林子杰_20373980_lab5

一、思考题

Thinking 5.1

正如大部分帖子所介绍，`/proc`系统是一个伪文件系统。这个文件系统的"组织管理"的数据并不在我们常认为的外部设备中，而是内存中的数据；而这些数据大部分都是内核数据。这一文件系统在Linux系统上有加载，可以通过命令行访问其"目录"下的信息（一般是只读，也有一些可写信息）。查询了一些资料，在支持这一文件系统的虚拟环境中，一般需要先挂载，再使用。随后，我们可以通过`ls`、`cat`等命令操作其文件树。

Windows中如何实现这一文件系统的功能？查询了一些相关资料，感觉信息少之又少。Linux下，`/proc`可以通过访问"文件"的形式查看系统资源，例如进程信息、网卡信息、内存信息等；于是发现Windows下也有相似的一组命令`typeperf`，可以用于监听系统信息。

该虚拟文件系统将许多内核资源抽象成便于用户访问的文件信息，便于用户使用；缺点当然是其在挂载后留在内存中占用内存空间，并且占用系统资源。

Thinking 5.2

根据之前学习的知识，kuseg0段的存取是通过高速缓存Cache来进行的，而这又是内核地址空间。倘若我们令外设从这一段读取数据，可能会使内核加载到错误的数据，这是很危险的。

Thinking 5.3

事实上，MOS操作系统中的文件控制块和Linux中的inode有一些相似之处；他们之间的不同，本质上是文件控制块和索引节点的不同。首先要明确，通过文件控制块来组织文件系统的结构，是顺序组织的，需要一一片连续的空间来存放FCB。我们在上面的实验中，如果仔细观察，便会发现，每个文件控制块的大小达到了256KB。如果我们按照路径查找文件，从磁盘中将一个控制块调入内存，取得指针，再调文件控制块进入内存，这样产生的性能浪费可想而知；但从始至终，我们也仅仅是使用了文件控制块的文件名。为了减少性能的浪费，部分操作系统选择将文件名和文件描述信息分开的做法，这其中存放文件描述信息的数据结构就被称为inode。inode也是存在于磁盘块中的可以描述文件信息的数据结构，但是它存放的信息和FCB有区别，并且它不包含文件名这一关键信息。

Thinking 5.4

在`include/fs.h`这一文件中，有一个宏定义`BY2FILE`，它的大小为256字节，每一个磁盘块的大小为4096字节，计算可得每个磁盘块最多能存储16个控制块。

在一个文件控制块中，我们有十个直接指向磁盘块的指针，还有一个间接指针，指向一个存放了磁盘块指针的磁盘块，大小为4096字节，共可以存放1024个块指针（间接指针的前10个不计）。故而一个目录下最多能有1024*16即16KB个文件（此处将FTYPE_DIR结构体看作目录）。

对于一个FTYPE_REG的目录项，有1024个块指针，每个块指针指向4KB大小的磁盘块，故而支持的文件最大为4MB。

Thinking 5.5

根据`fs.h`中的宏定义的注释可知，可映射的最大磁盘空间为1GB

Thinking 5.6

显然不能，文件系统作为用户进程，倘若允许其能够访问内核地址空间，后果不堪设想。

Thinking 5.7

名称	含义	大小
BY2SECT	一张扇区的字节数	512
SECT2BLK	一个磁盘块的扇区数	8
DISKMAX	磁盘可映射的最大虚拟地址	0x40000000
BY2BLK	一个磁盘块的字节数	4096
BIT2BLK	一个磁盘块的比特数	32768
NDIRECT	File结构体的直接块指针数	10
NINDIRECT	File结构体间接指针数	1024
BY2FILE	一个FILE结构体的字节数	256
MAXFILESIZE	一个文件的最大字节数	1024 * 4096

Thinking 5.8

这种转换之所以可行，应当和结构体内部属性排列顺序有关。

```
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};

struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

`Filefd`结构体的第一个成员便是`Fd`类型，且结构体指针本身就指向结构体的首地址，这种转换因此可行。代码中还有另一处类似的例子。

```
struct Fsreq_open *req;
req = (struct Fsreq_open *)fsipcbuf;

struct Fsreq_open {
    char req_path[MAXPATHLEN];
    u_int req_omode;
};
```

其中，`fsipcbuf`是一个`u_char`类型的数组，`req`对应结构体的第一个成员也是一个`cahr`数组。不过`fsipcbuf[]`和`req_path[]`不等长。

Thinking 5.9

在`fs`目录下，新建一个`helloworld`文件，并通过修改Makefile将其挂载在磁盘镜像中。同时编写测试程序，在`fork()`之后得到文件描述符并打印输出，可以发现父子进程仍旧共享文件描述符。

```
void umain() {
    int r;
    int fdnum;
    if ((r = open("/newmotd", O_RDWR)) < 0) {
        user_panic("open /newmotd: %d", r);
    }
    writef("envid : %d ,fdnum : %d\n", syscall_getenvid(), r);
    fdnum = r;
    char *str = "Hello World!";
    if ((r = write(fdnum, str, strlen(str))) < 0) {
        user_panic("write /newmotd: %d", r);
    }
    fork();
    if ((r = open("/helloworld", O_RDWR)) < 0) {
        user_panic("open /helloworld: %d", r);
    }
    fdnum = r;
    writef("%s\n", str);
    if ((r = write(fdnum, str, strlen(str))) < 0) {
        user_panic("write /helloworld: %d", r);
    }
    writef("envid : %d ,fdnum : %d\n", syscall_getenvid(), r);
}
```

随后在本地的编译器中编写C语言文件：

```c
#include <unistd.h>
#include "stdio.h"

int main() {
    FILE *f;
    f = fopen("in.txt", "r");
    fseek(f, 0, 0);
    fork();
    printf("env_id : %d ,f : %d\n", getppid(), f);
}
```

发现父子进程共享文件定位指针。

Thinking 5.10

```c
/* Fd结构体表示文件描述符，供用户使用，是单纯的内存数据 */
struct Fd {
    u_int fd_dev_id;        //该文件所处的设备编号(id)
    u_int fd_offset;        //该文件在设备中的偏移
    u_int fd_omode;         //该文件的打开模式
};

/* Filefd结构体表示文件描述符和文件，供用户使用，是单纯的内存数据 */
struct Filefd {
    struct Fd f_fd;         //某个文件的文件描述符
    u_int f_fileid;         //打开该文件的文件id
    struct File f_file;     //该文件的文件控制块
};

