

一、实验目的：

现在文件系统是根据 LBA 号读写磁盘实现数据获取和写入。本项目实现将文件系统放在内存上，数据获取与写入是读写拷贝内存。其意义是可以将文件系统从磁盘驱动的限制上解放出来，不会因缺少驱动而导致文件系统不工作。

二、实验内容：

基于内存的文件系统实现。目前我们开发的文件系统 FAT32、Orange、EXT2 等最终都是驻留在硬盘上的（目前是 IDE 硬盘）。如果要把这些文件系统驻留在 U 盘（SSD 卡之类的）甚至内存，就需要编写这类存储介质的驱动程序，而文件系统的代码理论上不需要修改。这个实验需要解决两个问题：

（1）确认目前 miniOS 中文件系统和设备（驱动程序）之间的数据结构设计是否能够适应（满足）文件系统可以驻留在任何存储介质上（只需要开发出相关的驱动）。如果不适应，需要如何修改？

（2）如何实现基于内存的文件系统，实际上就是如何实现对内存的驱动。想想这应该比较简单，实现两个函数，读内存扇区和写内存扇区，实际上就是读写内存的 512 个字节。细节问题是，如何在内核空间申请一块连续的逻辑空间，比如说 1M，把这块内存空间作为文件系统的存储空间。

三、项目需求及分析：

一个字符设备或块设备都有一个主设备号（major）和一个次设备号（minor）。主设备号用来标识与设备文件相连的驱动程序，用来反映设备类型。次设备号被驱动程序用来辨别操作的是哪个设备，用来区分同类型的设备。（拥有相同驱动程序的设备的主设备号是相同的，次设备号用于区分其中的二级设备。例如，X86 系统的 IDE 硬盘的驱动程序是一样的，就可以把他们的主设备号看成一样的。每一块物理硬盘，又可以分成几个分区，每一个分区就可以看成一个“独立的”设备，这些独立的设备就是采用次设备号来区分的。如果系统中另外还有一块 SSD 盘，它的主设备号就应该是另外一个号，因为对 SSD 的驱动与 IDE 是不一样的。但是对于一个文件系统来说（例如 FAT32），它既可以驻留在 IDE 盘上，也可以驻留在 SSD 盘上，甚至可以驻留在内存中——内存文件系统，所以，文件系统的代码不需要修改，但是向下对接的是不同的驱动程序——通过主设备号来区分。）

Linux 内核中，使用 dev_t 来描述设备号。

```
typedef u_long dev_t; // 在 32 位机中是 4 个字节，高 12 位表示主设备号，低 20 位表示次设备号。
```

Linux 内核中提供以下几个宏来操作 dev_t：

```
#define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS)) // 从设备号中提取主设备号
#define MINOR(dev) ((unsigned int) ((dev) & MINORMASK)) // 从设备号中提取次设备号
#define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi)) // 将主、次设备号拼凑为设备号
```

主要思想是：新建一个主设备号，即内存设备；和其下的若干的从设备号，然后实现对“内存设备扇区”的读写。当然还要包括内存文件存储区的分配。

miniOS 的“设备”实际上是指的一个主设备号下面的一个具体的从设备。所以，miniOS 是有主、从设备之分的。一个设备号 dev 被分成了两部分，貌似高 8 位是主设备号，低 8 位是从设备号。主设备号相同就是同一种驱动程序，很多函数可以直接用。

但是，目前的 miniOS 的 driver 含义要窄一些，只是 IDE 硬盘的一个通道（IDE 有两个通道，所以目前 miniOS 的#define MAX_DRIVES 2），每个通道最多可以挂两块硬盘。目前的想法是，每个 driver 是一类驱动程序，比如 SATA、软盘、甚至内存等，与目前 miniOS 的主设备号一致，如下：

```
/* major device numbers (corresponding to kernel/global.c::dd_map[]) */
#define NO_DEV 0
#define DEV_FLOPPY 1
#define DEV_CDROM 2
#define DEV_HD 3
#define DEV_CHAR_TTY 4
#define DEV_SCSI 5
```

可以在这里添加一个主设备号 DEV_MEM 6:

```
/* make device number from major and minor numbers */
#define MAJOR_SHIFT 8
#define MAKE_DEV(a,b) ((a << MAJOR_SHIFT) | b)

/* separate major and minor numbers from device number */
#define MAJOR(x) ((x >> MAJOR_SHIFT) & 0xFF)
#define MINOR(x) (x & 0xFF)
```

目前还需要考虑文件系统向下与驱动程序的接口。计划设置两个数据结构，为了区分目前 miniOS 中 drive 的概念，把操作系统向下对驱动程序的数据结构称为“物理设备的驱动”：

```
#define NR_PHY_DEV_DRV 10 // 暂定为 10
struct t_phy_device_driver
{
    char driver_name[14];
    u8 flag; // 表示是否空闲?
    u8 type_driver;
    // 物理设备类型，驱动程序编码，也可以理解为主设备号。例如，DRV_IDE_HD（如果想把 IDE 的四块硬盘看作是四个驱动程序
    // 的话，可以定义四个常量 DRV_PRIMARY_MASTER 等，占据 IDE_HD 的 4 项），DRV_SATA_HD 等。
    // 代码可以据此判断调用相应的驱动程序。
    struct driver_op * drv_op;
    // 指向相应的驱动程序结构的指针
}
t_phy_device_driver phy_device_driver_table[NR_PHY_DEV_DRV]
```

驱动程序类型可以定义：

```
#define NO_DRV 0
#define DRV_FLOPPY 1
#define DRV_CDROM 2
#define DRV_IDE_HD 3 // 目前已经实现 IDE 驱动
#define DRV_CHAR_TTY 4
#define DRV_SCSI_HD 5
#define DRV_SATA_HD 6
#define DRV_MEM 7 // 对应我们要编写的驱动器类型
struct t_phy_driver_op
{
    int (*open_drv) (const char*, int, int); // 打开设备驱动程序，是否需要？细化参数
    int (*close_drv) (int); // 关闭设备驱动程序，是否需要？细化参数
    int (_read_sector) (int, void *, int); // 读一个扇区，指向函数的指针，指向具体的读扇区函数
    int (_write_sector) (int, void *, int); // 写一个扇区，指向函数的指针，指向具体的写扇区函数
}
```

注：

```
int *read_sector (int dev, void *buf ,int nr_sector);
// 从设备 dev 上读第 nr_sector 个扇区到 buf 中, buf 的大小不能少于一个扇区的大小, 512 字节
int *write_sector (int dev, void *buf ,int nr_sector);
// 把 buf 中数据, 写到设备 dev 上的第 nr_sector 个扇区中
int (_open_drv) (const char_ name ,int size, int drv_type);
// 打开或建立 drv_type 类型的驱动程序, 名字是 name, 大小是 size。返回值是设备名 dev
int (_close_drv) (int dev);
```

内存文件操作的驱动, 需要实现诸如下面的函数:

```
int read_sector_mem (int dev, void *buf ,int nr_sector);
int write_sector_mem (int dev, void *buf ,int nr_sector);

int open_drv (const char_* drv_name, int size, int drv_type);
// 打开设备驱动程序或者激活设备驱动程序。
//drv_name 是设备的名称,
/*size 是设备的大小, 以字节为单位, 应该是扇区长度 (512) 的倍数, 不是整数倍的话需要扩展到整数倍。
按照这个数字 (在内核中) 申请一段逻辑地址连续的内存, 把相关参数填写到对应的 dev_mem_table 表中*/
//drv_type 是指驱动程序类型, 具体见前面的宏, 比如, 内存的驱动类型是 DRV_MEM
```

驱动程序的编程者, 需要为每个物理设备编写两个 (或四个) 相应的驱动程序, 例如, 这对返回值就是设备号 (包含主设备号和从设备号), 定义如下:

```
struct t_phy_driver_op phy_driver_op_table[NR_PHY_DEV_DRV]
/*对每一个物理设备还需要一个相应的物理设备表, 有点像 IDE 硬盘的 hd_info[]。表中的每一项,
代表一个该物理设备的从设备。例如, 对于内存来讲: */
struct t_dev_mem
{
U32 start_mem;
U32 end_mem;
U32 size;
}
```

struct t_dev_mem dev_mem_table[3] 表示最多可以建立 3 个内存文件系统 (数字可修改), 每一个驻留一个文件系统。内存设备的从设备号就是这张表的下标 (索引号)。

再来看一下文件系统在硬盘上的运行过程。

首先是文件系统的初始化, 这一过程中, 与要用到根节点的信息, 根节点在 init_fs 中定义:

```
root_inode = get_inode(orange_dev, ROOT_INODE);
```

