin = NULL;//初始设定文件系统为空
    return NULL;
}
```

root 指针定义如下：

```
static struct filenode *root = NULL;
```

它用于程序中进行链表操作，并且每次更新它都会相应地更新 root_for_begin 指向的空间中的值：

```
*root_for_begin = root;
```

保证可以从 mem[0] 中读出 root 的值并恢复整个文件系统。

函数实现

内存映射

采用栈实现：

```
int mstack[blocknr];
int mstack_pointer = blocknr - 1;
int get_mem() {
    int blocknum;
```

```

    if (mstack_pointer > 0) {

        blocknum = mstack[mstack_pointer];

        mem[blocknum] = mmap(NULL, blocksize, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0);

        memset(mem[blocknum], 0, blocksize);

        mstack_pointer--;

        return blocknum;

    }
    else {
        return -1; //若栈内已无可取元素, 返回错误信号
    }
} //取得内存

void free_mem(int blocknum) {

    munmap(mem[blocknum], blocksize);

    mem[blocknum] = NULL;

    mstack_pointer++;

    mstack[mstack_pointer] = blocknum; //被释放的元素推入栈

    return;

} //释放内存

```

修改文件的内存分配

```

int content_change(struct filenode *fnode, int size) {
//size为正: 增加分配的页数; size为负: 减少分配的页数
    int i = 0, j = 0;
    int msize = -size;
    int curr_page_num = fnode->block_num;
    int temp;
    int mem_got[blocknr];
    if (size < 0) {
        for (i = 0; i < msize; i++) {
            free_mem(fnode->content[curr_page_num - i - 1]);
        }
        fnode->block_num -= msize; //改变block数的记录
        return -ENOSPC;
    } //当要释放内存时
    else {
        for (i = 0; i < size && (fnode->block_num + i) < maxblock; i++) {

            temp = get_mem();

```

```

        if (temp == -1)break;
        else {
            mem_got[i] = temp;
        }
    } //当要扩充内存时, 不能扩充超过上限
    for (j = 0; j < i; j++) {
        fnode->content[curr_page_num + j] = mem_got[j];
    }
    fnode->block_num += i; //改变block数的记录
    return i; //返回成功取得的block数
}
}

```

写操作

```

static int oshfs_write(const char *path, const char *buf, size_t size, off_t offset, struct
fuse_file_info *fi)
//将缓冲区(buf)中的数据写到一个打开的文件中
{
    struct filenode *node = get_filenode(path); //打开文件
    int s;
    int c = (offset + size - 1) / blocksize + 1 - (node->block_num); //要增(减)的页数
    if (c < 0)c = 0; //写是不会截短的
    s=content_change(node, c); //获取成功增(减)的页数(以及增(减)页)
    if (s == c) { //得到正确结果
        node->st.st_size = offset + size;
    }
    else //错误处理
    {
        return -ENOSPC;
    }
    int f = offset / blocksize; //计算被写入的第一页的页码
    int o = offset % blocksize ; //计算从第一页哪里开始写入
    int n = (o + size - 1) / blocksize;
    int i = 0;
    char *buf0 = buf;
    if (n == 0)
    {
        memcpy(mem[node->content[f]] + o, buf0, size);
    }
    else
    {
        for (i = 0; i <= n; i++) {
            if (i == 0) {
                memcpy(mem[node->content[f]] + o, buf0, blocksize - o );
                buf0 += blocksize - o;
            }
            else if (i <= n - 1) {
                memcpy(mem[node->content[f + i]], buf0, blocksize);
                buf0 += blocksize;
            }
        }
    }
}

```

```

        else {
            memcpy(mem[node->content[f + i]], buf0, size - (long int)(buf0-buf));
        }
    }
}
return size;
}

```

读操作