/* 仅定义在serv.c中的结构体，仅由文件系统使用，用于保存已打开的文件，是单纯的内存数据 */
struct Open {
    struct File *o_file;    //所打开文件的文件控制块的指针
    u_int o_fileid;         //所打开文件的文件id
    int o_mode;             //所打开文件的打开方式
    struct Filefd *o_ff;    //该文件描述符在文件进程中的虚拟地址
};
```

这三个结构体都是在内存中的数据，围绕文件系统和用户进程定义，用于协助文件系统的实现；同时他们也是硬件中的文件在软件中的抽象映像。

Thinking 5.11

图中有三类箭头，代表UML类图中的三类消息：

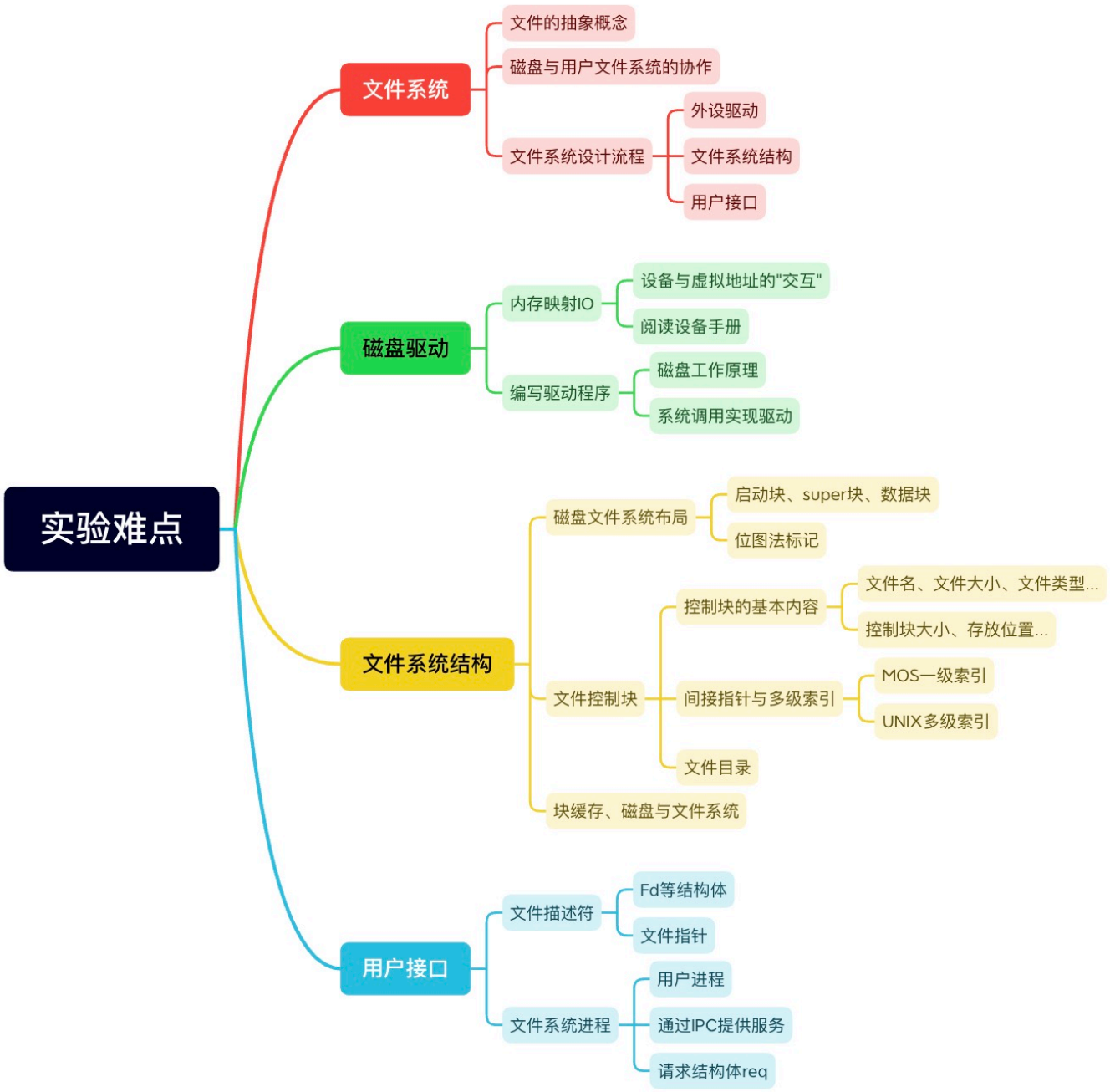
- 黑色三角+实线+实心黑点组成的箭头。这类消息可能没有发送者或接受者。
- 黑色三角+实线组成的箭头。表示同步消息，发送者发送消息后，暂停活动等待接受者响应。
- 开三角+虚线组成的箭头。表示返回消息，和同步消息一并使用。

在前面的实验中，我们已经实现了IPC机制，不同进程之间可以通过IPC机制进行通信。我们的文件系统对应的进程控制块为`envs[1]`，可以通过这个控制块的信息从而使用IPC机制在用户进程和文件系统进程之间实现进程通信。

Thinking 5.12

这一循环，每次都会调用`ipc_recv()`函数，以检测其他进程是否向文件系统进程发送数据，倘若有，则响应；倘若没有，则阻塞。这一循环不会使CPU陷入忙等，且文件系统进程为用户进程，不会损害到内核；MOS系统结束运行时，该进程会被`kill`而不会长期存在。

二、实验难点



三、体会与感想

首先表示，lab5的东西真的好多好多，比前面每一个lab的内容都要多。东西主要多在如下几个方面：

- 代码量增加：新增了整整一个文件夹的代码，`user`目录下也出现了很多需要阅读的新文件
- 代码高度协作：结构体、函数穿插联系更加紧密，需要不断夸文件阅读
- 知识跨度广泛：从硬件到软件、从进程通信到内存映射文件系统，处处涉猎

尽管单个知识点的难度相对lab4较小，但是需要阅读的东西实在太多，很难在短时间内吃的比较透彻。对于一些边角知识还是感到模棱两可。

不过本次lab也深化了我对IPC机制、软硬件关系、文件系统服务等知识的理解，尤其是后面两者，以前在学习理论的时候对这方面的知识总是感到模糊不清，在完成这一试验后，感觉能够初步建立软硬件结合的框架了。

同以往一样，将本次实验的笔记附在下面。

Lab5实验笔记

一、实验目的

计算机不可能将所有的数据都存放在内存中，很多静态的数据是需要存放在磁盘、光盘等外部设备中。当然，这些外部设备组织数据的方法比较"冰冷"。大部分都是通过区域、块来划分数据。这虽然方便了操作系统管理数据，但是对用户来说并不友好。用户更加期望使用自己喜欢的名字来记录一份有特定功能的数据。我们在这个Lab中，就要组织一个磁盘镜像，并且在操作系统中实现用户友好的文件系统。

二、文件系统概述

1.基本认识

前面提到，外部设备上的数据的组织、排布形式往往是便于操作系统管理而不方便用户调用的，毕竟用户是人，人相比计算机，长于抽象而短语计算。故而，优秀抽象模式，便是文件系统的追求。我们今天所用到的图形界面文件系统，是经过层层迭代的工业品。实际上，最开始的文件系统，甚至没有目录，没有格式的区分，只有一层抽象形式。这可能是难以想象的，但这的确是文件系统起步阶段时的真实情况。科学家们经过不断探索，得到了我们今天看到的树状组织、层层下降的文件系统。我们在本次实验中要实现的文件系统便是这么一个文件系统。

Thinking 5.1

正如大部分帖子所介绍，`/proc`系统是一个伪文件系统。这个文件系统的"组织管理"的数据并不在我们常认为的外部设备中，而是内存中的数据；而这些数据大部分都是内核数据。这一文件系统在Linux系统上有加载，可以通过命令行访问其"目录"下的信息（一般是只读，也有一些可写信息）。查询了一些资料，在支持这一文件系统的虚拟环境中，一般需要先挂载，再使用。随后，我们可以通过`ls`、`cat`等命令操作其文件树。

Windows中如何实现这一文件系统的功能？查询了一些相关资料，感觉信息少之又少。Linux下，`/proc`可以通过访问"文件"的形式查看系统资源，例如进程信息、网卡信息、内存信息等；于是发现Windows下也有相似的一组命令`typeperf`，可以用于监听系统信息。

该虚拟文件系统将许多内核资源抽象成便于用户访问的文件信息，便于用户使用；缺点当然是其在挂载后留在内存中占用内存空间，并且占用系统资源。

2.磁盘文件系统

磁盘文件系统是一种利用数据存储设备保存计算机文件的文件系统。最常用的磁盘文件系统是磁盘驱动器（一种可以读写磁盘的硬件）。

3.用户空间文件系统

在操作系统中实现的文件系统属于软件文件系统。在Linux操作系统中，文件系统是内核的一部分，这是宏内核操作系统的特点；对应地，在微内核操作系统中，文件系统在用户空间实现，通过对外接口为用户进程提供服务。

4.文件系统的设计与实现

我们的文件系统主要需要实现以下几个部分：

- **外部存储设备驱动。**硬件厂商设计的外部设备的运行方式不尽相同，但是我们不可能让主流操作系统去适应五花八门的外部设备，否则用户的体验会非常糟糕。故而我们要求外部设备提供通用、明确的接口。我们的IDE磁盘也是如此。
- **文件系统结构。**在本部分，我们将实现磁盘和操作系统中的文件结构。具体一点，是一个支持多级目录的操作系统。
- **文件系统的用户接口。**我们将实现文件系统的用户接口，方便用户调用。

实际上，我们需要实现的文件系统也是一个运行在操作系统中的进程。我们将通过 `fs/fsformat.c` 来创建磁盘镜像，在 `fs/fs.c` 中实现文件系统的基本功能函数，文件系统进程通过 `fs/ide.c` 与磁盘镜像进行交互，这一进程通过 `fs_serv.c` 来运行。该进程通过IPC通信与用户进程 `user/fsipc.c` 内的通信函数进行交互；用户进程在 `user/file.c` 中实现用户接口，并在 `user/fd.c` 中引入文件描述符，抽象地操作文件、管道等内容。

二、IDE磁盘驱动

1.介绍

在理论课上，我们接触到操作系统和某些外部设备的中间级：驱动。事实上，对于一些非常低层的外部设备，操作系统的开发者也不希望自己去设计它们的具体操作方法——这些工作一般都交给硬件服务商来做，毕竟操作系统本身已经足够复杂了。操作系统的开发者们更希望硬件开发者能够按照他们给出的API开发一些能够操作硬盘的最基础的代码。这些代码就是我们经常提到的驱动。

MOS操作系统的磁盘也需要驱动，其中的部分任务已经完成，我们还需要完成剩下的一部分任务。

2.内存映射I/O

内存映射I/O，即MMIO，是现在应用最广泛的一种IO方式。IO设备一般通过自己的寄存器拿取外界数据，而在MMIO中，这些寄存器被映射到了一些内存地址物理中，这些内存地址又被映射到了虚拟地址上；驱动程序在其间起到了传输数据的桥梁的作用。这样做的优点便是优化了用户的使用体验，用户可以在编码的时候操作这些外设的地址空间。

MOS操作系统采取了MMIO的模式，且在我们的MOS中，一些内核地址空间可以通过硬件实现物理地址和虚拟地址的转换（kseg0段和kseg1段），这又简化了我们的工作：我们只需要将外部设备的物理地址映射到内核中固定的虚拟地址上，驱动程序就可以直接通过读写这些地址来操作外部设备。

在MOS中，console设备被映射到`0x10000000`，simulated IDE disk被映射到`0x13000000`，这些都是低于内存空间上限的地址，也是这些设备在内存中被读写的地址。鉴于上面提到的kseg0段和kseg1段的一些优势，我们决定，在编写设备驱动的时候，将这些外设的物理地址，转化为存在于kseg1段的虚拟地址。转换很简单，**加上偏移0xA0000000**即可。举个例子，对于console设备，当我们向虚拟地址`0x10000000+0xA0000000`写入一个字符的时候，外部设备就能通过驱动程序"感受"到这次写入，从而在在可视化界面上打印出这个字符。

当然，还记得IO设备等一系列外设的特点吗？一般外部设备和操作系统进行交流的时候，都会采取中断的方式，此处读写内核虚拟地址的操作也不例外。为了提供读写外设的驱动，我们需要实现两个系统调用即`sys_write_dev()`和`sys_read_dev()`。

