CHAPTER 5

文件系统

5.1 实验目的

1. 了解文件系统的基本概念和作用。
2. 了解普通磁盘的基本结构和读写方式。
3. 了解实现设备驱动的方法。
4. 掌握并实现文件系统服务的基本操作。
5. 了解微内核的基本设计思想和结构。

5.2 文件系统概述

计算机的文件系统是一种存储和组织计算机数据的方法，它使得计算机程序对数据访问和查找变得容易，文件系统使用文件和树形目录的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念，用户使用文件系统来保存数据不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的那个块地址没有被使用，硬盘上的存储空间管理（分配和释放）功能由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。

文件系统通常使用硬盘和光盘这样的存储设备，并维护文件在设备中的物理位置。但是，实际上文件系统也可能仅仅是一种访问数据的界面而已，实际的数据是通过网络协议（如 NFS、SMB、9P 等）提供的或者内存上，甚至可能根本没有对应的文件（如 proc 文件系统）。

严格地说，文件系统是一套实现了数据的存储、分级组织、访问和获取等操作的抽象数据类型（Abstract data type）。

Thinking 5.1 查阅资料，了解 Linux/Unix 的 /proc 文件系统是什么？有什么作用？Windows 操作系统又是如何实现这些功能的？proc 文件系统这样的设计有什么好处和可以改进的地方？ ■

5.2.1 磁盘文件系统

磁盘文件系统是一种设计用来利用数据存储设备来保存计算机文件的文件系统，最常用的数据存储设备是磁盘驱动器，可以直接或者间接地连接到计算机上。严格来说，磁盘文件系统和操作系统中使用的文件系统不一定相同，如我们可以在 Linux 中挂载使用 Ext4、FAT32 等多种文件系统的磁盘驱动器，但是 Linux 中运行的程序访问这些文件系统的界面都是 Linux 的 VFS 文件系统。

本实验中实现的磁盘文件系统和操作系统中使用的文件系统拥有相同的结构，即便如此，其在磁盘上和内存中的表示也存在一定的差异；下文中我们使用文件系统一词统一指代两者。

5.2.2 用户空间文件系统

在以 Linux 为代表的宏内核操作系统中，文件系统是内核的一部分。文件系统作为内核资源的索引发挥了重要的定位内核资源的作用，重要的 mmap，ioctl, read, write 操作都依赖文件系统实现。与此相对的是众多微内核中使用的用户空间文件系统，其特点是文件系统在用户空间中实现，通过特殊的系统调用接口或者通用机制为其他用户程序提供服务。与此概念相关的还有用户态驱动的概念。

5.2.3 文件系统的设计与实现

在本次实验中，我们将要实现一个简单但结构完整的文件系统。整个文件系统包括以下几个部分：

1. **外部储存设备驱动** 通常，外部设备的操作需要通过按照一定操作序列读写特定的寄存器来实现。为了将这种操作转化为具有通用、明确语义的接口，我们必须实现相应的驱动程序。在本部分，我们实现了 IDE 磁盘的用户态驱动程序。

2. **文件系统结构** 在本部分，我们实现磁盘上和操作系统中的文件系统结构，并通过驱动程序实现文件系统操作相关函数。

3. **文件系统的用户接口** 在本部分，我们提供接口和机制使得用户程序能够使用文件系统，这主要通过一个用户态的文件系统服务来实现。同时，我们引入了文件描述符等结构使操作系统和用户程序可以抽象地操作文件而忽略其实际的物理表示。

接下来我们一一详细解读这些部分的实现。

5.3 IDE 磁盘驱动

为了在磁盘等外部设备上实现我们的文件系统，我们必须为这些外部设备编写驱动程序。实际上，我们的操作系统中已经实现了一个简单的驱动程序，那就是位于 driver 目录下的串口通信的驱动程序。在这个驱动程序中我们使用了内存映射 I/O(MMIO) 技术编写驱动。

本次要实现的硬盘驱动程序，同已经实现的串口驱动相比，相同的是我们同样使用 MMIO 技术编写磁盘的驱动；而不同之处在于，我们需要驱动的物理设备——IDE 磁盘功能更加复杂，并且本次我们编写的驱动程序**完全运行在用户空间中**。