即将 orange 分区内的第一个文件的节点分给根节点。

初始化文件系统的过程中, 需要先获得 orange 文件系统对应的分区:

```

for (i = 0; i < NR_PRIM_PER_DRIVE; i++)
{
    if (hd_info[drive].primary[i].fs_type == fs_type)
        return ((DEV_HD << MAJOR_SHIFT) | i);
}

//added by mingxuan 2020-10-29
for (i = 0; i < NR_SUB_PER_DRIVE; i++)
{
    if (hd_info[drive].logical[i].fs_type == fs_type)
        // logic的下标i加上hd1a才是该逻辑分区的次设备号
        return ((DEV_HD << MAJOR_SHIFT) | (i + MINOR_hd1a));
}

```

这里初始化过程中，将 superblock 读到缓冲区中，然后从缓冲区中读出 superblock，这里可能作用是试一下读写是否正常，暂时没发现什么实际意义。

读 superblock 到缓冲区的过程：

superblock 是分区中的第二个扇区，这里直接手动设置了读取信息：

```

int i;
MESSAGE driver_msg;
char fsbuf[SECTOR_SIZE]; //local array, to substitute global fsbuf. added by xw, 18/12/27

driver_msg.type = DEV_READ;
driver_msg.DEVICE = MINOR(dev);
driver_msg.POSITION = SECTOR_SIZE * 1;
driver_msg.USEMEM = USE_MEM; //added by wangshining 2021-12-28
driver_msg.BUF = fsbuf;
driver_msg.CNT = SECTOR_SIZE;
driver_msg.PROC_NR = proc2pid(p_proc_current); ///TASK_A
hd_rdwt(&driver_msg);
for (i = 0; i < NR_SUPER_BLOCK; i++)
    if (super_block[i].fs_type == ORANGE_TYPE)
        break;
if (i == NR_SUPER_BLOCK)
    disp_str("Cannot find orange's superblock!");
// assert(i == 0); /* currently we use only the 1st slot */

struct super_block *psb = (struct super_block *)fsbuf;
super_block[i] = *psb;

super_block[i].sb_dev = dev;
super_block[i].fs_type = ORANGE_TYPE;

```

然后就是调用 mkfs () 函数来建立 superblock、建立 node-map、建立 sect-map 和写入节点信息。函数中调用硬盘驱动中的函数来获取分区信息：

```

// assert(dd_map[MAJOR(ROOT_DEV)].driver_nr != INVALID_DRIVER);
// send_recv(BOTH, dd_map[MAJOR(ROOT_DEV)].driver_nr, &driver_msg);
hd_ioctl(&driver_msg);

```

之后根据分区信息，设定超级块的信息。这里的实现方法是先建立一个超级块的数据结构，并给各种属性赋值，然后写入第一个扇区：

```

sb.magic = MAGIC_V1;
sb.nr_inodes = bits_per_sect;
sb.nr_inode_sects = sb.nr_inodes * INODE_SIZE / SECTOR_SIZE;
sb.nr_sects = driver_msg.USEM ? meo.size : geo.size; /* partition size in sector */ //modified by w
sb.nr_imap_sects = 1;
sb.nr_smap_sects = sb.nr_sects / bits_per_sect + 1;
sb.n_1st_sect = 1 + 1 + /* boot sector & super block */
                sb.nr_imap_sects + sb.nr_smap_sects + sb.nr_inode_sects;
sb.root_inode = ROOT_INODE;
sb.inode_size = INODE_SIZE;
struct inode x;
sb.inode_isize_off = (int)&x.i_size - (int)&x;
sb.inode_start_off = (int)&x.i_start_sect - (int)&x;
sb.dir_ent_size = DIR_ENTRY_SIZE;
struct dir_entry de;
sb.dir_ent_inode_off = (int)&de.inode_nr - (int)&de;
sb.dir_ent_fname_off = (int)&de.name - (int)&de;
sb.sb_dev = orange_dev; //added by mingxuan 2020-10-30
sb.fs_type = ORANGE_TYPE; //added by mingxuan 2020-10-30
memset(fsbuf, 0x90, SECTOR_SIZE);
memcpy(fsbuf, &sb, SUPER_BLOCK_SIZE);
WR_SECT(orange_dev, 1, fsbuf); // modified by mingxuan 2020-10-27

```

然后是对 node-map 进行赋值，这里也进行了一次写入：

```

for (i = 0; i < (NR_CONSOLES + 3); i++) //modified by mingxuan 2019-5-22
    fsbuf[0] |= 1 << i;
WR_SECT(orange_dev, 2, fsbuf); //modified by mingxuan 2020-10-27

```

后面是对 sect-map 进行初始化。这里 sect-map 占用了 nr_imap_sects 个扇区，然后将根目录占用的位设置为 1，其余为 0：

```

for (i = 0; i < nr_sects / 8; i++)
    fsbuf[i] = 0xFF;

for (j = 0; j < nr_sects % 8; j++)
    fsbuf[i] |= (1 << j);

//WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects, fsbuf); //modified by xw, 18/12/27 //deleted by mingxuan 20
WR_SECT(orange_dev, 2 + sb.nr_imap_sects, fsbuf); //modified by mingxuan 2020-10-27
/* zero memory the rest sector-map */
memset(fsbuf, 0, SECTOR_SIZE);
for (i = 1; i < sb.nr_smap_sects; i++)
    //WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + i, fsbuf); //modified by xw, 18/12/27 //deleted by mi
    WR_SECT(orange_dev, 2 + sb.nr_imap_sects + i, fsbuf); //modified by mingxuan 2020-10-27

```

然后是记录加载进去的文件(app.rar)的信息，主要是扇区占用信息，节点占用信息上面已经更新过：

```

int bit_offset = INSTALL_START_SECTOR -
                sb.n_1st_sect + 1; /* sect M <-> bit (M - sb.n_1stsect + 1) */
int bit_off_in_sect = bit_offset % (SECTOR_SIZE * 8);
int bit_left = INSTALL_NR_SECTORS;
int cur_sect = bit_offset / (SECTOR_SIZE * 8);
RD_SECT(orange_dev, 2 + sb.nr_imap_sects + cur_sect, fsbuf); //modified by mingxuan 2020-10-27
while (bit_left)
{
    int byte_off = bit_off_in_sect / 8;
    /* this line is ineffecient in a loop, but I don't care */
    fsbuf[byte_off] |= 1 << (bit_off_in_sect % 8);
    bit_left--;
    bit_off_in_sect++;
    if (bit_off_in_sect == (SECTOR_SIZE * 8))
    {
        WR_SECT(orange_dev, 2 + sb.nr_imap_sects + cur_sect, fsbuf); //modified by mingxuan 2020-10-27
        cur_sect++;
        RD_SECT(orange_dev, 2 + sb.nr_imap_sects + cur_sect, fsbuf); //modified by mingxuan 2020-10-27
        bit_off_in_sect = 0;
    }
}

```

接下来给节点信息赋值。这里包括根目录节点, dev_tty0, dev_tty1, dev_tty2, app.tar 文件:

```
for (i = 0; i < NR_CONSOLES; i++)
{
    pi = (struct inode *) (fsbuf + (INODE_SIZE * (i + 1)));
    pi->i_mode = I_CHAR_SPECIAL;
    pi->i_size = 0;
    pi->i_start_sect = MAKE_DEV(DEV_CHAR_TTY, i);
    pi->i_nr_sects = 0;
}

/* inode of /app.tar */
//added by mingxuan 2019-5-19
pi = (struct inode *) (fsbuf + (INODE_SIZE * (NR_CONSOLES + 1)));
pi->i_mode = I_REGULAR;
pi->i_size = INSTALL_NR_SECTORS * SECTOR_SIZE;
pi->i_start_sect = INSTALL_START_SECTOR;
pi->i_nr_sects = INSTALL_NR_SECTORS;
//WR_SECT(ROOT_DEV, 2 + sb.nr_imap_sects + sb.nr_smap_sects, fsbuf); //modified by xw, 18/12/27 //d
WR_SECT(orange_dev, 2 + sb.nr_imap_sects + sb.nr_smap_sects, fsbuf);
```

最后, 将 orange 文件系统的 superblock 读到缓冲区, 并给根目录赋值一个节点。至此, 文件初始化完成。

第二步是打开文件过程。文件系统中, 每个文件都有一个文件描述符 filp, 它代表了一个文件, 当你打开一个文件的时候, 文件系统会先查找空闲的 filp 来进行分配。

```
int i;
//for (i = 0; i < NR_FILES; i++) {
/* 0, 1, 2 are reserved for stdin, stdout, stderr. modified by xw, 18/8/28 */
//for (i = 3; i < NR_FILES; i++) { //deleted by mingxuan 2019-5-20
for (i = 0; i < NR_FILES; i++)
{ //modified by mingxuan 2019-5-20
    if (p_proc_current->task.filp[i] == 0)
    {
        fd = i;
        break;
    }
}
```