```
/* 写外设的系统调用 */
int sys_write_dev(int sysno, u_int va, u_int dev, u_int len)
{
    /* 在MOS实验中，我们有三个可写的外部设备，故而我们需要判断需要读写的外设是哪一个外设
     * 由于bcopy()函数拷贝时左闭右开的特性，我们在判断拷贝范围是否有效时，需要判断两端的
     * 开区间 */
    int flag = 0 ,dst = dev + 0xA0000000;
    if(va >= ULIM) return -E_INVAL;
    if(flag == 0 && dev >= 0x10000000 && (dev + len) <= 0x10000020) flag = 1;
    if(flag == 0 && dev >= 0x13000000 && (dev + len) <= 0x13004200) flag = 1;
    if(flag == 0 && dev >= 0x15000000 && (dev + len) <= 0x15000200) flag = 1;
    if(flag == 0) return -E_INVAL;
    /* 使用bcopy()，将va处长度为len字节的内容拷贝到dst这一虚拟地址 */
    bcopy((void *)va ,(void *)dst ,len);
    //printf("va %d\n",va);
    return 0;s
}
/* 读外设的系统调用 */
int sys_read_dev(int sysno, u_int va, u_int dev, u_int len)
{
    /* 读写外设的系统调用非常相似，不再做介绍了 */
    int flag = 0 ,dst = dev + 0xA0000000;
    if(va >= ULIM) return -E_INVAL;
    //printf("dev : %x len : %x\n",dev,len);
    if(flag == 0 && dev >= 0x10000000 && (dev + len) <= 0x10000020) flag = 1;
    if(flag == 0 && dev >= 0x13000000 && (dev + len) <= 0x13004200) flag = 1;
    if(flag == 0 && dev >= 0x15000000 && (dev + len) <= 0x15000200) flag = 1;
    if(flag == 0) return -E_INVAL;
    bcopy((void *)dst ,(void *)va ,len);
    return 0;
}
```

这两个函数并不难填写，我们在此处主要是要理解好读写外设的原理及方法。

Thinking 5.2

根据之前学习的知识，kuseg0段的存取是通过高速缓存Cache来进行的，而这又是内核地址空间。倘若我们令外设从这一段读取数据，可能会使内核加载到错误的数据，这是很危险的。

3.IDE磁盘

在我们的MOS操作系统中，我们使用仿真器模拟了一个虚拟磁盘，我们将操纵这个磁盘。当然，在操作磁盘前，我们有必要了解一些磁盘相关的知识。

- 扇区(sector)：磁盘盘片被划分成很多扇形的区域，叫做扇区。扇区是磁盘执行读写操作的单位，一般是512字节。扇区的大小是一个磁盘的硬件属性。
- 磁道(track): 盘片上以盘片中心为圆心，不同半径的同心圆。
- 柱面(cylinder)：硬盘中，不同盘片相同半径的磁道所组成的圆柱。
- 磁头(head)：每个磁盘有两个面，每个面都有一个磁头。当对磁盘进行读写操作时，磁头在盘片上快速移动。

扇区是读写磁盘的基本单位。我们前面提到过，由于采取了MMIO的IO方法，我们读写磁盘的时候可以通过直接读写地址空间实现。在我们的仿真器中，规定好了一系列的地址空间的映射，我们需要做的是按照规定将数据读写在指定位置， 这样磁盘就能通过驱动程序"感知"到我们对它发出的请求以及数据。在我们的MOS操作系统中，磁盘在内存中映射到的起始地址是`0x13000000`，其结束地址为`0x13004200`，在这个范围中，Gxemul仿真器对一些偏移做出了下列规定。

偏移(Offset)	写入效果(Effect)
0x0000	设置从磁盘镜像开始的偏移量，用于下一次操作
0x0008	以字节为单位设置高32位偏移
0x0010	选择下一次读写使用的磁盘ID
0x0020	设置读写操作，0为写，1为读
0x0030	得到上一次操作的状态，0表示失败，1表示成功
0x4000-0x41ff	读写数据缓冲区，大小为512字节

向这些偏移写入相关信息，磁盘就能根据这些信息来识别操作、存取数据。

4.驱动程序编写

我们已经知道如何驱动磁盘按照我们的要求读写数据，下面我们要做的就是编写系统调用，为文件系统进程提供能够读写内核空间的系统调用。，为了实现系统调用，在`ide.c`中，我们需要完成两个函数`ide_read()`和`ide_write()`，来完成这一项工作。

```
/* diskno是磁盘号，secno是开始读取的的扇区号，dst是写入虚拟地址空间的目标地址
 * nesecs是需要读取的扇区数，我们需要读取的数据长度，按照字节进行计算，则是512*
 * nesecs */
void ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
{
  int offset_begin = secno * 0x200;               //读取开始的位置，通过扇区数计算
```