5.3.1 内存映射 I/O

在第二次试验中，我们已经了解了 MIPS 存储器地址映射的基本内容。几乎每一种外设都是通过读写设备上的寄存器来进行数据通信，外设寄存器也称为 **I/O 端口**，我们使用 I/O 端口来访问 I/O 设备。外设寄存器通常包括控制寄存器、状态寄存器和数据寄存器。这些硬件 I/O 寄存器被映射到指定的内存空间，例如，在 Gxemul 中，console 设备被映射到 `0x10000000`，simulated IDE disk 被映射到 0x13000000，等等。更详细的关于 Gxemul 的仿真设备的说明，可以参考Gxemul Experimental Devices。

驱动程序访问的是 I/O 空间，与一般我们说的内存空间是不同的。外设的 I/O 空间地址是系统启动后才知道（具体讲，这个辛苦的工作是由 BIOS 完成后告知操作系统的），通常的体系结构（如 x86）并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，所以驱动程序并不能直接通过物理地址访问 I/O 内存资源，而**必须将它们映射到内核虚地址空间内然后才能根据映射所得到的核心虚地址范围**，通过访存指令访问这些 I/O 内存资源。

幸运的是实验中使用的 MIPS 体系结构并没有复杂的 I/O 端口的概念，而是统一使用内存映射 I/O 的模型。MIPS 的地址空间中，其在内核地址空间中（kseg0 和 kseg1 段）实现了硬件级别的物理地址和内核虚拟地址的转换机制，其中，对 kseg1 段地址的读写是未缓存（uncached）的，即所有的操作不会写入高速缓存，这种可见性正是设备驱动所需要的。由于我们在模拟器上运行操作系统，I/O 设备的物理地址是完全固定的。这样一来我们的驱动程序任务就转变成了简单的读写某些固定的内核虚拟地址。

我们的操作系统内核中，之前的操作钟，将物理内存转换为内核虚拟地址，都是转换到 kseg0 段的内核虚拟地址。使用的是`KADDR`宏，也就是将物理地址加上`ULIM`的值(`0x80000000`)。而正如我们上面提到的，编写设备驱动的时候我们需要将物理地址转换为 kseg1 段的内核虚拟地址，也就是将物理地址加上 kseg1 的偏移值 (`0xA0000000`)。

> **Thinking 5.2** 如果我们通过 kseg0 读写设备，我们对于设备的写入会缓存到 Cache 中。通过 kseg0 访问设备是一种**错误**的行为，在实际编写代码的时候这么做会引发不可预知的问题。请你思考：这么做这会引起什么问题？对于不同种类的设备（如我们提到的串口设备和 IDE 磁盘）的操作会有差异吗？可以从缓存的性质和缓存刷新的策略来考虑。 ■

以我们编写完成的串口设备驱动为例，Gxemul 提供的 console 设备的地址：0x10000000，设备寄存器映射如表5.1所示：

表 5.1: Gxemul Console 内存映射

Offset	Effect
0x00	Read: getchar() (non-blocking; returns 0 if no char was available)
	Write: putchar(ch)
0x10	Read or write: halt()
	(Useful for exiting the emulator.)

现在，我们通过往内存的 (0x10000000+0xA0000000) 地址写入字符，就能在 shell 中看到对应的输出。`drivers/gxconsole/console.c`中的`printcharc`函数的实现如下所示：

```
1 void printcharc(char ch)
2 {
3   *((volatile unsigned char *) PUTCHAR_ADDRESS) = ch;
4 }
```

而在本次实验中，我们需要编写 IDE 磁盘的驱动完全位于用户空间，用户态进程若是直接读写内核虚拟地址将会由处理器引发一个地址错误（ADEL/S)。所以我们对于设备的读写必须通过系统调用来实现。这里我们引入了 sys_write_dev 和 sys_read_dev 两个系统调用来实现设备的读写操作。这两个系统调用接受用户虚拟地址，设备的物理地址和读写的长度（按字节计数）作为参数，在内核空间中完成 I/O 操作。

Exercise 5.1 请根据 lib/syscall_all.c 中的说明，完成 `sys_write_dev` 函数和 `sys_read_dev` 函数的，并且在 user/lib.h, user/syscall_lib.c 中完成用户态的相应系统调用的接口。

编写这两个系统调用时需要注意物理地址、用户进程虚拟地址同内核虚拟地址之间的转换。

同时还要检查物理地址的有效性，在实验中允许访问的地址范围为：console: [0x10000000, 0x10000020)，disk: [0x13000000, 0x13004200)，rtc: [0x15000000, 0x15000200)，当出现越界时，应返回指定的错误码。 ■

5.3.2 IDE 磁盘

在我们的操作系统实验中，我们使用的 Gxemul 模拟器提供的“磁盘”是一个 IDE 仿真设备，我们需要此基础上实现我们的文件系统，接下来，我们将了解一些读写 IDE 磁盘的基础知识。