每个 filp 都有一个指针指向 inode, 这个结构体存储了文件在文件系统中的位置、文件占用分区大小等信息, 这个位置信息是以分区为基准的相对位置。

```
openstruct inode {
    u32 i_mode;    /**< Access mode */
    u32 i_size;    /**< File size */
    u32 i_start_sect; /**< The first sector of the data */
    u32 i_nr_sects; /**< How many sectors the file occupies */
    u8 _unused[16]; /**< Stuff for alignment */

    /* the following items are only present in memory */
    int i_dev;
    int i_cnt;    /**< How many procs share this inode */
    int i_num;    /**< inode nr. */
};
```

查找文件时, 会先从目录节点中读取其中存储文件名的扇区, 来进行对比, 查找文件的 index, 即在文件目录中是第几个文件。


```

for (i = 0; i < nr_dir_blks; i++)
{
    //RD_SECT_SCHED(dir_inode->i_dev, dir_blk0_nr + i, fsbuf); //modified by xw, 18/12/27
    RD_SECT(dir_inode->i_dev, dir_blk0_nr + i, fsbuf); //modified by mingxuan 2019-5-20
    pde = (struct dir_entry *)fsbuf;
    for (j = 0; j < SECTOR_SIZE / DIR_ENTRY_SIZE; j++, pde++)
    {
        if (memcmp(filename, pde->name, MAX_FILENAME_LEN) == 0)
            return pde->inode_nr;
        if (++m > nr_dir_entries)
            break;
    }
    if (m > nr_dir_entries) /* all entries have been iterated */
        break;
}

```

查找到文件后，会给查找文件的 inode，如果没有分配，则新建一个 inode 进行分配。

```

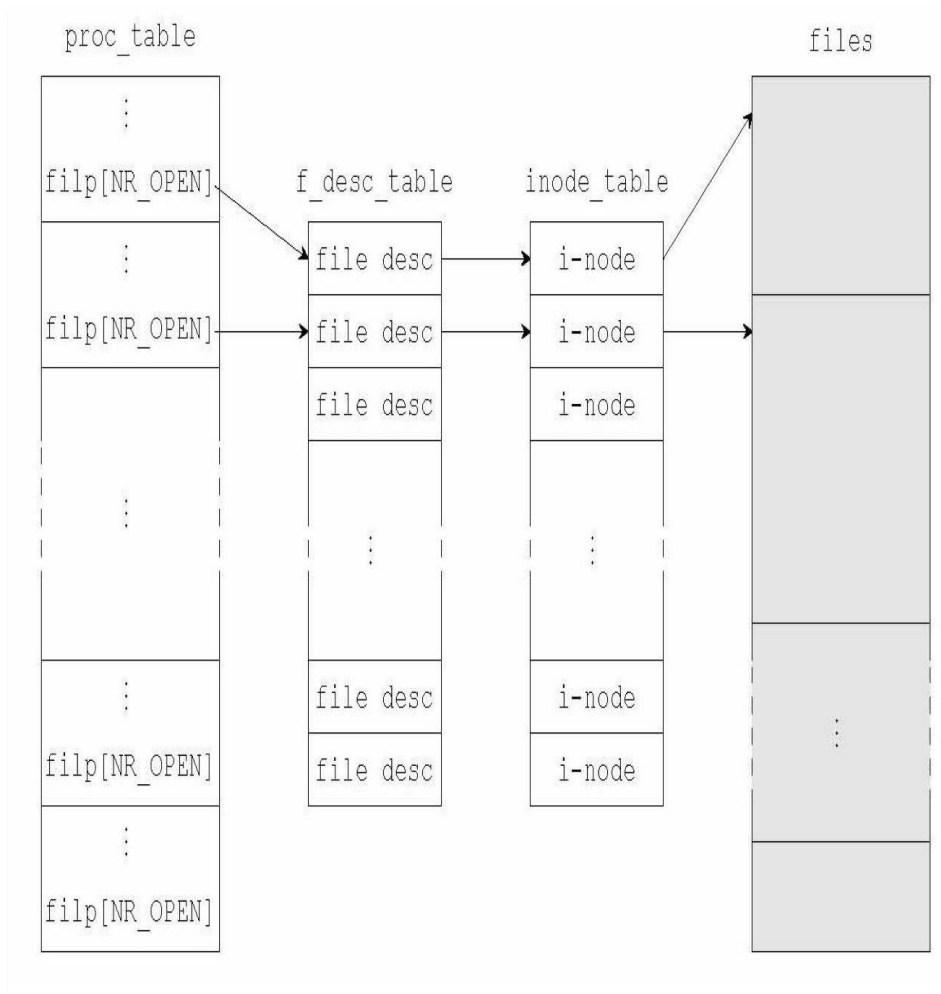
for (p = &inode_table[0]; p < &inode_table[NR_INODE]; p++)
{
    if (p->i_cnt)
    { /* not a free slot */
        if ((p->i_dev == dev) && (p->i_num == num))
        {
            /* this is the inode we want */
            p->i_cnt++;
            return p;
        }
    }
    else
    { /* a free slot */
        if (!q) /* q hasn't been assigned yet */
            q = p; /* q <- the 1st free slot */
    }
}
if (!q)
    disp_str("Panic: the inode table is full");

q->i_dev = dev;
q->i_num = num;
q->i_cnt = 1;

struct super_block *sb = get_super_block(dev);
int blk_nr = 1 + 1 + sb->nr_imap_sects + sb->nr_smap_sects + ((num - 1) / (SECTOR_SIZE / INODE_SIZE));
char fsbuf[SECTOR_SIZE]; //local array, to substitute global fsbuf. added by xw, 18/12/27
RD_SECT(dev, blk_nr, fsbuf); //added by xw, 18/12/27
struct inode *pinode =
    (struct inode *)((u8 *)fsbuf +
        ((num - 1) % (SECTOR_SIZE / INODE_SIZE)) * INODE_SIZE);
q->i_mode = pinode->i_mode;
q->i_size = pinode->i_size;
q->i_start_sect = pinode->i_start_sect;
q->i_nr_sects = pinode->i_nr_sects;

```

inode、flip、文件和进程表之间的关系如下图所示，每一个进程中有一个 filp 表，里面对应着文件描述符 filp，每个 filp 数据结构中有一个指向 inode 的指针，每个 inode 存储着文件的位置信息。



第三部分是文件读写过程，打开文件后，会返回一个文件标识数组的下标 fd，文件读写过程中，可以通过 fd 找到文件对应的描述结构体 filp；通过文件描述符，得到文件位置、node 和 mode。

```

int pos = p_proc_current->task.filp[fd]->fd_pos;
struct inode *pin = p_proc_current->task.filp[fd]->fd_node.fd_inode; //modified by mingxuan 2019-5-17
int imode = pin->i_mode & I_TYPE_MASK;
  
```

根据 mode 来判断进行读写的是设备还是文件，如果是文件，则先根据读写模式结算出读写的终止地址。通过地址可以结算出文件在硬盘中的位置和读写的扇区大小；调用 rw_sector 在硬盘中进行读写。

四、具体实现：

【总体思路】文件系统的读写是在硬盘上还是在内存上，根本上还是存储介质驱动层的改动，因为文件系统中信息传递和地址解算都是以相对分区的扇区地址为参考，与在存储介质上的地址基本无关，是在读写时才加上分区的基地址，所以最重要的部分是在内存上读写的函数和分配内存的函数，文件系统方面基本不用修改。

【实现内容】

1. 通过文件前缀名对文件系统进行划分，通过设备号进行文件系统的初始化划分。

在调用的时候需要让文件系统知道它是在哪个存储设备上运行的，这样它才能把相应的目录节点分配出来，但是因为 vfs 中调用的文件系统中的驱动并不知道是谁调用的自己，这个文件系统的 superblock 信息是存储在 vfs 中的，所以需要给它一个让它能找到这个 superblock 的方法。一种方法是直接把 superblock 传进去，这样的话需要修改文件的输入；另一种实现