```
  int offset_end = offset_begin + nsecs * 0x200;  //读取结束的位置
  int offset = 0;   //偏移，用于判断循环结束条件
  int dst_addr;     //目标
  int data_sheet = 0;
  int return_value = 0;

  while (offset_begin + offset < offset_end) {
    /* dst_addr为读取磁盘的目标地址，等与offset_begin与offset之和 */
    dst_addr = offset_begin + offset;
    /* 向0x13000010写入diskno磁盘号，告知磁盘存取的磁盘号 */
    if(syscall_write_dev(&diskno, 0x13000010, 4) < 0) {
      user_panic("Failed to save diskno!\n");
    }
    /* 将dst_addr写入0x13000000，告知磁盘读取位置的地址距离磁盘首地址的偏移 */
    if(syscall_write_dev(&dst_addr, 0x13000000, 4) < 0) {
      user_panic("Failed to save dst_addr!\n");
    }
    /* 向0x13000020写入0，告知磁盘本次操作为读操作 */
    if(syscall_write_dev(&data_sheet, 0x13000020, 4) < 0) {
      user_panic("Failed to save data_sheet!\n");
    }
    /* 取回磁盘操作的返回值 */
    if(syscall_read_dev(&return_value, 0x13000030, 4) < 0) {
      user_panic("Failed to read return_value!\n");
    }
    /* 如果返回值为0，表示操作失败，panic */
    if(return_value == 0) {
      user_panic("Failed to read!\n");
    }
    /* 磁盘已经将数据写入0x13004000的512字节大小的缓冲区，现将其读至目标地址dst
     * 偏移offset的位置。受限于缓冲区的大小，我们每次只能读取512字节的数据 */
    if(syscall_read_dev(dst + offset, 0x13004000, 512) < 0) {
      user_panic("Failed to read data!\n");
    }
    /* 偏移增加512，直至达到offset_end */
    offset += 0x200;
  }
}

void ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)
{
  int offset_begin = secno * 0x200;
  int offset_end = offset_begin + nsecs * 0x200;
  int offset = 0;
  int dst_addr;
  int data_sheet = 1;
  int return_value = 0;
  // DO NOT DELETE WRITEF !!!
  writef("diskno: %d\n", diskno);
```

```
  while (offset_begin + offset < offset_end) {
    dst_addr = offset_begin + offset;
    /* 向0x13000010写入diskno，告知磁盘即将读取的磁盘号 */
    if(syscall_write_dev(&diskno, 0x13000010, 4) < 0) {
      user_panic("Failed to save diskno!\n");
    }
    /* 向0x13000000写入dst_addr，告知磁盘写入位置相对于磁盘首地址的偏移 */
    if(syscall_write_dev(&dst_addr, 0x13000000, 4) < 0) {
      user_panic("Failed to save dst_addr!\n");
    }
    /* 向缓冲0x13004000区写入src+offset处开始的512字节的数据 */
    if(syscall_write_dev(src + offset, 0x13004000, 512) < 0) {
      user_panic("Failed to save data!\n");
    }
    /* 向0x13000020处写入1，告知磁盘下一次操作为写操作 */
    if(syscall_write_dev(&data_sheet, 0x13000020, 4) < 0) {
      user_panic("Failed to save data_sheet!\n");
    }
    /* 磁盘完成操作，此时从磁盘中取出返回值，判断操作是否成功 */
    if(syscall_read_dev(&return_value ,0x13000030, 4) < 0) {
      user_panic("Failed to read data!\n");
    }
    /* 操作失败，则panic */
    if(return_value == 0) {
      user_panic("Failed to write!\n");
    }
    /* 偏移增加 */
    offset += 0x200;
  }
}
```

理解驱动程序编写这一部分并不困难，重要的是我们需要将这一部分的知识和以前学习过的中断异常、系统调用结合起来（在exam和本次的exercise中，都需要我们自己实现中断函数的接口）。

三、文件系统结构

1.初步认识

在前面，我们已经实现了磁盘的驱动，我们可以使用驱动实现操作系统和磁盘的交互了。接下来，我们将组织磁盘和操作系统中的文件系统结构

磁盘系统文件布局：

" alt="6631653918859_.pic" style="zoom:50%;" />

如图中所示，我们的磁盘被分成了一个一个的磁盘块，即**block**。磁盘块是相对于磁盘而言的一个虚拟概念，也是操作系统与磁盘交互的最小单位（通常来说，直接操作扇区即**sector**，会因为数量过多而在寻址时产生麻烦）。

在MOS操作系统中，每一个磁盘块的大小是4096字节。我们在图中可以看到，第0、1、2个磁盘块被赋予了如下的特殊含义：

- Block0：用于存放启动信息和分区表信息

- Block1：超级块，存放描述文件的基本信息，包括魔数、磁盘大小、根目录位置。在我们的MOS中，超级块的结构如下：

```
struct Super {
    u_int s_magic;       //魔数，用于识别文件系统，为常量
    u_int s_nblocks;     //磁盘块数量，本文件系统为1024
    struct File s_root; //根目录块，其f_type为FTYPE_DIR，f_name为”/”
};
```

这里面一个比较有意思的部分是`s_root`的定义，我们发现其中有一些域已经被明确了，这一部分的工作是在`fsformat.c`的`init_disk()`中完成的：

```
/* FTYPE_DIR为1，表示目录；另有一个FTYPE_REG，为0，表示常规文件 */
super.s_root.f_type = FTYPE_DIR;
/* f_name为"/" */
strcpy(super.s_root.f_name, "/");
```

- Block2：位图块，用于存放位图。位图是什么？在之前的实验中，我们对一些资源（进程块、内存页）等采用链表进行管理；另外一种便于管理资源的方法就是位图法，我们将用位图法管理磁盘资源，用一个bit表示每个磁盘块的使用情况——0表示占有，1表示空闲。

```
/* 在我们的MOS中，NBLOCK = 1024，BIT2BLK = 4096 * 8，nbitblock的含义是"存放位图需要使用的磁盘块的块数"。很显然，在我的MOS中，nbitblock为1，因为我们只需要1024个bit就足矣管理所有的磁盘块，这甚至都用不到block2空间的四分之一。计算的方法之所以这么复杂，是为了向上取整 */
nbitblock = (NBLOCK + BIT2BLK - 1) / BIT2BLK;
for(i = 0; i < nbitblock; ++i) {
    /* 设置存放位图块的类型为BLOCK_MAP，其值为4。此处循环执行一次结束 */
    disk[2+i].type = BLOCK_BMAP;
}
for(i = 0; i < nbitblock; ++i) {
    /* memset(void *s, int ch, size_t n)函数，将s中当前位置后面的n个字节用ch替换，这里首先将disk[2](循环只执行一次)的data域全部bit设为1。disk，就是我们模拟的磁盘块的首地址，BY2BLK的大小为4096，我们首先将这个磁盘块的所有位置1 */
    memset(disk[2+i].data, 0xff, BY2BLK);
}
if(NBLOCK != nbitblock * BIT2BLK) {
    /* 当然，倘若位图无法用满整个磁盘块，我们需要将多余的位置为0，避免错误的访问。diff是我们计算出的位图需要使用的byte的数量，利用这个数，我们可以将其它无用的位置为0 */
    diff = NBLOCK % BIT2BLK / 8;
    memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
}
```

理解完位图法后，我们需要完成`exercise5.3`的一个`free_block()`函数。这个函数的功能为释放一个位图块，即将其在位图中对应的bit置为1。其实现如下：

```
void free_block(u_int blockno)
{
    /* 首先，我们需要判断blockno是否有效。根据我们在上面的理解，我们知道blockno的大小不能超过1024，这个值使用super->s_nblocks取得。但是，blockno为0的时候，操作也是非法的，这是为什么？这是根据块的功能决定的，block0对应的块用于存放关键启动信息，倘若这个块不能被访问，那么我们就无法使用文件系统，所以我们不允许对这个块进行操作 */
    if(blockno == 0 || super == 0 || blockno >= super->s_nblocks) return;
    /* 用blockno整除32，可以得到该blockno对应的字，然后通过左移1，来修改位图 */
    bitmap[blockno / 32] |= (1 << (blockno % 32));
}
```

除了这个需要我们自己实现的函数外，我们还可以再看一个例子。

```
int block_is_free(u_int blockno)
{
    /* 首先判断超级块是否有效 */
    if (super == 0 || blockno >= super->s_nblocks) {
        return 0;
    }
    /* 计算blockno对应的字，取出并且判断 */
    if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
        return 1;
    }
    return 0;
}
```

磁盘文件系统结构的介绍大致如此。

文件系统详细结构

对于操作系统来说，想要管理一类资源，就需要有相应的数据结构。在MOS中，我们使用文件控制块来管理文件资源。

```
struct File {
    u_char f_name[MAXNAMELEN];  //文件名
    u_int f_size;               //文件大小
    u_int f_type;               //文件类型，包括FTYPE_REG和FTYPE_DIR
    u_int f_direct[NDIRECT];    //直接文件指针，共有十个，每个指向一个4KB大小的文件块
    u_int f_indirect;           //间接文件指针，指向一个磁盘块，其中存储了文件指针
    struct File *f_dir;         //指向文件所属的文件目录
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