Note 5.3.1 IDE 的英文全称为“Integrated Drive Electronics”，即“电子集成驱动器”，是目前最主流的硬盘接口，也是光储类设备的主要接口。IDE 接口，也称之为 ATA 接口。

磁盘的物理结构

我们首先简单介绍一下与磁盘相关的基本知识。磁盘相关的几个基本概念：

1. 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。
2. 磁道 (track)：盘片上以盘片中心为圆心，不同半径的同心圆。
3. 柱面 (cylinder)：硬盘中，不同盘片相同半径的磁道所组成的圆柱。
4. 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。

典型的磁盘的基本结构如图5.1所示：

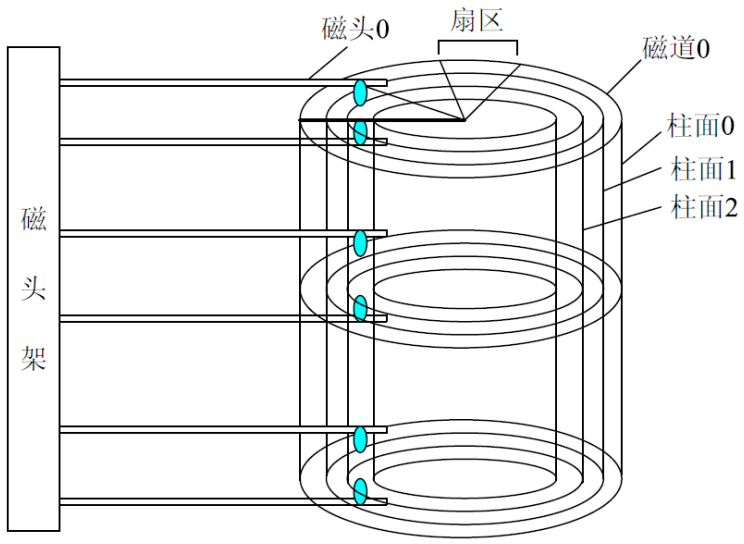

图 5.1: 磁盘结构示意图

IDE 磁盘操作

前文中我们提到过，扇区 (Sector) 是磁盘读写的基本单位，Gxemul 也提供了对扇区进行操作的基本方法。

Gxemul 提供的 Simulated IDE disk 的地址是 0x13000000，I/O 寄存器相对于 `0x13000000` 的偏移和对应的功能如表5.2所示：

5.3.3 驱动程序编写

通过对`printcharc` 函数的实现的分析，我们已经掌握了 I/O 操作的基本方法，那么，读写 IDE 磁盘的相关代码也就不难理解了。以从硬盘上读取一些 Sectors 为例，我们首先考虑一个**内核态的驱动**是如何编写的：

`read_sector` 函数：

表 5.2: Gxemul IDE disk I/O 寄存器映射

Offset	Effect
0x0000	Write: Set the offset (in bytes) from the beginning of the disk image. This offset will be used for the next read/write operation.
0x0008	Write: Set the high 32 bits of the offset (in bytes). (*)
0x0010	Write: Select the IDE ID to be used in the next read/write operation.
0x0020	Write: Start a read or write operation. (Writing 0 means a Read operation, a 1 means a Write operation.)
0x0030	Read: Get status of the last operation. (Status 0 means failure, non-zero means success.)
0x4000-0x41ff	Read/Write: 512 bytes data buffer.

```
1  extern int read_sector(int diskno, int offset);
```

```
1  # read sector at specified offset from the beginning of the disk image.
2  LEAF(read_sector)
3      sw   a0, 0xB3000010   # select the IDE id.
4      sw   a1, 0xB3000000   # offset.
5      li   t0, 0
6      sb   t0, 0xB3000020   # start read.
7      lw   v0, 0xB3000030
8      nop
9      jr   ra
10     nop
11 END(read_sector)
```