方法(我们使用的方法)是把它的文件前缀名传进去,这样能避免一次信息传递,但是也增加了一些查找方面的消耗。

2. 调用 `do_malloc_4K` 分配 1MB 空间,然后将物理地址转换为线性地址。

由于是分配 1M 的空间,所以会调用 `do_malloc_4K` 这个函数 256 次,分配完之后,会得到一系列这个分块的起始的物理地址,然后再调用 `K_PHY2LIN` 将物理地址转换为线性地址,把线性地址也存储下来。

在内存中进行读写,主要解决一个跨内存块的问题,因为它是一次 4K 来进行分配的,所以有可能它是不连续的,所以当读取的时候如果跨了两个内存块就需要进行一些相应的操作。

3. 在 `vfs` 中添加一种文件系统,保证普通文件系统和内存文件系统共存。

本质工作就是新添加一个文件系统。

【修改部分展示】

1. 定义内存文件系统相关的数据结构

`t_dev_mem` 是内存分区的信息, `t_mem_info` 是某一个内存块的信息, `mem_fs` 是分配的内存块的信息。

```
OS > miniOS-v1.3.2 > include > h mem.h > ...
1  #ifndef _ORANGES_MEM_H_
2  #define _ORANGES_MEM_H_
3  //added by wangshining 2021-12-26
4  #define NR_MEM_PER_DRIVE 3
5  #define MEM_FS_SIZE 256
6  struct t_dev_mem
7  {
8      u32 start_mem;
9      u32 fs_type;
10     u32 size;
11 };
12
13 struct t_mem_info
14 {
15     void *base;
16     int size;
17     int use_count;
18     struct t_dev_mem dev_mem_table[NR_MEM_PER_DRIVE];
19 };
20 struct mem_fs
21 {
22     int lin_addr;
23     void *phy_address;
24 } mem_fs[260];
25 PUBLIC int read_sector_mem(void *buf, int position, int size);
26 PUBLIC int write_sector_mem(void *buf, int position, int size);
```

2. 声明为全局变量

定义一个 memory 设备为全局变量,后面主要是对它进行操作;内存块也设定一个全局变量。

```
OS > miniOS-v1.3.2 > kernel > C hd.c > out_hd_queue(HDQueue *, RWInfo **)
31 PRIVATE u8 membuf[num_4M];
32 //PRIVATE struct hd_info hd_info[1];
33 PUBLIC struct hd_info hd_info[1]; //modified by mingxuan 2020-10-27
34 PUBLIC struct t_mem_info mem_info; //add by wangshining 2021-12-25
```

```
OS > miniOS-v1.3.2 > include > h global.h > ...
58 extern struct t_mem_info mem_info; //added by wangshining 2021-12-26
59 extern struct mem_fs mem_fs[260]; //added by wangshining 2021-12-28
```

3. 声明内存文件系统相关函数，函数实现在 mem.c 中：

```
OS > miniOS-v1.3.2 > include > h mem.h > ...
28 /*add by wangshining 2021-12-20*/
29
30 PUBLIC void init_mem();
31 PUBLIC void mem_open();
32 PUBLIC void mem_close();
33 PUBLIC void mem_service();
34 PUBLIC void mem_rdwt(MESSAGE *p);
35 #endif
```

4. 初始化文件系统：

主要操作是为文件系统分配空间并划分分区，分配空间后将物理地址转换为线性地址。对分区进行初始化，包括将第二个分区设为 orange 分区，分配了 1024 个扇区，第一个分区的 start_mem 和 size 是 0，没有文件系统在上面。

```
OS > miniOS-v1.3.2 > kernel > C mem.c > ...
25 /*****
26 *                               init_mem
27 *****/
28 PUBLIC void init_mem()
29 {
30     /* add by wangshining 2021-12-20*/
31
32     memset(&mem_info, 0, sizeof(mem_info));
33
34     mem_info.size = 2048;
35     mem_info.use_count = 0;
36     int i, tempaddr;
37     int phy_addr;
38     tempaddr = MEMFSLinBase;
39     for (i = 0; i < MEM_FS_SIZE; i++, tempaddr += num_4K)
40     {
41         mem_fs[i].lin_addr = tempaddr;
42         mem_fs[i].phy_address = do_kmalloc_4k();
43         phy_addr = mem_fs[i].phy_address;
44         //deleted by wngshining 2021-12-28
45         // disp_str("mem_start:");
46         // disp_int(i);
47         // disp_str(" ");
48         // disp_int(mem_fs[i].lin_addr);
49         // disp_str(" ");
50         // disp_int(phy_addr);
51         // disp_str("\n");
```

```
52     lin_mapping_phy(mem_fs[i].lin_addr, phy_addr,  
53                     p_proc_current->task.pid, PG_P | PG_USU | PG_RWW, PG_P | PG_USU | PG_RWW);  
54  
55     mem_info.base = (void *)((MEMFSLinBase)&0xFFFF000);  
56     for (i = 1; i < NR_MEM_PER_DRIVE; i++)  
57     {  
58         mem_info.dev_mem_table[i].start_mem = (i - 1) * 1024;  
59         mem_info.dev_mem_table[i].size = 1024;  
60     }  
61     mem_info.dev_mem_table[1].fs_type = ORANGE_TYPE;  
62     mem_info.dev_mem_table[2].fs_type = FAT32_TYPE;  
63     mem_info.dev_mem_table[0].start_mem = 0;  
64     mem_info.dev_mem_table[0].size = 0;  
65     mem_info.dev_mem_table[0].fs_type = NO_FS_TYPE;  
66 }
```

5. 仿照硬盘编写，service 函数用于输出内存分区信息：

其实除了我们的写法，还可以改成：当不使用它的时候，把分配的空间释放掉，这样可能会更好一点。

```
OS > miniOS-v1.3.2 > kernel > C mem.c > ...  
61  /*add by wangshining 2021-12-26 */  
62  PUBLIC void mem_open()  
63  {  
64      if (mem_info.use_count++ == 0)  
65      {  
66          mem_service();  
67      }  
68      return;  
69  }
```

```
OS > miniOS-v1.3.2 > kernel > C mem.c > mem_service()  
70  /*add by wangshining 2021-12-26 */  
71  PUBLIC void mem_close()  
72  {  
73      mem_info.use_count--;  
74      return;  
75  }
```

6. 解析文件系统信息：

包括分区的起始地址、设备的起始地址、大小等。

```
OS > miniOS-v1.3.2 > kernel > mem.c > mem_service()
76  /*add by wangshining 2021-12-26 */
77  PUBLIC void mem_service()
78  {
79      disp_str("mem_base:");
80      disp_int(mem_info.base);
81      disp_str("\n");
82      disp_str("mem_size:");
83      disp_int(mem_info.size);
84      disp_str("\n");
85      int i;
86      for (i = 0; i < NR_MEM_PER_DRIVE; i++)
87      {
88          disp_str("PART_");
89          disp_int(i);
90          disp_str(": base ");
91          disp_int(mem_info.dev_mem_table[i].start_mem);
92          disp_str(", size ");
93          disp_int(mem_info.dev_mem_table[i].size);
94          disp_str("\n");
95      }
96      return;
97  }
```

7. 对内存进行读写:

因为内存块是每 4K 一个块，一个 4K 是 8 个扇区，所以每读 8 个扇区就要进行一个操作，首先 position 需要除以 8 来找到它属于哪个内存块，然后再从内存块中找它使要从哪个扇区开始读，然后将 buf 从虚拟地址转变成线性地址(内存块存储的也是线性地址)，每次最多读 8 块，每读完一次后块的序列号就要+1，然后继续读。

```

OS > miniOS-v1.3.2 > kernel > mem.c > read_sector_mem(void *, int, int, int)
106 PUBLIC int read_sector_mem(void *buf, int position, int size, int proc_nr)
107 {
108     int mem_p = position >> 3;
109     int off = position % 8;
110     int *p = mem_fs[mem_p].lin_addr;
111     p += (off * 512) >> 2;
112     int size_left = size >> SECTOR_SIZE_SHIFT;
113     int i = 0;
114     int size_i = 0;
115     void *la = (void *)va2la(proc_nr, buf);
116     while (size_left)
117     {
118         int left_sect = min(8, size_left);
119         if (i == 0)
120         {
121             left_sect = min(8 - off, size_left);
122             //disp_int(left_sect);
123             memcpy(la, p, left_sect * 512);
124         }
125         else
126         {
127             p = mem_fs[mem_p + i].lin_addr;
128             la = (void *)va2la(proc_nr, buf + size_i);
129             memcpy(la, p, left_sect * 512);
130         }
131         size_i += left_sect * 512;
132         i++;
133         size_left -= left_sect;
134     }
135     return 1;
136 }