在这个结构体中，一些比较好描述的区域已经在代码块的注释中进行了标注。当然，接下来还有一些信息需要解释。首先明确一件事，就是在我们的MOS操作系统中，为了进行简化，我们并不适用复杂的三级索引（参考理论课），仅仅使用一级索引，并且在实验中，我们不使用间接文件指针块的前十个指针。另外，数组`f_pad`用于填充文件控制块，使得每个磁盘块可以放下整数个`File`结构体。

前面提到，我们的文件控制块分为`FTYPE_DIR`和`FTYPE_REG`，前者表示目录文件，后者表示常规(Regular)文件。对于普通的文件，其指向的一系列磁盘块存储着文件的内容；而对于目录文件来说，其指向的一系列磁盘块则存放了一系列的磁盘控制块。当我们需要查找某个文件的时候，我们会从超级块中读取根目录，然后循着给出的文件路径逐一查找。

我们通过`fsformat`程序来创建磁盘镜像`gxemul/fs.img`，这个程序是由`fs/fsformat.c`编译而来的。在创建磁盘镜像的时候，我们使用的是Linux的gcc编译器，因此编译生生成的`fsformat`独立于MOS之外。

此处，指导书提醒我们"请阅读 fs/fsformat.c 和 fs/Makefile，掌握如何将文件和文件夹按照文件系统的格式写入磁盘，了解文件系统结构的具体细节，学会添加自定义文件进入磁盘镜像。"。这一部分的要求或许会稍后补充。

Thinking 5.3

事实上，MOS操作系统中的文件控制块和Linux中的inode有一些相似之处；他们之间的不同，本质上是文件控制块和索引节点的不同。首先要明确，通过文件控制块来组织文件系统的结构，是顺序组织的，需要一一片连续的空间来存放FCB。我们在上面的实验中，如果仔细观察，便会发现，每个文件控制块的大小达到了256KB。如果我们按照路径查找文件，从磁盘中将一个控制块调入内存，取得指针，再调文件控制块进入内存，这样产生的性能浪费可想而知；但从始至终，我们也仅仅是使用了文件控制块的文件名。为了减少性能的浪费，部分操作系统选择将文件名和文件描述信息分开的做法，这其中存放文件描述信息的数据结构就被称为inode。inode也是存在于磁盘块中的可以描述文件信息的数据结构，但是它存放的信息和FCB有区别，并且它不包含文件名这一关键信息。

Thinking 5.4

在`include/fs.h`这一文件中，有一个宏定义`BY2FILE`，它的大小为256字节，每一个磁盘块的大小为4096字节，计算可得每个磁盘块最多能存储16个控制块。

在一个文件控制块中，我们有十个直接指向磁盘块的指针，还有一个间接指针，指向一个存放了磁盘块指针的磁盘块，大小为4096字节，共可以存放1024个块指针（间接指针的前10个不计）。故而一个目录下最多能有1024*16即16KB个文件（此处将FTYPE_DIR结构体看作目录）。

对于一个FTYPE_REG的目录项，有1024个块指针，每个块指针指向4KB大小的磁盘块，故而支持的文件最大为4MB。

在`exercise 5.4`中，我们需要完成一个`create_file()`函数，用于创建文件并返回其指针。这里涉及了比较多层的函数，我们来逐一剖析。

```
/* 在给定目录下，返回一个新的File结构体指针 */
struct File *create_file(struct File *dirf) {
    struct File *dirblk;
    int i, bno, found;
    /* dirf->f_size / BY2BLK，计算现在dirf使用了多少文件指针 */
    int nblk = dirf->f_size / BY2BLK;
    /* 若nblk为0，表示当前dirf还没有使用文件指针。这种情况下，可以直接分配磁盘块，并且返回该磁盘块的首地址即可，注意要将该首地址转为File结构体指针 */
    if(nblk == 0) {
        /* 注意，这里的data是一个uint8_t数组的首地址，返回时要将其转换为File结构体指针 */
        return (struct File *)(disk[(make_link_block(dirf, nblk))].data);
    }
```

```
    /* 如果nblk不为0，则dirf已经分配过磁盘块作为存放目录的块，我们需要在"最新"的块中寻找一个未被使用过的File结构体，返回其指针。若nblk小于等于10，bno可以从dirf的直接指针数组中取出 */
    if(nblk <= NDIRECT) {
        bno = dirf->f_direct[nblk - 1];
    } else {
    /* 若nblk大于10，那么bno需要从dirf的间接块中取出 */
        bno = ((uint32_t *)(disk[dirf->f_indirect].data))[nblk - 1];
    }
    /* 将磁盘块data转化为File结构体指针赋值给dirblk */
    dirblk = (struct File *)disk[bno].data;
    /* 遍历disk[bno]的File块，遇到没有命名即没有使用的文件块，可以返回 */
    for(i = 0;i < FILE2BLK;i++) {
        if(dirblk[i].f_name[0] == '\0') {
            return &dirblk[i];
        }
    }
    /*  */
    return (struct File *)(disk[(make_link_block(dirf, nblk))].data);
}

int make_link_block(struct File *dirf, int nblk) {
    /* 使用next_block()函数从磁盘disk数组中取出一个新的磁盘块，并将类型设为BLOCK_FILE */
    int bno = next_block(BLOCK_FILE);
    /* 将bno根据nblk的位置，复制给drif目录下的指针 */
    save_block_link(dirf, nblk, bno);
    /* dirf目录对应的文件大小自增4096字节 */
    dirf->f_size += BY2BLK;
    return bno;
}

void save_block_link(struct File *f, int nblk, int bno)
{
    /* 判断nblk是否小于1024，若大于1024，则超出文件范围，不合法 */
    assert(nblk < NINDIRECT);
    /* 如果nblk小于10，可以将bno赋值给直接链接指针的nblk索引对应的元素 */
    if(nblk < NDIRECT) {
        f->f_direct[nblk] = bno;
    } else {
    /* nblk大于10，需要使用间接指针 */
        if(f->f_indirect == 0) {
            /* 若f_indirect为0，表示该目录的间接指针块尚未被分配，使用next_block为其分配一个磁盘块，并将类型赋为BLOCK_INDEX */
            f->f_indirect = next_block(BLOCK_INDEX);
        }
        /* 给间接指针块的nblk索引位置赋值为bno，注意将格式转换为数组 */
        ((uint32_t *)(disk[f->f_indirect].data))[nblk] = bno;
    }
}
```

初次编写，可能会感觉这个函数比较有难度。这个函数不仅调用了多个其他函数、考差了学生对文件块磁盘块以及目录结构的理解，而且对C语言的指针的一些特性也进行了考察，对于C语言基础一般的同学来说我是有难度的。

2.块缓存

块缓存，指的是借助虚拟内存来实现的磁盘块缓存的技术。在我们的操作系统中，文件系统是一个用户进程，它和其他进程一样都拥有4GB的虚拟内存空间。我们将DISKMAP~DISKMAP+DISKMAX
即`[0x10000000,0x50000000)`这一段虚拟地址空间用作缓冲区，在磁盘读入内存的时候用于映射相关的页。

map_block()和ummap_block()函数

显然，当我们将一个磁盘块载入内存的时候，我们需要位置分配物理内存；当其从磁盘块中退出的时候，我们需要释放该物理内存。这一部分的操作由`fs/fs.c`的`map_block()`与`unmap_block()`来完成。

```
int map_block(u_int blockno)
{
    /* 使用block_is_mapped()函数判断blockno对应的磁盘块是否已经映射 */
    if(block_is_mapped(blockno)) return 0;
    /* 使用系统调用映射磁盘块。我们可以通过diskaddr得到虚拟地址 */
    return syscall_mem_alloc(0, diskaddr(blockno), PTE_V | PTE_R);
}

void unmap_block(u_int blockno)
{
    int r;
    /* 检查该磁盘块是否已经映射，倘若没有，函数可直接结束 */
    if(!block_is_mapped(blockno)) return;
    /* 若当前磁盘块不空闲，且被修改过，那么在解除映射之前，需要将该blockno在内存中对应的新的数据写回磁盘 */
    if(!block_is_free(blockno) && block_is_dirty(blockno)) {
        write_block(blockno);
    }
    /* 使用系统调用解除映射 */
    if((r = syscall_mem_unmap(0, diskaddr(blockno))) < 0) {
        user_panic("Failed to ummap!\n");
    }
    /* 验证解除映射的操作是否成功 */
    user_assert(!block_is_mapped(blockno));
}
```