我们来分析这段代码。当需要从磁盘的指定位置读取一个 sector 时，需要调用 `read_sector` 函数来将磁盘中对应 sector 的数据读到设备缓冲区中。**所有的地址操作都需要将物理地址转换成内核虚拟地址**。这里的设备基地址对应的 kseg1 的内核虚拟地址是 0xB3000000。根据 Gxemul 提供的与 IDE disk 相关的数据表格，首先，设置 IDE disk 的 ID，从 `read_sector` 函数的声明 `extern int read_sector(int diskno, int offset);` 中可以看出，diskno 是第一个参数，对应的就是 $a0 寄存器的值，因此，将其写入到 0xB3000010 处，这样就表示我们将使用编号为 $a0 的磁盘。我们的试验中，只使用了一块 simulated IDE disk，因此，这个值应该为 0。接下来，将相对于磁盘起始位置的 offset 写入到 0xB3000000 位置，表示在距离磁盘起始处 offset 的位置开始进行磁盘操作。然后，根据 Gxemul 的 data sheet，向内存 0xB3000020 处写入 0 来开始读磁盘（如果是写磁盘，则写入 1）。最后，将磁盘操作的状态码放入 $v0 中，作为结果返回。在外层的函数中，我们通过判断 `read_sector` 函数的返回值，就可以知道读取磁盘的操作是否成功。如果成功，将这个 sector 的数据 (512 bytes) 从设备缓冲区 (offset 0x4000-0x41ff) 中拷贝到目的位置。至此，我们就完成了对磁盘的读操作。

写磁盘的操作与读磁盘的一个区别在于写磁盘需要先将要写入对应 sector 的 512 bytes 的数据放入设备缓冲中，然后向地址 0xB3000020 处写入 1 来启动操作，并从 0xB3000030 处获取写磁盘操作的返回值。

相应的，**用户态磁盘驱动**使用系统调用代替直接对物理地址的读写，完成寄存器配置和数据拷贝等功能。

Exercise 5.2 参考这个内核态驱动，完成 fs/ide.c 中的 `ide_write` 函数，以及 `ide_read` 函数，实现对磁盘的读写操作。 ■

5.4 文件系统结构

实现了 IDE 磁盘的驱动，我们就有了在磁盘上实现文件系统的基础。接下来我们设计整个文件系统的结构，并在磁盘和操作系统中分别实现对应的结构。

Note 5.4.1 Unix/Linux 操作系统一般将磁盘分成两个区域：inode 区域和 data 区域。inode 区域用来保存文件的状态属性，以及指向数据块的指针。data 区域用来存放文件的内容和目录的元信息 (包含的文件)。我们实验使用的操作系统的文件系统也采用类似的设计。

5.4.1 磁盘文件系统布局

磁盘空间的基本布局如图5.2所示。

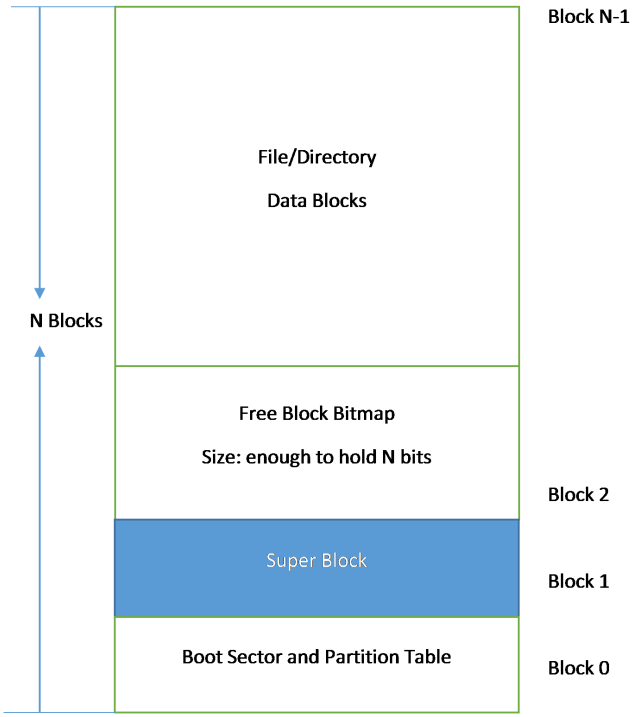

图 5.2: 磁盘空间布局示意图

从图中可以看到磁盘最开始的一个扇区 (512 字节) 被当成是启动扇区和分区表使用。接下来的一个扇区用作超级块 (Super Block)，用来描述文件系统的基本信息，如 Magic Number，磁盘大小，以及根目录的位置。

Note 5.4.2 在真实的文件系统中，一般会维护多个超级块，通过复制分散到不同的磁盘分区中，以防止超级块的损坏造成整个磁盘无法使用。

我们的操作系统中超级块的结构:

```
struct Super {
    u_int s_magic;      // Magic number: FS_MAGIC
    u_int s_nblocks;    // Total number of blocks on disk
    struct File s_root; // Root directory node
};
```