```

8. 驱动层相关改动，首先是进行地址解算

把文件系统分区的信息读出来，主要是存储分区的大小信息。读写过程中进行地址解算，传进来的地址是已经转换成字节的地址，需要再转换成扇区的地址，加上文件系统运行的分区的起始地址，使得后面能进一步的进行读写。

```

OS > miniOS-v1.3.2 > kernel > hd.c > hd_ioctl(MESSAGE *)
408 if (p->REQUEST == DIOCTL_GET_GEO)
409 {
410     //added by wangshining 2021-12-26
411     //这里跟hd_rdwt的改动方法一样，在这里加个分支
412     if (p->USEMEM)
413     {
414         void *dst = va2la(p->PROC_NR, p->BUF);
415         void *src = va2la(proc2pid(p_proc_current), &memi->dev_mem_table[device]);
416         phys_copy(dst, src, sizeof(struct t_dev_mem));
417     }
418     else
419     {
420         void *dst = va2la(p->PROC_NR, p->BUF);
421         void *src = va2la(proc2pid(p_proc_current),
422             device < MAX_PRIM ? &hdi->primary[device] : &hdi->logical[(device - MINOR_hd1a) % N
423         );
424         phys_copy(dst, src, sizeof(struct part_info));
425     }
426 }
427 }
428 }

```

```

OS > miniOS-v1.3.2 > kernel > hd.c > hd_rdwt(MESSAGE *)
137     u64 pos = p->POSITION;
138
139     //We only allow to R/W from a SECTOR boundary:
140
141     u32 sect_nr = (u32)(pos >> SECTOR_SIZE_SHIFT); // pos / SECTOR_SIZE
142
143     int logidx = (p->DEVICE - MINOR_hd1a) % NR_SUB_PER_DRIVE;
144     sect_nr += p->DEVICE < MAX_PRIM ? hd_info[drive].primary[p->DEVICE].base : hd_info[drive].logical[logidx].ba
145     //这里还是以扇区为单位进行读写，开始读的地址为t_mem_info.base + sect_mt * SECTOR_SIZE
146     u32 sect_mt = (u32)(pos >> SECTOR_SIZE_SHIFT);
147     void *la = (void *)va2la(p->PROC_NR, p->BUF);
148     struct hd_cmd cmd;
149     if (p->USEMEM)
150     {
151         sect_mt += mem_info.dev_mem_table[p->DEVICE].start_mem;
152     }
153     /*
154     这里由于传进来的position是 扇区地址*SECTOR_SIZE
155     并且文件系统的super block、i_node map也是通过这个函数进行读写操作的，所以在这个函数内进行分支，调用接口从内存中
156     */

```

9. 正式的内存读写部分：

传入 memory 的 flag (USEMEM)，当其值为 1 时进行读写，从内存中把信息读取到 membuf 缓冲区中，函数及那个数据读取到缓冲区中，然后从缓冲区中复制到进程位置处。

```

OS > miniOS-v1.3.2 > kernel > hd.c > hd_rdwt(MESSAGE *)
158     if (p->USEMEM)
159     {
160         if (p->type == DEV_READ)
161         {
162             read_sector_mem(membuf, sect_mt, p->CNT, p->PROC_NR);
163             phys_copy(la, membuf, p->CNT);
164         }
165         else
166         {
167             phys_copy(membuf, la, p->CNT);
168             write_sector_mem(membuf, sect_mt, p->CNT, p->PROC_NR);
169         }
170     }
171     else
172     {
173         cmd.features = 0;
174         cmd.count = (p->CNT + SECTOR_SIZE - 1) / SECTOR_SIZE;
175         cmd.lba_low = sect_nr & 0xFF;
176         cmd.lba_mid = (sect_nr >> 8) & 0xFF;
177         cmd.lba_high = (sect_nr >> 16) & 0xFF;
178         cmd.device = MAKE_DEVICE_REG(1, drive, (sect_nr >> 24) & 0xF);
179         cmd.command = (p->type == DEV_READ) ? ATA_READ : ATA_WRITE;
180         hd_cmd_out(&cmd);
181     }

```

10. 文件系统相关改动：

主要是初始化的改动和读写的一小部分改动。初始化的改动有三种方式，分别是：修改 init、修改 mkfs 和增加 flag。由于内存文件系统和正常的 orange 系统的初始化代码有很大程度上的重合性，所以最好的方法是能够用一个函数给不同的参数来初始化，但是这样的实现会比较复杂，我们这里选择了公用 mkfs 函数的方法，并新添加了一个 init_fs_mem()，不同主要在于两点，第一点是获得分区的时候传递的设备是 MEM_MASTER (第 122 行)，第二点是判断的时候，当主设备号不同的时候执行不同的 mkfs，(第 128 行和第 134 行)。


```

OS > miniOS-v1.3.2 > kernel > C mem_fs.c > ...
116 PUBLIC void init_fs_mem()
117 {
118
119     disp_str("Initializing file system... \n");
120     int i;
121     struct super_block *sb = super_block; //deleted by mingxuan 2020-10-30
122     int orange_dev = get_fs_dev(MEM_MASTER, ORANGE_TYPE); //added by mingxuan 2020-10-27
123     read_super_block(orange_dev); // modified by mingxuan 2020-10-27
124     sb = get_super_block(orange_dev); // modified by mingxuan 2020-10-27
125     disp_str("Superblock Address:");
126     disp_int(sb);
127     disp_str(" \n");
128     if (sb->magic != MAGIC_V1 && MAJOR(sb->sb_dev) == DEV_HD)
129     { //deleted by mingxuan 2019-5-20
130         mkfs(0);
131         disp_str("Make file system Done.\n");
132         read_super_block(orange_dev); // modified by mingxuan 2020-10-27
133     }
134     else if (sb->magic != MAGIC_V1 && MAJOR(sb->sb_dev) == DEV_MEM)
135     {
136         mkfs(1);
137         disp_str("Make file system Done.\n");
138         read_super_block(orange_dev); // modified by mingxuan 2020-10-27
139     }
140     //root_inode = get_inode(ROOT_DEV, ROOT_INODE); // deleted by mingxuan 2020-10-27
141     mem_root_inode = get_inode(orange_dev, ROOT_INODE); // modified by mingxuan 2020-10-27
142 }

```

11. get_fs_dev 相关改动:

根据设备 drive 来划分, 不需要额外增加 flag (第 81 行), 当 drive 为 MEM_MASTER 时就会从内存中寻找分区而不是从硬盘中寻找。

```

OS > miniOS-v1.3.2 > kernel > C mem_fs.c > get_fs_dev(int, int)
78 int get_fs_dev(int drive, int fs_type)
79 {
80     int i = 0;
81     if (drive == MEM_MASTER)
82     {
83         for (i = 0; i < NR_MEM_PER_DRIVE; i++)
84         {
85             if (mem_info.dev_mem_table[i].fs_type == fs_type)
86             {
87                 return ((DEV_MEM << MAJOR_SHIFT) | i);
88             }
89         }
90     }
91     else
92     {
93         for (i = 0; i < NR_PRIM_PER_DRIVE; i++)
94         {
95             if (hd_info[drive].primary[i].fs_type == fs_type)
96                 return ((DEV_HD << MAJOR_SHIFT) | i);
97         }
98         //added by mingxuan 2020-10-29
99         for (i = 0; i < NR_SUB_PER_DRIVE; i++)
100         {
101             if (hd_info[drive].logical[i].fs_type == fs_type)
102                 return ((DEV_HD << MAJOR_SHIFT) | (i + MINOR_hd1a)); // logic的下标i加上hd1a
103         }
104     }
105 }

```

对设备的定义如下所示, 后面如果要添加新的设备可以在这里添加:

```
OS > miniOS-v1.3.2 > include > h fs_const.h > ...
83  #define PRIMARY_MASTER 0x0
84  #define PRIMARY_SLAVE 0x1
85  #define SECONDARY_MASTER 0x2
86  #define SECONDARY_SLAVE 0x3
87  #define MEM_MASTER 0x4
```

12. 根据读取到的分区信息，设置 superblock 的信息：

此时设备号已经到主设备，所以读的时候会进行判断，来读取到正确的分区的信息。

```
OS > miniOS-v1.3.2 > kernel > fs.c > mkfs(int)
268  struct part_info geo;
269  struct t_dev_mem meo; //add by wangshining 2021-12-25
270
271  driver_msg.type = DEV_IOCTL;
272  driver_msg.USEMEM = USE_MEM; //add by wangshining 2021-12-25
273  //driver_msg.DEVICE = MINOR(ROOT_DEV); // deleted by mingxuan 2020-10-27
274  driver_msg.DEVICE = MINOR(orange_dev); // modified by mingxuan 2020-10-27
275  driver_msg.REQUEST = DIOCTL_GET_GEO;
276  driver_msg.BUF = driver_msg.USEMEM ? &meo : &geo; //add by wangshining 2021-12-25
277  driver_msg.PROC_NR = proc2pid(p_proc_current);
278  hd_ioctl(&driver_msg);
279  disp_str(" dev size: ");
280  //added by wangshining 2021-12-29
281  if (driver_msg.USEMEM)
282  {
283      disp_int(meo.size);
284  }
285  else
286  {
287      disp_int(geo.size);
288  }
289  disp_str(" sectors\n");
```