这也是两个需要在实验中完成的函数，它们调用的一些别的函数也比较有意思：

```
u_int va_is_mapped(u_int va)
{
    /* 判断虚拟地址va是否已经被映射。这里通过检查其在二级页表中对应的页表项的PTE_V位是否有效来判断 */
    return (((*vpd)[PDX(va)] & (PTE_V)) && ((*vpt)[VPN(va)] & (PTE_V)));
}
```

```c
u_int block_is_mapped(u_int blockno)
{
    /* 使用va_is_mapped()函数进行判断 */
    u_int va = diskaddr(blockno);
    if (va_is_mapped(va)) {
        return va;
    }
    return 0;
}

int block_is_free(u_int blockno)
{
    /* 判断磁盘块是否空闲。倘若blockno超出范围，直接返回(表示不空闲或不存在) */
    if (super == 0 || blockno >= super->s_nblocks) {
        return 0;
    }
    /* 仍旧是通过位图法进行判断，该位有效则返回1 */
    if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
        return 1;
    }
    return 0;
}

u_int block_is_dirty(u_int blockno)
{
    /* 判断磁盘块是否被写过。条件是blockno对应的虚拟页被映射且被写过 */
    u_int va = diskaddr(blockno);
    return va_is_mapped(va) && va_is_dirty(va);
}

u_int va_is_dirty(u_int va)
{
    /* 判断va对应的虚拟页是否被写过。可以直接判断二级页表表项的脏位 */
    return (* vpt)[VPN(va)] & PTE_D;
}

void write_block(u_int blockno)
{
    u_int va;
    /* 写磁盘块，将blockno在虚拟内存中映射到的页的数据写回磁盘 */
    if (!block_is_mapped(blockno)) {
        user_panic("write unmapped block %08x", blockno);
    }
    /* 使用ide_write()写回。diskno为0，secno为blockno * 8，扇区号为块数乘上每块拥有的扇区数，虚拟地址使用diskaddr获得，写的扇区数也为SECT2BLK */
    va = diskaddr(blockno);
    ide_write(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
    syscall_mem_map(0, va, 0, va, (PTE_V | PTE_R | PTE_LIBRARY));
}
```

这些函数和前几个Lab的知识都有联系，也在本次实验中发挥了比较重要的作用，还是值得一看并尝试理解复习的。这里还有一个有意思的tip：

```
u_int diskaddr(u_int blockno)
{
    if(super && blockno >= super->s_nblocks)
        user_panic("Super block error...\n");
    return DISKMAP + blockno * BY2BLK;
}
```

这个函数得到blockno对应磁盘块的虚拟地址，乍一看，我们会发现我们的磁盘在MOS操作系统中，是线性映射的~~还真是简单粗暴（笑）~~。这个函数也是我们在`exercise 5.5`中需要完成的工作。

read_block()和write_block()

这两个函数用于读写磁盘块，前者可以讲指定编号的磁盘块读入内存，后者可以讲数据写入指定编号的磁盘块。`write_block()`在前面已经介绍过了，所以我们在此处仅分析`read_block()`函数的代码：

```
int read_block(u_int blockno, void **blk, u_int *isnew)
{
    u_int va;
    /* 首先检查blockno是否合法，若不合法则panic */
    if (super && blockno >= super->s_nblocks) {
        user_panic("reading non-existent block %08x\n", blockno);
    }
    /* 判断blockno对应的磁盘块是否空闲，若空闲则panic。使用位图前首先判断位图是否在内存中。 */
    if (bitmap && block_is_free(blockno)) {
        user_panic("reading free block %08x\n", blockno);
    }
    va = diskaddr(blockno);
    if (block_is_mapped(blockno)) {
        /* 如果blockno已经映射且isnew不为0，则将*isnew设为0 */
        if (isnew) {
            *isnew = 0;
        }
    } else {
        /* 如果blocknp未映射且isnew不为0，那么将*isnew设为1 */
        if (isnew) {
            *isnew = 1;
        }
        /* 使用系统调用声明一个新的页面，并且使用ide_read()将数据写入这个页面 */
        syscall_mem_alloc(0, va, PTE_V | PTE_R);
        ide_read(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
    }
    /* 将该虚拟地址赋值给blk */
    if (blk) {
        *blk = (void *)va;
    }
    return 0;
```

```
}
```

file_get_block()

```
int file_get_block(struct File *f, u_int filebno, void **blk)
{
    int r;
    u_int diskbno;
    u_int isnew;
    /* 使用给定的File结构体指针和目标块在目录f下的序号即filebno，找到目标块在磁盘块中的位置，并将该位置赋值给diskbno */
    if ((r = file_map_block(f, filebno, &diskbno, 1)) < 0) {
        return r;
    }
    /* 使用read_block()将目标磁盘块的数据读出 */
    if ((r = read_block(diskbno, blk, &isnew)) < 0) {
        return r;
    }
    return 0;
}
```

指导书在这里列举了一些函数的使用与原理，无非是想让我们的多读代码，多多了解文件系统发挥作用的方法；多阅读这些代码也有助于我们快速了解实验的原理和本质。

`exercise 5.7`的`dir_lookop()`函数是为了查找某个目录下是否存在指定的文件。

```
int dir_lookup(struct File *dir, char *name, struct File **file)
{
    int r;
    u_int i, j, nblock;
    void *blk;
    struct File *f;
    /* 计算当前文件占用了多少个磁盘块 */
    nblock = dir->f_size / BY2BLK;
    for (i = 0; i < nblock; i++) {
        /* 这里，我们可以使用我们刚刚了解的file_get_block()函数来逐一取出dir文件目录下的文件控制块 */
        if((r = file_get_block(dir ,i ,&blk)) < 0) return r;
        /* 将取出的磁盘块指针转化为File结构体指针，即文件控制块指针 */
        f = (struct File *)blk;
        /* FILE2BLK为16，表示一个磁盘块所拥有的FCB的数量 */
        for(j = 0;j < FILE2BLK;j++) {
            /* 找到同名文件！将其地址赋值给**file，返回0 */
            if(strcmp((char *)f[j].f_name ,name) == 0) {
                f[j].f_dir = dir;
                *file = &f[j];
                return 0;
            }
        }
```

```
    }
    /* 未能找到，返回错误值 */
    return -E_NOT_FOUND;
}
```

Thinking 5.5

根据`fs.h`中的宏定义的注释可知，可映射的最大磁盘空间为1GB

Thinking 5.6

显然不能，文件系统作为用户进程，倘若允许其能够访问内核地址空间，后果不堪设想。

Thinking 5.7

名称	含义	大小
BY2SECT	一张扇区的字节数	512
SECT2BLK	一个磁盘块的扇区数	8
DISKMAX	磁盘可映射的最大虚拟地址	0x40000000
BY2BLK	一个磁盘块的字节数	4096
BIT2BLK	一个磁盘块的比特数	32768
NDIRECT	File结构体的直接块指针数	10
NINDIRECT	File结构体间接指针数	1024
BY2FILE	一个FILE结构体的字节数	256
MAXFILESIZE	一个文件的最大字节数	1024 * 4096

四、文件系统的用户接口

1.用户接口

文件系统需要向用户提供使用的接口。我们的MOS操作系统是微内核操作系统，文件系统属于用户进程，向其他进程提供服务。此处，不仅涉及进程通信的问题，也涉及抽象表示文件的问题。我们引入了文件描述符作为用户程序管理、操作文件资源的方式。