超级块的内容包括一个 magic number，磁盘中 block 的个数，以及根目录。

操作系统有两种常用的方式来管理资源：空闲链表和位图。在第二次试验和第三次试验中，我们使用了空闲链表来管理空闲内存资源和进程控制块。在文件系统中，我们将使用位图 (Bitmap) 法来管理空闲的磁盘资源。具体来说，我们使用一个二进制位来表示磁盘上的一个块 (Block) 是否使用。

接下来，我们来学习如何使用 bitmap 来标识磁盘中的所有块的使用情况。

tools/fsformat 是用于创建符合我们定义的文件系统结构的工具，用于将多个文件按照我们的内核所定义的文件系统写入到磁盘镜像中。这里我们参考 tools/fsformat 表述文件系统标记空闲块的机制。在写入文件之前，我们将所有的块全部都标为空闲块:

```
    nbitblock = (nblock + BIT2BLK - 1) / BIT2BLK;
    for(i = 0; i < nbitblock; ++i) {
        memset(disk[2+i].data, 0xff, nblock/8);
    }
    if(nblock != nbitblock * BY2BLK) {
        diff = nblock % BY2BLK / 8;
        memset(disk[2+(nbitblock-1)].data+diff, 0x00, BY2BLK - diff);
    }
```

nbitblock 表示要记录整个磁盘上所有块的使用信息，我们需要多少个块来存储位图。紧接着，我们将所有位图块的每一位都标记为 0x1，表示这一块磁盘处于空闲状态。需要格外注意的是，如果 bitmap 还有剩余，我们不能将最后一块位图块靠后的一部分内容标记为空闲，因为这些位所对应的磁盘块并不存在，不可被使用。因此，在 fs/fsformat.c 中的 `init_disk` 函数中，在将所有的位图块都置为 0x1 之后，还需要根据实际情况，将多出来的位图标记为 0x0。

相对应的，在我们的操作系统中，文件系统也需要根据 bitmap 来判断和标记磁盘的使用情况。fs/fs.c 中的 `block_is_free` 函数就用来通过 bitmap 中的特定位来判断指定的磁盘块是否被占用。

```
int block_is_free(u_int blockno)
{
    if (super == 0 || blockno >= super->s_nblocks) {
        return 0;
    }
    if (bitmap[blockno / 32] & (1 << (blockno % 32))) {
        return 1;
```

```
    }
    return 0;
}
```

Exercise 5.3 文件系统需要负责维护磁盘块的申请和释放，在回收一个磁盘块时，需要更改 bitmap 中的标志位。如果要将一个磁盘块设置为 free，只需要将 bitmap 中对应的 bit 的值设置为 0x1 即可。请完成 fs/fs.c 中的 `free_block` 函数，实现这一功能。同时思考为什么参数 `blockno` 的值不能为 0 ？

```
// Overview:
//  Mark a block as free in the bitmap.
void
free_block(u_int blockno)
{
    // Step 1: Check if the parameter `blockno` is valid (`blockno` can't be zero).

    // Step 2: Update the flag bit in bitmap.

}
```

■

5.4.2 文件系统详细结构

在操作系统的学习中，我们多次提到，操作系统要想管理一类资源，就得有相应的数据结构。我们使用文件控制块来描述和管理文件。文件在磁盘上的组织形式如图5.3所示：

对应的内存中的文件控制块的定义：

```
// file control blocks, defined in include/fs.h
struct File {
    u_char f_name[MAXNAMELEN];  // filename
    u_int f_size;           // file size in bytes
    u_int f_type;           // file type
    u_int f_direct[NDIRECT];
    u_int f_indirect;
    struct File *f_dir;
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];
};
```

我们使用的操作系统内核中，文件名的最大长度为 MAXNAMELEN(128)，每个文件控制块设有 10 个直接指针，用来记录文件的数据块在磁盘上的位置。每个磁盘块的大小为 4KB，也就是说，这十个直接指针能够表示最大 40KB 的文件，而当文件的大小大于 40KB 时，就需要用到间接指针。File 结构体中有一个域为 `f_indirect`，只想一个间接磁盘块，用来存储指向文件内容的磁盘块的指针。为了简化计算，我们不使用间接磁盘块的前十个指针。文件控制块的结构如图5.3。

Note 5.4.3 我们的文件系统中文件控制块只是用了一级间接指针域，也只有一个。而在真实的文件系统中，对了支持更大的文件，通常会使用多个间接磁盘块，或使用多级间接磁盘块。我们使用的操作系统内核在这一点上做了极大的简化。