```

static int oshfs_read(const char *path, char *buf, size_t size, off_t offset, struct
fuse_file_info *fi)
//从一个已经打开的文件中读出数据
{
    struct filenode *node = get_filenode(path);                //寻找对应的节点
    long int ret=size;//记录读取量
    if (offset + size > node->st.st_size) { //超量
        ret = node->st.st_size - offset;
    }
    if (ret < 0) return ret; //读错了
    int f = offset / blocksize;    //f=读起始页
    int o = offset % blocksize;    //o=页内偏移
    int n = (o + ret - 1) / blocksize; //n=被读取的页数-1
    int i = 0;
    char* buf0 = buf;
    if (n == 0) { //不需要跨页读取
        memcpy(buf, mem[node->content[f]] + o, ret);
    }
    else { //需要跨页读取
        for (i = 0; i <= n; i++) {
            if (i == 0)
            {
                memcpy(buf0, mem[node->content[f]] + o, blocksize - o);
                buf0 += blocksize - o;
            }
            else if (i <= n - 1)
            {
                memcpy(buf0, mem[node->content[f + i]], blocksize);
                buf0 += blocksize;
            }
            else
            {
                memcpy(buf0, mem[node->content[f + i]], ret - (long int)(buf0 - buf));
            }
        }
    }
    return ret;
}

```

截短操作

```

static int oshfs_truncate(const char *path, off_t size)

```

```

{
    struct filenode *node = get_filenode(path);
    int c = size / blocksize + 1 - node->block_num;//c为增（减）页数（向上取整）
    int s;//s为成功增（减）的页数
    s = content_change(node, c);
    if (s == c) { //得到正确结果
        node->st.st_size = size;
    }
    else { //错误处理
        return -ENOENT;
    }
    return 0;
}

```

其他函数都是在原来的示例代码的基础上稍加修改。

自己的测试

首先是用讲解里的测试方法

```

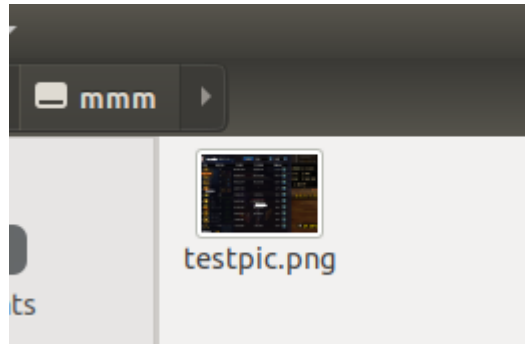
milk@ubuntu:~$ ./myfs mmm
milk@ubuntu:~$ cd mmm
milk@ubuntu:~/mmm$ echo helloworld > testfile
milk@ubuntu:~/mmm$ ls -l testfile
-rw-r--r-- 1 milk milk 11 Aug  8  4432873 testfile
milk@ubuntu:~/mmm$ cat testfile
helloworld
milk@ubuntu:~/mmm$ dd if=/dev/zero of=testfile bs=1M count=400
400+0 records in
400+0 records out
419430400 bytes (419 MB, 400 MiB) copied, 5.87288 s, 71.4 MB/s
milk@ubuntu:~/mmm$ ls -l testfile
-rw-r--r-- 1 milk milk 419430400 Aug  8  4432873 testfile
milk@ubuntu:~/mmm$ dd if=/dev/urandom of=testfile bs=1M count=1 seek=10
1+0 records in
1+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0299026 s, 35.1 MB/s
milk@ubuntu:~/mmm$ ls -l testfile
-rw-r--r-- 1 milk milk 11534336 Aug  8  4432873 testfile
milk@ubuntu:~/mmm$ dd if=testfile of=/dev/null
22528+0 records in
22528+0 records out
11534336 bytes (12 MB, 11 MiB) copied, 0.0395731 s, 291 MB/s
milk@ubuntu:~/mmm$ rm testfile
milk@ubuntu:~/mmm$ ls -al
total 4
drwxr-xr-x  0 root root    0 Dec 31  1969 .
drwxr-xr-x 26 milk milk 4096 May 12 03:35 ..
milk@ubuntu:~/mmm$

```

※因为我的文件系统只有512MB，所以有一个输入2000MiB的测试我改成了400MiB

除此之外我还写了一张图片进入mmm，能正常显示：

```
milk@ubuntu:~$ cp testpic.png mmm
milk@ubuntu:~$ cd mmm
milk@ubuntu:~/mmm$ ls
testpic.png
milk@ubuntu:~/mmm$
```



使用nano修改txt文件也能正常增删。