2.文件描述符

用户打开文件时，需要文件描述符来存储文件的基本信息和用户进程关于文件的状态；同时，文件描述符也起到描述用户对于文件操作的作用。

这一部分的内容算是比较多，但指导书上介绍的并不多，所以我们不仅会关注实验中需要完成的代码。首先来阅读`open()`函数。

```c
/* 按照路径和模式打开其对应的文件，成功则返回文件描述符的序号，失败则返回错误码 */
int open(const char *path, int mode)
{
    struct Fd *fd;
    struct Filefd *ffd;
    u_int size, fileid;
    int r;
    u_int va;
    u_int i;
    /* 首先，使用fd_alloc()声明一个文件描述符，用于返回 */
    if((r = fd_alloc(&fd)) < 0) return r;
    /* 使用fsipc_open()与文件系统进行通信，告知文件系统路径以及打开模式 */
    if((r = fsipc_open(path, mode, fd)) < 0) return r;
    /* 已经将信息发送给文件系统，可以使用文件描述符fd。对变量一一赋值 */
    va = fd2data(fd);
    ffd = (struct Filefd *)fd;
    fileid = ffd->f_fileid;
    size = ffd->f_file.f_size;
    /* 将文件内容映射到目标位置*/
    for (i = 0; i < size; i += BY2PG) {
        if((r = fsipc_map(fileid, i, va + i)) < 0) return r;
    }
    /* 返回文件描述符的序号 */
    return fd2num(fd);
}

/* 按照路径打开文件，失败返回错误码，成功返回0 */
int fsipc_open(const char *path, u_int omode, struct Fd *fd)
{
    u_int perm;
    struct Fsreq_open *req;
    /* fsipcbuf是一个u_char类型的数组。虽然req是结构体类型，但Fsreq_open结构体的首属性是也是字符串，故而这样的转型可行 */
    req = (struct Fsreq_open *)fsipcbuf;
    /* 路径大于限制长度，返回错误码。MAXPATHLEN的大小为1024，和Fsreq_open的path数组大小相同，保证了传递不溢出。值得一提的是，fsipcbuf的大小为4096。 */
    if (strlen(path) >= MAXPATHLEN) {
        return -E_BAD_PATH;
    }
    /* 将路径从path数组拷贝至req的path数组；将模式也传递至req */
    strcpy((char *)req->req_path, path);
    req->req_omode = omode;
    /* 通过fsipc发送数据至文件系统进程 */
    return fsipc(FSREQ_OPEN, req, (u_int)fd, &perm);
}

struct Fsreq_open {
    char req_path[MAXPATHLEN];
    u_int req_omode;
```

```
}

int fsipc_map(u_int fileid, u_int offset, u_int dstva){
    int r;
    u_int perm;
    struct Fsreq_map *req;
    req = (struct Fsreq_map *)fsipcbuf;
    req->req_fileid = fileid;
    req->req_offset = offset;
    /* 仍旧是通过进程通信实现映射 */
    if ((r = fsipc(FSREQ_MAP, req, dstva, &perm)) < 0) {
        return r;
    }
    if ((perm & ~(PTE_R | PTE_LIBRARY)) != (PTE_V)) {
        user_panic("fsipc_map: unexpected permissions %08x for dstva %08x", perm,
dstva);
    }
    return 0;
}
```

可以看到，这个部分的内容和进程通信的联系非常大，可以说没有进程通信，我们的文件系统就无法向用户提供服务。

Thinking 5.8

这种转换之所以可行，应当和结构体内部属性排列顺序有关。

```
struct Fd {
    u_int fd_dev_id;
    u_int fd_offset;
    u_int fd_omode;
};

struct Filefd {
    struct Fd f_fd;
    u_int f_fileid;
    struct File f_file;
};
```

`Filefd`结构体的第一个成员便是`Fd`类型，且结构体指针本身就指向结构体的首地址，这种转换因此可行。代码中还有另一处类似的例子。

```
struct Fsreq_open *req;
req = (struct Fsreq_open *)fsipcbuf;

struct Fsreq_open {
    char req_path[MAXPATHLEN];
    u_int req_omode;
};
```

其中，`fsipcbuf`是一个`u_char`类型的数组，`req`对应结构体的第一个成员也是一个`cahr`数组。不过`fsipcbuf[]`和`req_path[]`不等长。

然后，我们来阅读`fd.c`中的`read()`函数，这个函数可以"从文件中读取数据"。

```
/* 从fdnum对应的文件描述符处读取n个字节的数据，并且存入缓冲区buf中 */
int read(int fdnum, void *buf, u_int n)
{
    int r;
    struct Dev *dev;
    struct Fd *fd;
    /* 首先使用fd_lookup()函数查找fdnum对应的文件描述符，并判断文件描述符对应的设备是否存在。若存在，赋值dev结构体，dev结构体配置了一系列访问方法 */
    if ((r = fd_lookup(fdnum, &fd)) < 0 ||
        (r = dev_lookup(fd->fd_dev_id, &dev)) < 0) {
        return r;
    }
    /* 检查权限，倘若权限为只写，则报错 */
    if ((fd->fd_omode & O_ACCMODE) == O_WRONLY) {
        writef("[%08x] read %d -- bad mode\n", env->env_id, fdnum);
        return -E_INVAL;
    }
    /* 使用dev结构体配置的访问方法读取数据 */
    r = (*dev->dev_read)(fd, buf, n, fd->fd_offset);
    /* 缓冲数据尾部添加\0保证安全 */
    if (r >= 0) {
        fd->fd_offset += r;
        *((char *)buf + r) = '\0';
    }
    return r;
}
```

`read()`函数和本文件下的另一个`write()`函数非常相似，只需有模学样即可。

Thinking 5.9

在`fs`目录下，新建一个`helloworld`文件，并通过修改Makefile将其挂载在磁盘镜像中。同时编写测试程序，在`fork()`之后得到文件描述符并打印输出，可以发现父子进程仍旧共享文件描述符。

```
void umain() {
```

```
    int r;
    int fdnum;
    if ((r = open("/newmotd", O_RDWR)) < 0) {
        user_panic("open /newmotd: %d", r);
    }
    writef("envid : %d ,fdnum : %d\n", syscall_getenvid(), r);
    fdnum = r;
    char *str = "Hello World!";
    if ((r = write(fdnum, str, strlen(str))) < 0) {
        user_panic("write /newmotd: %d", r);
    }
    fork();
    if ((r = open("/helloworld", O_RDWR)) < 0) {
        user_panic("open /helloworld: %d", r);
    }
    fdnum = r;
    writef("%s\n", str);
    if ((r = write(fdnum, str, strlen(str))) < 0) {
        user_panic("write /helloworld: %d", r);
    }
    writef("envid : %d ,fdnum : %d\n", syscall_getenvid(), r);
}
```

随后在本地的编译器中编写C语言文件：

```
#include <unistd.h>
#include "stdio.h"

int main() {
    FILE *f;
    f = fopen("in.txt", "r");
    fseek(f, 0, 0);
    fork();
    printf("env_id : %d ,f : %d\n", getppid(), f);
}
```

发现父子进程共享文件定位指针。

Thinking 5.10

```
/* Fd结构体表示文件描述符，供用户使用，是单纯的内存数据 */
struct Fd {
    u_int fd_dev_id;        //该文件所处的设备编号(id)
    u_int fd_offset;        //该文件在设备中的偏移
    u_int fd_omode;         //该文件的打开模式
};

/* Filefd结构体表示文件描述符和文件，供用户使用，是单纯的内存数据 */
```

```
struct Filefd {
    struct Fd f_fd;         //某个文件的文件描述符
    u_int f_fileid;         //打开该文件的文件id
    struct File f_file;     //该文件的文件控制块
};

/* 仅定义在serv.c中的结构体，仅由文件系统使用，用于保存已打开的文件，是单纯的内存数据 */
struct Open {
    struct File *o_file;    //所打开文件的文件控制块的指针
    u_int o_fileid;         //所打开文件的文件id
    int o_mode;             //所打开文件的打开方式
    struct Filefd *o_ff;    //该文件描述符在文件进程中的虚拟地址
};
```

这三个结构体都是在内存中的数据，围绕文件系统和用户进程定义，用于协助文件系统的实现；同时他们也是硬件中的文件在软件中的抽象映像。

3.文件系统服务

MOS操作系统中，文件系统通过IPC（进程通信）机制为其他进程提供服务。内核开始运行时，文件系统便被创建；而文件系统也有一系列的初始化机制，保证其能够正常地为其他进程提供服务。借用指导书上的UML时序图来描述文件系统服务用户进程的流程。

具体到细节，用户程序向文件系统发出操作请求时，会将请求的内容放在一系列`fsreq`结构体中进行消息的传递。文件系统进程收到IPC请求后，可以根据请求所传递的参数和要求执行相应的文件操作，并将结果通过IPC机制反馈给用户进程。