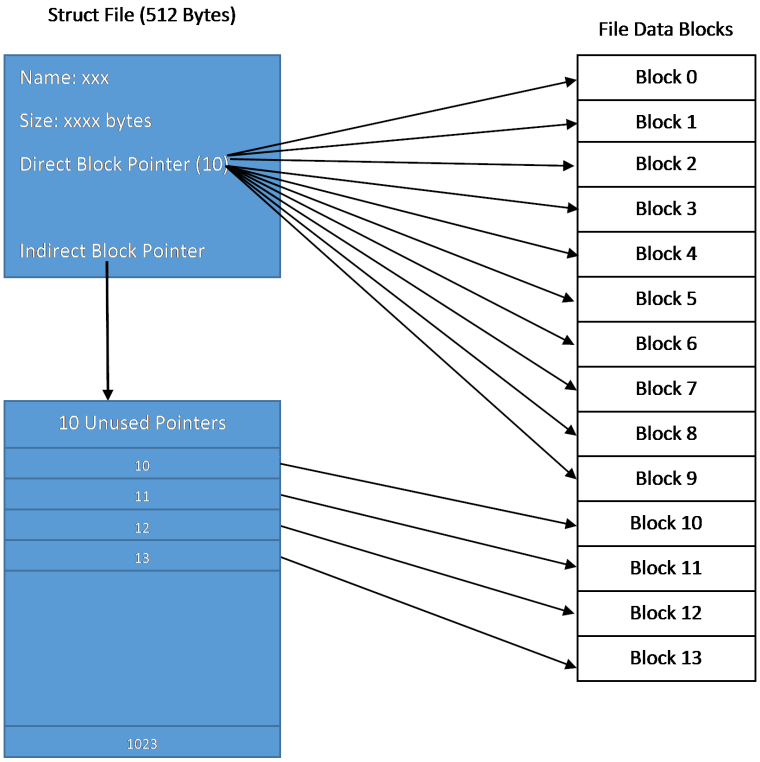

图 5.3: 文件控制块

Thinking 5.3 一个 Block 最多存储 1024 个指向其他磁盘块的指针，试计算，我们的文件系统支持的单个文件的最大大小为多大？ ■

对于普通的文件，其指向的磁盘块存储着文件内容，而对于目录文件来说，其指向的磁盘块存储着该目录下各个文件对应的的文件控制块。当我们要查找某个文件时，首先从超级块中读取根目录的文件控制块，然后沿着目标路径，挨个查看当前目录的子文件是否与下一级目标文件同名，如此便能查找到最终的目标文件。

Thinking 5.4 查找代码中的相关定义，试回答一个磁盘块中最多能存储多少个文件控制块？一个目录最多能有多少个子文件？ ■

为了更加细致地了解文件系统的内部结构，我们使用 C 语言程序 (fs/fsformat.c) 来模拟对磁盘的操作，掌握如何将文件和文件夹按照文件系统的格式写入磁盘，我们也正是通过 fsformat 程序来创建一个磁盘文件 fs/fs.img 供内核使用。请阅读 fs/fsformat.c 中的代码，掌握文件系统结构的具体细节。

Exercise 5.4 请文件系统的设计，完成 fsformat.c 中的 `create_file`函数，并按照个人兴趣完成 `write_directory` 函数（不作为考察点），实现将一个文件或指定目录下的文件按照目录结构写入到 fs/fs.img 的根目录下的功能。关于如何创建二进制文件的镜像，请参考 fs/Makefile。

在实现的过程中，你可以将你的实现同我们给出的参考可执行文件 tools/fsformat

进行对比。具体来讲，你可以通过 Linux 提供的 xxd 命令将两个 fsformat 产生的二进制镜像转化为可阅读的文本文件，手工进行查看或使用 diff 等工具进行对比。 ■

5.4.3 块缓存

块缓存指的是借助虚拟内存来实现磁盘块缓存的设计。我们的操作系统中，文件系统服务是一个用户进程（将在下文介绍），一个进程可以拥有 4G 的虚拟内存空间，将 `DISKMAP` 到 `DISKMAP+DISKMAX` 这一段虚存地址空间 (0x10000000-0xcfffffff) 作为缓冲区，当磁盘读入内存时，用来映射相关的页。`DISKMAP` 和 `DISKMAX` 的值定义在 fs/fs.h 中：

```
1 #define DISKMAP   0x10000000
2 #define DISKMAX   0xc0000000
```