13. 设置 inode_map 和 sect_map 以及目录节点等：

内存文件系统中并不需要加载 tty 文件和 app.tar 文件（预先加载的程序文件），所以一开始没有必要把那么多的节点设置为使用状态，inode 只给了两个：起始的 0 和目录，并只把目录的 sect-map 设为 1，写入的也只是目录的信息。

```

/*****inode map*****/
memset(fsbuf, 0, SECTOR_SIZE);
//for (i = 0; i < (NR_CONSOLES + 2); i++) //deleted by mingxuan 2019-5-22
for (i = 0; i < (NR_CONSOLES + 3); i++) //modified by mingxuan 2019-5-22
    fsbuf[0] |= 1 << i;
WR_SECT(orange_dev, 2, fsbuf); //modified by mingxuan 2020-10-27
/*****sector map*****/
memset(fsbuf, 0, SECTOR_SIZE);
int nr_sects = NR_DEFAULT_FILE_SECTS + 1;
/*
 * ~~~~~~|~ |
 * | `--- bit 0 is reserved
 * | `----- for '/'
 */
for (i = 0; i < nr_sects / 8; i++)
    fsbuf[i] = 0xFF;

for (j = 0; j < nr_sects % 8; j++)
    fsbuf[i] |= (1 << j);
WR_SECT(orange_dev, 2 + sb.nr_imap_sects, fsbuf); //modified by mingxuan 2020-10-27
/* zeromemory the rest sector-map */
memset(fsbuf, 0, SECTOR_SIZE);
for (i = 1; i < sb.nr_smap_sects; i++)
{
    WR_SECT(orange_dev, 2 + sb.nr_imap_sects + i, fsbuf); //modified by mingxuan 2020-10-27
}

```

```

memset(fsbuf, 0, SECTOR_SIZE);
struct inode *pi = (struct inode *)fsbuf;
pi->i_mode = I_DIRECTORY;

//modified by mingxuan 2019-5-21
pi->i_size = DIR_ENTRY_SIZE * 1; /* 5 files: (预定义5个文件)

/* *****
 * dir
 * ***** */
memset(fsbuf, 0, SECTOR_SIZE);
struct dir_entry *pde = (struct dir_entry *)fsbuf;
pde->inode_nr = 1;

WR_SECT(orange_dev, sb.n_1st_sect, fsbuf); //modified by mingxuan 2020-10-27

```

14. Read_super_block 根据设备号来判断，这里只有两个分支：

在 read_super_block 这个函数中，传进去 dev，之前只用了了次设备号，这里要用到主设备号，根据主设备号判断它是内存上的文件系统还是硬盘上的文件系统。由于目前只有两个分支，所以采用了一种类似于硬编码的方式，直接判断 ma_dev 是否等于 DEV_MEM 并根据判断结果直接给了 flag，这样的话其实扩展性不强，后面进行一次判断会更好一些。

superblock 缓冲区中要读哪一个文件系统也是根据主设备号来判断的。

```

OS > miniOS-v1.3.2 > kernel > C mem_fs.c > ...
969 PUBLIC void read_super_block(int dev) //modified by mingxuan 2020-10-30
970 {
971     int i;
972     MESSAGE driver_msg;
973     int ma_dev = MAJOR(dev);
974     char fsbuf[SECTOR_SIZE]; //local array, to substitute global fsbuf. added by xw, 18/12/27
975     driver_msg.type = DEV_READ;
976     driver_msg.DEVICE = MINOR(dev);
977     driver_msg.POSITION = SECTOR_SIZE * 1;
978     driver_msg.USEMEM = (ma_dev == DEV_MEM) ? USE_MEM : NOT_USE_MEM; //added by wangshining 2021-12-28
979     driver_msg.BUF = fsbuf;
980     driver_msg.CNT = SECTOR_SIZE;
981     driver_msg.PROC_NR = proc2pid(p_proc_current); ///TASK_A

    for (i = 0; i < NR_SUPER_BLOCK; i++)
        if (((super_block[i].sb_dev == ma_dev) || (MAJOR(super_block[i].sb_dev) == ma_dev))
            && super_block[i].fs_type == ORANGE_TYPE)
            break;

```

15. 文件系统中的读写扇区部分：

和上面一样，这里也是通过设备号进行分支，即进行判断 `ma_dev` 是否等于 `DEV_MEM`，由于只有两个分支，所以暂时没有考虑扩展性。

```

OS > miniOS-v1.3.2 > kernel > C mem_fs.c > ...
484 PRIVATE int rw_sector(int io_type, int dev, u64 pos, int bytes, int proc_nr, void *buf)
485 {
486     MESSAGE driver_msg;
487     int ma_dev = MAJOR(dev);
488     driver_msg.type = io_type;
489     driver_msg.DEVICE = MINOR(dev);
490     //attention
491     // driver_msg.POSITION = (unsigned long long)pos;
492     driver_msg.POSITION = pos;
493     driver_msg.CNT = bytes; /// hu is: 512
494     driver_msg.PROC_NR = proc_nr;
495     driver_msg.BUF = buf;
496     driver_msg.USEMEM = (ma_dev == DEV_MEM) ? USE_MEM : NOT_USE_MEM;
497
498     hd_rdwt(&driver_msg);
499     return 0;
500 }

```

16. 通过文件前缀名对文件系统进行划分：

文件系统并不知道是谁调用的它里面的函数，而且它的设备号在之前是类似于写死的，所以我们需要有一种方法来读到它的设备号，尤其是主设备号（代表存储介质），这里我们采用的方法是保留文件名的前缀，通过把前缀拿出来与 `vfs_table` 中的名字进行对比，找到它的 `vfs_table`，它的信息也就相应能够读出来了，之后我们就可以根据信息来判断返回的目录节点。

```
OS > miniOS-v1.3.2 > kernel > fs.c > strip_path(char *, const char *, inode **)
1081     for (i = 0; i < len; i++)
1082     {
1083         if (pathname[i] == '/')
1084         {
1085             a = i;
1086             a++;
1087             break;
1088         }
1089         else
1090         {
1091             //dev_name[i] = path[i];
1092             fs_name[i] = pathname[i]; //modified by mingxuan 2020-10-18
1093         }
1094     }
1095     fs_name[i] = '\0';
1096     // disp_str(fs_name);
1097     // disp_str("\n");
1098     if (s == 0)
1099         return -1;
1100
1101     if (*s == 'd')
1102     {
1103     }
1104     else
1105     {
1106         for (i = 0; i < NR_FS; i++)
1107         {
1108             if (!strcmp(fs_name, vfs_table[i].fs_name))
1109             {
1110                 index = i;
1111
1112                 s = s + a;
1113             }
1114         }
1115     }
```

17. 内存文件系统的 super_block:

与普通 orange 文件系统的主要区别就在于设备号，它的主设备号是 DEV_MEM，代表内存文件系统。