`user/fsipc.c`操作中定义了请求文件系统使用到的一系列IPC操作，`user/file.c`重则定义了提供给用户程序的读写、创建、删除、修改文件的接口。

```
/* file.c提供给用户的文件系统接口 */
/* open()函数，前面已经介绍过，用于打开某路径下的文件，成功打开则返回其文件标识符 */
int open(const char *path, int mode);

/* 关闭一个文件标识符 */
int file_close(struct Fd *fd)
{
    int r;
    struct Filefd *ffd;
    u_int va, size, fileid;
    u_int i;
    /* "经典"类型转换 */
    ffd = (struct Filefd *)fd;
    fileid = ffd->f_fileid;
    size = ffd->f_file.f_size;
    /* 得到文件表示符映射数据的虚拟地址 */
    va = fd2data(fd);
```

```
    /* 告知文件系统哪些页面为脏页，即被修改过的页面 */
    for (i = 0; i < size; i += BY2PG) {
        fsipc_dirty(fileid, i);
    }
    /* 使用IPC机制，告知文件系统需要关闭的文件id */
    if ((r = fsipc_close(fileid)) < 0) {
        writef("cannot close the file\n");
        return r;
    }
    /* 解除文件在内存中的映射，释放内存 */
    if (size == 0) {
        return 0;
    }
    /* 按照文件大小，逐一取消页面映射 */
    for (i = 0; i < size; i += BY2PG) {
        if ((r = syscall_mem_unmap(0, va + i)) < 0) {
            writef("cannont unmap the file.\n");
            return r;
        }
    }
    return 0;
}

/* 从fd指向的文件标识符对应的文件读取n字节的数据至缓冲区，位置由seek pointer决定 */
static int file_read(struct Fd *fd, void *buf, u_int n, u_int offset)
{
    u_int size;
    struct Filefd *f;
    f = (struct Filefd *)fd;
    /* 获取文件大小 */
    size = f->f_file.f_size;
    /* 偏移大于文件大小，返回0 */
    if (offset > size) {
        return 0;
    }
    /* 读取最后的长度大于文件大小，则只需读到文件末尾 */
    if (offset + n > size) {
        n = size - offset;
    }
    /* 进行拷贝，读取完成 */
    user_bcopy((char *)fd2data(fd) + offset, buf, n);
    return n;
}

/* 将n字节的数据从缓冲区写至fd指向的文件表示符对应的文件，位置同样由seek point决定 */
static int file_write(struct Fd *fd, const void *buf, u_int n, u_int offset)
{
    int r;
    u_int tot;
```

```
    struct Filefd *f;
    f = (struct Filefd *)fd;
    /* tot为写完后的文件大小 */
    tot = offset + n;
    /* 写后溢出，返回错误码。此处溢出指的是超出了文件大小的最大上限 */
    if (tot > MAXFILESIZE) {
        return -E_NO_DISK;
    }
    /* 如果本次写操作将使文件增大，则增加文件大小 */
    if (tot > f->f_file.f_size) {
        if ((r = ftruncate(fd2num(fd), tot)) < 0) {
            return r;
        }
    }
    /* 进行拷贝，完成写操作 */
    user_bcopy(buf, (char *)fd2data(fd) + offset, n);
    return n;
}

/* 截断或增长文件大小，使之等于size。这个函数比较长，不在此解读了 */
int ftruncate(int fdnum, u_int size)

/* 按照路径，删除一个文件或者目录 */
int remove(const char *path)
{
    /* 直接调用fsipc.c中的函数即可 */
    return fsipc_remove(path);
}
```

这上面的函数中，`remove()`函数是我们需要完成的函数。不过它十分简单，只需要我们调用一个定义在`fsipc.c`中的函数即可。既然这个文件中的函数也有这么强大的功能，下面我们就来解读一下它的部分代码。

```
/* 传递type和fsreq至文件系统进程，其中type使用value传递。dstva为接受返回数据的虚拟地址，perm为该虚拟地址对应的页面的权限 */
static int fsipc(u_int type, void *fsreq, u_int dstva, u_int *perm)
{
    u_int whom;
    /* 使用ipc_send()发送数据 */
    ipc_send(envs[1].env_id, type, (u_int)fsreq, PTE_V | PTE_R);
    /* 等待进程通信返回结果 */
    return ipc_recv(&whom, dstva, perm);
}

/* 按照路径path和打开方式omode打开文件，上文已介绍 */
int fsipc_open(const char *path, u_int omode, struct Fd *fd);

/* 告知文件系统进程设置文件的大小 */
int fsipc_set_size(u_int fileid, u_int size)
{
```

```
    struct Fsreq_set_size *req;
    /* req使用某个结构体，作为想文件系统发出某种请求的媒介 */
    req = (struct Fsreq_set_size *)fsipcbuf;
    /* 设置结构体的信息 */
    req->req_fileid = fileid;
    req->req_size = size;
    /* 通过fsipc函数传递操作类型(FSREQ_SET_SIZE)及细节数据(req) */
    return fsipc(FSREQ_SET_SIZE, req, 0, 0);
}

/* 告知文件系统进程将某文件下的某个块标记为脏，即已修改 */
int fsipc_dirty(u_int fileid, u_int offset)
{
    struct Fsreq_dirty *req;
    /* 同样，使用另外一个结构体传递信息 */
    req = (struct Fsreq_dirty *)fsipcbuf;
    req->req_fileid = fileid;
    req->req_offset = offset;
    /* 通过fsipc函数传递操作类型以及细节数据 */
    return fsipc(FSREQ_DIRTY, req, 0, 0);
}

/* 告知文件系统按照路径删除某个文件 */
int fsipc_remove(const char *path)
{
    struct Fsreq_remove *req;
    /* 首先，检查文件路径是否合法 */
    if (strlen(path) >= MAXPATHLEN) return -E_BAD_PATH;
    /* 使用合适的结构体，存放细节数据 */
    req = (struct Fsreq_remove *)fsipcbuf;
    /* 结构体中需要存放的细节数据为文件路径 */
    strcpy((char *)req->req_path, path);
    /* 使用fsipc()函数向文件系统发出请求 */
    return fsipc(FSREQ_REMOVE, req, 0, 0);
}
```

简单阅读了几个函数，发现他们的工作流程大致相同：将缓冲区`fsipcbuf`转型为相应的结构体指针，赋值给`req`并向其中存放细节数据，然后调用`fsipc()`函数向文件系统请求服务，注意所请求服务的类型。

最后，我们还需要在文件系统中添加对应的服务函数，以此完善文件系统的功能。

```
void serve_remove(u_int envid, struct Fsreq_remove *rq)
{
    int r;
    u_char path[MAXPATHLEN];
    /* 拷贝路径，并在最后加上\0 */
    user_bcopy(rq->req_path, path, MAXPATHLEN);
    path[MAXPATHLEN - 1] = '\0';
    /* 调用文件系统的函数file_remove()删除该文件，同时利用IPC机制向用户进程发送服务完成的信息，唤醒用户进程并告知服务结果 */
    if((r = file_remove(path)) < 0) ipc_send(envid, r, 0, 0);
    ipc_send(envid, 0, 0, 0);
}
```

这便是完成文件系统服务的一套流程。

Thinking 5.11

图中有三类箭头，代表UML类图中的三类消息：

- 黑色三角+实线+实心黑点组成的箭头。这类消息可能没有发送者或接受者。
- 黑色三角+实线组成的箭头。表示同步消息，发送者发送消息后，暂停活动等待接受者响应。
- 开三角+虚线组成的箭头。表示返回消息，和同步消息一并使用。

在前面的实验中，我们已经实现了IPC机制，不同进程之间可以通过IPC机制进行通信。我们的文件系统对应的进程控制块为`envs[1]`，可以通过这个控制块的信息从而使用IPC机制在用户进程和文件系统进程之间实现进程通信。

Thinking 5.12

这一循环，每次都会调用`ipc_recv()`函数，以检测其他进程是否向文件系统进程发送数据，倘若有，则响应；倘若没有，则阻塞。这一循环不会使CPU陷入忙等，且文件系统进程为用户进程，不会损害到内核；MOS系统结束运行时，该进程会被`kill`而不会长期存在。