Thinking 5.5 请思考，在满足磁盘块缓存的设计的前提下，我们实验使用的内核支持的最大磁盘大小是多少？ ■

为了建立起磁盘地址空间和进程虚存地址空间之间的缓存映射，我们采用这样一种设计：结构如图5.4。

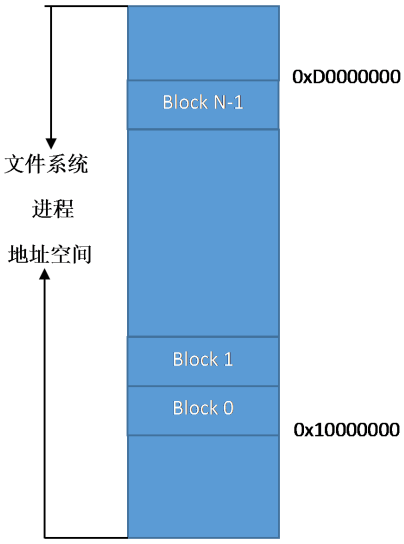

图 5.4: 块缓存示意图

Exercise 5.5 fs/fs.c 中的 diskaddr 函数用来计算指定磁盘块对应的虚存地址。完成 `diskaddr` 函数，根据一个块的序号 (block number)，计算这一磁盘块对应的 512 bytes 虚存的起始地址。(提示：fs/fs.h 中的宏 `DISKMAP` 和 `DISKMAX` 定义了磁盘映射虚存的地址空间)。 ■

当我们把一个磁盘块 (block) 中的内容载入到内存中时，我们需要为之分配对应的物理内存，当结束使用这一磁盘块时，需要释放对应的物理内存以回收操作系统资源。fs/fs.c 中的 `map_block` 函数和 `unmap_block` 函数实现了这一功能。

Exercise 5.6 实现 `map_block` 函数，检查指定的磁盘块是否已经映射到内存，如果没有，分配一页内存来保存磁盘上的数据。对应地，完成 `unmap_block` 函数，用于解除磁盘块和物理内存之间的映射关系，回收内存。(提示：注意磁盘虚拟内存地址空间和磁盘块之间的对应关系)。 ■

`read_block` 函数和 `write_block` 函数用于读写磁盘块。`read_block` 函数将指定编号的磁盘块都入到内存中，首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 `ide_read` 函数来读取磁盘上的数据到对应的虚存地址处。

在完成块缓存系统之后我们就可以实现从操作系统的文件系统操作函数：

Exercise 5.7 补完 `dir_lookup` 函数，查找某个目录下是否存在指定的文件。(提示：使用`file_get_block`可以将某个指定文件指向的磁盘块读入内存)。 ■

5.5 文件系统的用户接口

文件系统在建立之后，需要向用户提供相关的接口使用。我们实验使用的操作系统内核符合一个典型的微内核的设计，文件系统属于用户态进程，以服务的形式供其他进程调用。这个过程中，不仅涉及了不同进程之间通信的问题，也涉及了文件系统如何隔离底层的文件系统实现，抽象地表示一个文件的问题。首先，我们引入文件描述符（file descriptor）作为用户程序管理、操作文件资源的方式。

5.5.1 文件描述符

当用户进程试图打开一个文件时，需要一个文件描述符来存储文件的基本信息和用户进程中关于文件的状态；同时，文件描述符也起到描述用户对于文件操作的作用。当用户进程向文件系统发送打开文件的请求时，文件系统进程会将这些基本信息记录在内存中，然后由操作系统将用户进程请求的地址映射到同一个物理页上，因此一个文件描述符至少需要独占一页的空间。当用户进程获取了文件大小等基本信息后，再次向文件系统发送请求将文件内容映射到指定内存空间中。

Thinking 5.6 阅读 user/file.c 中的众多代码，发现很多函数中都会将一个 struct Fd * 型的指针转换为 struct Filefd * 型的指针，请解释为什么这样的转换可行。 ■

Exercise 5.8 完成 user/file.c 中的 `open` 函数。(提示：若成功打开文件则该函数返回文件描述符的编号)。 ■

当我们要读取一个大文件中间的一小部分内容时，从头读到尾是极为浪费的，因此我们需要一个指针帮助我们在文件中定位，在 C 语言中拥有类似功能的函数是 `fseek`。而在读写期间，每次读写也会更新该指针的值。请自行查阅 C 语言有关文件操作的函数，理解相关概念。

Exercise 5.9 参考 user/fd.c 中的 `write` 函数，完成 `read` 函数。 ■

Thinking 5.7 请解释 File, Fd, Filefd, Open 结构体及其各个域的作用。比如各个结构体会在哪些过程中被使用，是否对应磁盘上的物理实体还是单纯的内存数据等。说明形式自定，要求简洁明了，可大致勾勒出文件系统数据结构与物理实体的对应关系与设计框架。 ■