```
OS > miniOS-v1.3.2 > kernel > vfs.c > init_super_block_table()
111     //super_block[5] is orange/mem's superblock
112     sb->sb_dev = DEV_MEM;
113     sb->fs_type = ORANGE_TYPE;
114     sb++;
```

18. 在 vfs 中添加一项，作为内存文件系统:

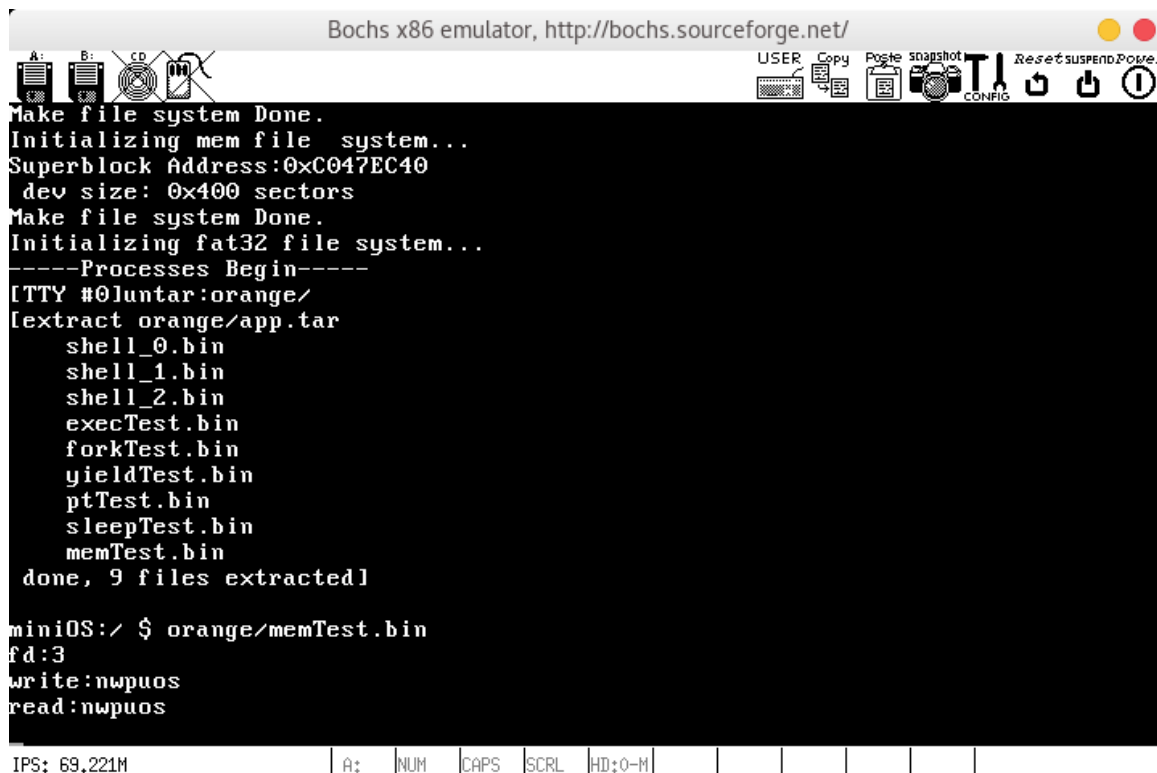
```
OS > miniOS-v1.3.2 > kernel > C vfs.c > init_vfs_table()
178 // orange
179 //device_table[4].dev_name="orange";
180 //device_table[4].op=&f_op_table[1];
181 vfs_table[5].fs_name = "orange1"; //modified by mingxuan 2020-10-18
182 vfs_table[5].op = &f_op_table[1];
183 vfs_table[5].sb = &super_block[5]; //added by mingxuan 2020-10-30
184 vfs_table[5].s_op = &sb_op_table[0]; //added by mingxuan 2020-10-30
185 }
```

五、调试运行结果：

1. 用户态测试 (memtest.c) :

因为内存文件系统并不能真正保存文件，只是在运行的时候能用，一旦重启又是空，所以首先第一步我们需要创建一个文件，open 系统调用中是 orange1，说明使用的是内存文件系统，之后返回文件描述符 fd，之后进行写入（比如我们测试时写入”nwpuos”），把 buf 里面的内容写进这个文件里面，之后关闭文件并再次打开，然后进行读操作，从中把 buf 的内容读出来，然后输出。我们可以看到正常的 nwpuos 被读出。说明内存文件系统的读写已经完成。

```
OS > miniOS-v1.3.2 > user > C memTest.c > ...
1  #include "stdio.h"
2
3  int main(int arg, char *argv[])
4  {
5      int i = 10;
6      //printf("test1:\n");
7      char buf[20] = "nwpuos";
8      int fd = open("orange1/test.txt", O_CREAT | O_RDWR);
9      printf("fd:%d\n", fd);
10     char dist[20] = "";
11     int len = write(fd, buf, 10);
12     printf("write:%s\n", buf);
13     close(fd);
14     fd = open("orange1/test.txt", O_RDWR);
15     // fd = do_vopen(str, O_CREAT | O_RDWR);
16     read(fd, dist, len);
17     dist[len] = '\0';
18     close(fd);
19     printf("read:%s\n", dist);
20
21     while (1)
22     {
23     };
24 }
```

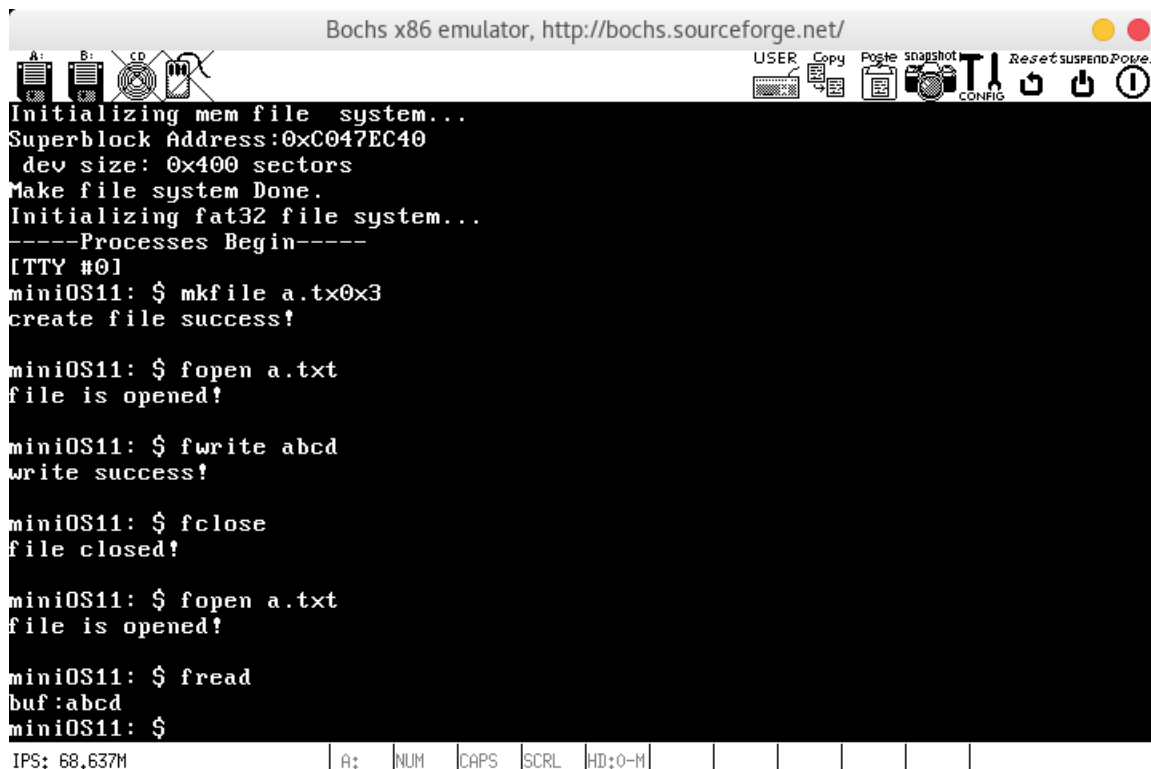
```
Bochs x86 emulator, http://bochs.sourceforge.net/
Make file system Done.
Initializing mem file system...
Superblock Address:0xC047EC40
dev size: 0x400 sectors
Make file system Done.
Initializing fat32 file system...
-----Processes Begin-----
[TTY #0]untar:orange/
[extract orange/app.tar
  shell_0.bin
  shell_1.bin
  shell_2.bin
  execTest.bin
  forkTest.bin
  yieldTest.bin
  ptTest.bin
  sleepTest.bin
  memTest.bin
done, 9 files extracted]
miniOS:/ $ orange/memTest.bin
fd:3
write:nwpuos
read:nwpuos

IPS: 69.221M | A: | NUM | CAPS | SCRL | HD:0-M | | | | | |
```

可以看到它的文件描述符值是 3，并且内容能够被正确读写，说明在用户态的文件的开闭和读写都是正常的。

2. 内核态测试 (ktest.c):

这里仿照了 ext2 的测试方法，在内核上进行测试，传进来的参数先判断第一段字符串也即具体是那种操作类型，主要有 mkfile、fopen、fwrite、fclose 等几个操作，文件名在一开始就加了前缀，这样便于测试，因为如果每次都要输入的话会比较麻烦。下图是测试时的截图，可以看到各项功能都能够正确运行，我们先创建一个名为 a.txt 的文件，之后打开它，进行写入操作，比如我们写入了 abcd，之后将文件关闭，再次打开，此时对文件进行读操作，可以看到之前写入的内容能够被成功读出。



```

Bochs x86 emulator, http://bochs.sourceforge.net/
-----
A: B: CD
USER Copy Paste snapshot CONFIG Reset suspend Power
Initializing mem file system...
Superblock Address:0xC047EC40
dev size: 0x400 sectors
Make file system Done.
Initializing fat32 file system...
-----Processes Begin-----
[TTY #0]
miniOS11: $ mkfile a.tx0x3
create file success!

miniOS11: $ fopen a.txt
file is opened!

miniOS11: $ fwrite abcd
write success!

miniOS11: $ fclose
file closed!