5.5.2 文件系统服务

我们的操作系统中的文件系统服务通过 IPC 的形式供其他进程调用，进行文件读写操作。具体来说，在内核开始运行时，就启动了文件系统服务进程 `ENV_CREATE(fs_serv)`，用户进程需要进行文件操作时，使用 `ipc_send/ipc_recv` 与 `fs_serv` 进行交互，完成操作。在文件系统服务进程的初始化函数中，首先调用了 `serv_init` 函数准备好全局的文件打开记录表 `opentab`，然后调用 `fs_init` 函数来初始化文件系统。`fs_init` 函数首先通过读取超级块的内容获知磁盘的基本信息，然后检查磁盘是否能够正常读写，最后调用 `read_bitmap` 函数检查磁盘块上的位图是否正确。执行完文件系统的初始化后，`serve` 函数被调用，文件系统服务开始运行，等待其他程序的请求。

Thinking 5.8 阅读`serve`函数的代码，我们注意到函数中包含了一个死循环`for (;;) {...}`，为什么这段代码不会导致整个内核进入 panic 状态？ ■

文件系统支持的请求类型定义在 include/fs.h 中，包含以下几种：

```
1 #define FSREQ_OPEN      1
2 #define FSREQ_MAP       2
3 #define FSREQ_SET_SIZE  3
4 #define FSREQ_CLOSE     4
5 #define FSREQ_DIRTY     5
6 #define FSREQ_REMOVE    6
7 #define FSREQ_SYNC      7
```

用户程序在发出文件系统操作请求时，将请求的内容放在对应的结构体中进行消息的传递，`fs_serv` 进程收到其他进行的 IPC 请求后，IPC 传递的消息包含了请求的类型和其他必要的参数，根据请求的类型执行相应的对文件系统的操作（文件的增、删、改、查等），将结果重新通过 IPC 反馈给用户程序。

Exercise 5.10 文件 user/fsipc.c 中定义了请求文件系统时用到的 IPC 操作，user/file.c 文件中定义了用户程序读写、创建、删除和修改文件的接口。完成 user/fsipc.c 中的 `fsipc_remove` 函数、user/file.c 中的 `remove` 函数，以及 fs/serv.c 中的 `serve_remove` 函数，实现删除指定路径的文件的功能。 ■

5.6 正确结果展示

在 init/init.c 中启动一个 fstest 进程和文件系统服务进程：

```
    ENV_CREATE(user_fstest);
    ENV_CREATE(fs_serv);
```

就能开始对文件系统的检测，运行文件系统服务，等待应用程序的请求。注意到我们必须将文件系统进程作为一号进程启动，其原因是我们在 user/fsipc.c 中定义的文件系统 ipc 请求的目标 env_id 为 1。

Note 5.6.1 使用 gxemul -E testmips -C R3000 -M 64 -d gxemul/fs.img elf-file 运行 (其中 elf-file 是你编译生成的 vmlinux 文件的路径)。

```
FS is running
FS can do I/O
superblock is good
diskno: 0
diskno: 0
read_bitmap is good
diskno: 0
alloc_block is good
file_open is good
file_get_block is good
file_flush is good
file_truncate is good
diskno: 0
file rewrite is good
serve_open 00000800 ffff000 0x2
open is good
read is good
diskno: 0
serve_open 00000800 ffff000 0x0
open again: OK
read again: OK
file rewrite is good
file remove: OK
```

5.7 实验思考

- Unix /proc 文件系统
- 设备操作与高速缓存
- 磁盘最大体积
- 文件系统支持的单个文件的最大体积
- 一个磁盘块最多存储的文件控制块及一个目录最多子文件
- 文件系统服务进程运行机制
- 文件系统的中的数据结构
- 用户进程和文件系统映射的不同

5.8 挑战性任务

本文中的用户态驱动部分使用了系统调用来实现了读写特定的内存地址（即设备的 MMIO 内存地址），文件系统进程也使用了固定的进程号来标识。这种设计有着一定的安全风险的同时有违微内核的设计思路。在微内核设计领域，针对这一问题已经提出了内存控制、权限分派、IPC 转发等等的解决方案。请你调研相关资料，修改系统调用或编写新的系统调用使得 **1)** 只有文件系统进程能读写 IDE 磁盘。**2)** 文件系统进程仅能读写 IDE 磁盘而不能读写其他的内核地址。