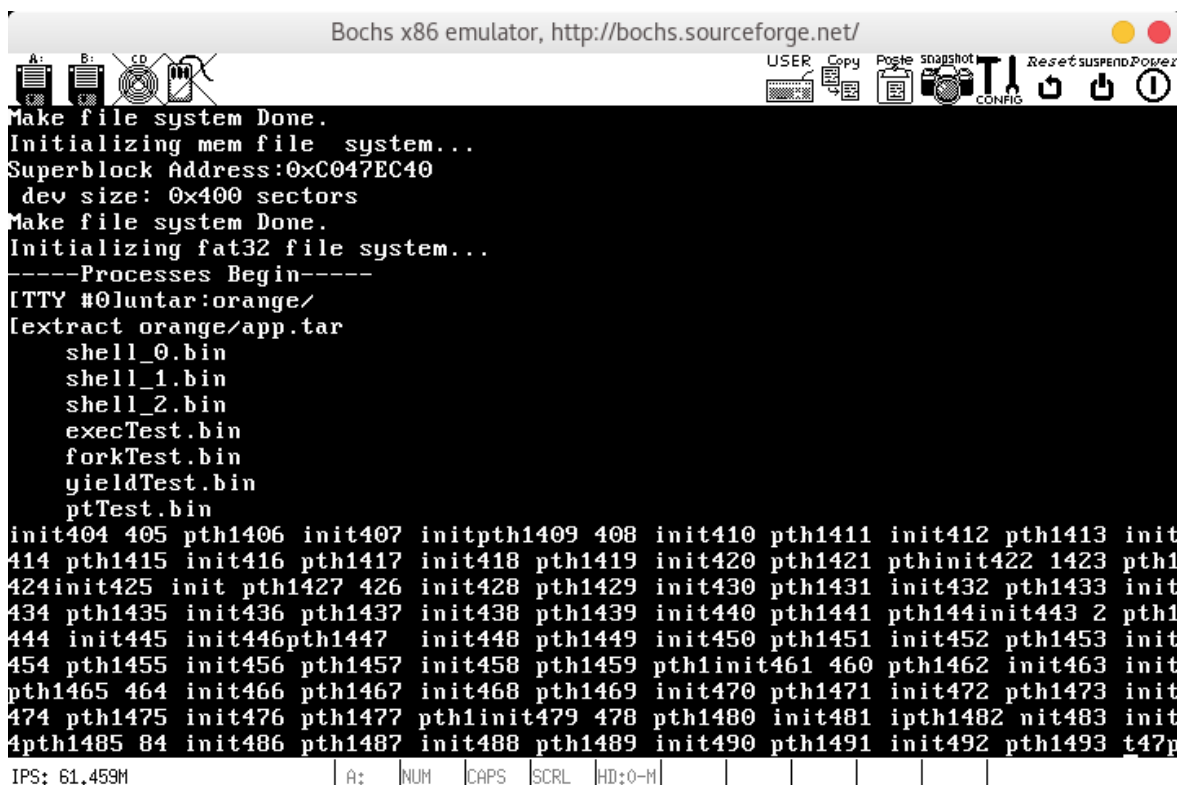
miniOS11: $ fopen a.txt
file is opened!

miniOS11: $ fread
buf:abcd
miniOS11: $
IPS: 68.637M
A: NUM CAPS SCRL HD:0-M

```

3. 证明内存文件系统能与之前存在的文件系统共存

由于我们之前的疏忽，磁盘文件系统相当于被我们删掉了，这里我们来做一个实验来验证经过修改后它们现在是可以共存的。这里我们用普通文件系统运行了 ptTest，可以观察到能够正常运行。



```

Bochs x86 emulator, http://bochs.sourceforge.net/
-----
A: B: CD
USER Copy Paste snapshot CONFIG Reset suspend Power
Make file system Done.
Initializing mem file system...
Superblock Address:0xC047EC40
dev size: 0x400 sectors
Make file system Done.
Initializing fat32 file system...
-----Processes Begin-----
[TTY #0]untar:orange/
[extract orange/app.tar
  shell_0.bin
  shell_1.bin
  shell_2.bin
  execTest.bin
  forkTest.bin
  yieldTest.bin
  ptTest.bin
init404 405 pth1406 init407 initpth1409 408 init410 pth1411 init412 pth1413 init
414 pth1415 init416 pth1417 init418 pth1419 init420 pth1421 pthinit422 1423 pth1
424init425 init pth1427 426 init428 pth1429 init430 pth1431 init432 pth1433 init
434 pth1435 init436 pth1437 init438 pth1439 init440 pth1441 pth144init443 2 pth1
444 init445 init446pth1447 init448 pth1449 init450 pth1451 init452 pth1453 init
454 pth1455 init456 pth1457 init458 pth1459 pth1init461 460 pth1462 init463 init
pth1465 464 init466 pth1467 init468 pth1469 init470 pth1471 init472 pth1473 init
474 pth1475 init476 pth1477 pth1init479 478 pth1480 init481 ipth1482 nit483 init
4pth1485 84 init486 pth1487 init488 pth1489 init490 pth1491 init492 pth1493 t47p
IPS: 61.453M
A: NUM CAPS SCRL HD:0-M

```

六、实验基础知识总结

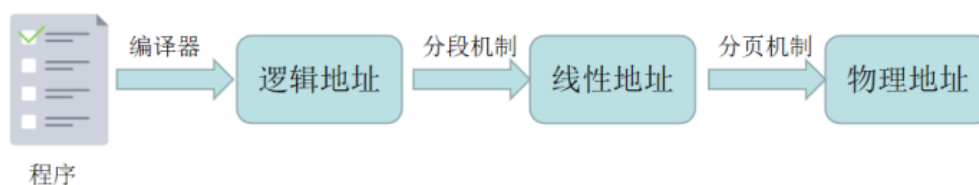
1. 熟练掌握一般文件系统的运行全过程，对其中的细节也要充分了解，比如读 superblock 到缓冲区的过程等，前面已经详细解释，这里不再赘述。

2. 熟练掌握内存中物理地址、线性地址、逻辑地址的关系：

逻辑地址是指由程序产生的与段相关的偏移地址部分。例如，你在进行 C 语言指针编程中，可以读取指针变量本身值(&操作)，实际上这个值就是逻辑地址，它是相对于你当前进程数据段的地址，不和绝对物理地址相干。只有在实模式下，逻辑地址才和物理地址相等（因为实模式没有分段或分页机制,Cpu 不进行自动地址转换）；逻辑也就是在保护模式下程序执行代码段限长内的偏移地址（假定代码段、数据段如果完全一样）。

线性地址是逻辑地址到物理地址变换之间的中间层。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。

物理地址是指出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。



七、所遇问题及解决方法

1. 读写内存时内存的实际值是正常的，但是无法正确的读出，总是全为 1。

解决：全部转化为线性地址。由于初始化的时候使用的是物理地址，而 memcpy 并不知道传过来的到底是线性地址还是物理地址，当为线性地址的时候，如果使用的是物理地址那么就显然不能读出正确的值。

2. 如何进行地址解算，如何确定文件起始扇区。

通过分析代码，最终我们明白，读写过程中的地址是分区的相对地址，并以此为依据对代码进行了修改，最终成功实现。

八、实验总结：

本次实验我们两人通过半个多月的努力，最终在 minios 上实现了内存文件系统，在此过程中，我们自己资料搜集，并进行资料阅读，深入研究当前 minios 中的相关实现并思考如何做修改，期间设计测试用例，编写测试代码，进行测试，根据测试结果不断的修改完善代码。这个过程不仅深化了我对操作系统的知识的了解，同时也提升了我的实践能力，提高了我应用所学知识到实践中去的能力，同时通过和队友的精诚协作，分工配合，最终将试点班任务完成，整个过程收获颇丰。

最大的后悔的地方是没有及早与老师讨论，其实我们与晓峰学长讨论的蛮多，但是说到底老师才是对整个项目进行宏观把控的人，希望自己能够在今后的学习生活中吸取教训，能够更积极地与老师讨论。最后，感谢谷老师和晓峰学长的辛勤付出，感谢老师给了我这次参加试点班的机会。