Lab 05

Contents

在这个实验中我们将实现一个简单的文件系统，完成该实验后， 你将至少可以在游戏中以文件名的形式读取文件。

实验提交

截止时间: 2016/06/01 23:59:59 (如无特殊原因，迟交的作业将损失50%的成绩(即使迟了 1 秒)，请大家合理分配时间)

请大家在提交的实验报告中注明你的邮箱，方便我们及时给你一些反馈信息。

学术诚信: 如果你确实无法完成实验，你可以选择不提交，作为学术诚信的奖励，你将会获得10%的分数；但若发现抄袭现象，抄袭双方(或团体)在本次实验中得 0 分。

提交地址：http://cslabcms.nju.edu.cn/

提交格式: 你需要将整个工程打包上传，特别地，我们会清除中间结果重新编译，若编译不通过，你将损失相应的分数（请在报告中注明你实验所使用的 gcc 的版本，以便助教处理一些 gcc 版本带来的问题）. 我们会使用脚本进行批量解压缩. 压缩包的命名只能包含你的学号。另外为了防止编码问题，压缩包中的所有文件都不要包含中文.如果你需要多次提交，请先手动删除旧的提交记录(提交网站允许下载，删除自己的提交记录)，否则若脚本解压时出现多次提交相互覆盖的现象，后果自负.我们只接受以下格式的压缩包：

- tar.gz
- tar.bz2
- zip

若提交的压缩包因格式原因无法被脚本识别，后果自负。

请你在实验截止前务必确认你提交的内容符合要求(格式、相关内容等)，你可以下载你提交的内容进行确认。如果由于你的原因给我们造成了不必要的麻烦，视情况而定，在本次实验中你将会被扣除一定的分数，最高可达 50% 。

git 版本控制：我们建议你使用 git 管理你的项目，如果你提交的实验中包含均匀合理的 git 记录，你将会获得 10% 的分数奖励（请注意，本实验的 Makefile 是由你自己准备的，你可以选择像 PA 中一样在每一次 make 后增加新的 git 记录作为备份，但是请注意，这样生成的 git log 一般是无意义的，所以不能作为加分项）。为此，请你确认提交的压缩包中包含一个名为 .git 的文件夹。

实验报告要求：仅接受 pdf 格式的实验报告，不超过 3 页 A4 纸，字号不能小于五号，尽可能表现出你实验过程的心得，你攻克的难题，你踩的不同寻常的坑。

分数分布： - 实验主体：80% - 实验报告：20%

解释：

1. 每次实验最多获得满分；
2. git 的分数奖励是在实验主体基础上计算的
3. git 记录是否“均匀合理”由助教判定；
4. 迟交扣除整个实验分数的 50%；
5. 作弊扣除整个实验分数的 100%；
6. 提交格式不合理扣除整个实验分数的一定比例；
7. 实验批改将用随机分配的方式进行；
8. 保留未解释细节的最终解释权；
9. 答辩时未能答对问题会扣掉总体 5% ~ 30% 的分数。

基本概念

一个文件系统包含两个部分：数据结构和操作接口。

数据结构

文件系统的数据结构用于解释磁盘中的字节流的具体含义。 经典的磁盘以 512B 为单位（扇区大小）读取数据，所以数据结构往往以 512B 的整数倍为界限进行组织。 文件系统以块作为数据的单位，它的大小是扇区的整数倍，下面默认块的大小也是 512B.

需要描述的内容主要有：

1. 磁盘的元信息：比如哪些块是可以分配的；
2. 目录结构：记录目录下的文件名以及找到描述文件数据结构的信息，目录也是文件的一种；
3. 文件结构：主要针对常规文件(regular file)，记录文件的元信息，比如文件名、文件大小以及寻找具体数据块的信息。

上手文件系统的一个困难就是要保证磁盘上的字节流与数据结构的一致性、正确性， 毕竟对于一个有内容的磁盘，我们只能先读出磁盘的数据，并将自行设计的文件系统的数据结构应用在其上。 打个比方，在 C 等语言中进行格式化输出有如下的写法：

```
printf("%d %lf\n", foo, bar);
```

这里 `%d` 和 `%lf` 可以理解为我们使用的文件系统的数据结构，而压到栈上的 `foo` 和 `bar` 则对应磁盘中的数据。 如果 `foo` 不是 `int`，或者 `bar` 不是 `double` 类型，则会输出错误的值甚至产生访问违例。 而在文件系统中，把目录文件解读成常规文件，把常规文件解读成目录，读取甚至覆写了错误的块，这都是我们不期望的。 要避免这个问题，需要仔细地确认对数据结构的操作过程，并进行反复的读写测试。

文件系统的数据结构设计相对自由，除了元信息，最核心的概念其实是链表，只不过指针换成了磁盘偏移量。

操作接口

文件系统的操作接口在上学期的 PA4 中已经涉及，相关的资料在此： 2014版、 2015版 。建议进行阅读、或者回忆自己在 PA4 里的工作。

基本的操作接口有：

1. open: 打开 / 创建文件
2. read: 读取文件，自动移动偏移量
3. write: 覆写文件，自动移动偏移量
4. lseek: 修改偏移量
5. close: 关闭文件
6. remove: 删除文件

1 ~ 5 是 Linux 提供的系统原语，而 6 是 C 标准库的函数，它对常规文件使用系统原语 `unlink`, 对目录文件使用系统原语 `rmdir` 来进行删除。 通过 `man open` 等了解它们的函数原型、参数和返回值的含义，以为自己的设计提供参考。

在内核中，每个进程的 PCB 中都要记录它拥有 / 打开了哪些文件，这通过文件描述符来进行标记。

一般设计中，我们都是在内存中维护文件的读写状态的。这样一个状态的集合就称为文件控制块（下称 FCB）。 你可以自然而然的将 FCB 与 PCB 的概念对应起来，于是相应的，文件描述符与进程号的概念也就对应起来了。 一般来说，我们使用 FCB 数组的下标作为文件描述符。

在一个 FCB 中，最基本的是要记录它对应哪个文件、它以何种方式被打开（读还是写）、以及当前读写的偏移量。 在 PA4 中， `Fstate` 对应 FCB, 但是 `Fstate` 数组与写死的文件列表是紧耦合的，文件描述符与仙剑的数据文件是一一对应的。 而我们的文件系统，描述符与文件名是脱钩的。

FCB 中可以准备一个大小为块大小倍数的数组作为内存缓冲区，以简化读写操作，减少磁盘访问。

实验内容 / 步骤

文件系统的设计相对自由，除了磁盘读写的操作涉及硬件，怎么解释磁盘中的字节流，完全可以按照自己的口味来。 不过为了简化实验，我们对文件系统提出一个底线要求，然后在这个底线上描述推荐的实验步骤，最后会罗列一些挑战性的内容。

按照 KISS 原则，只要能支持按照文件名读取文件，我们的文件系统最差可以有如下功能：

1. 没有目录，只有隐含的根目录，文件数量有上线
2. 目录项结构体中记录文件名、文件大小、文件信息块的偏移量
3. 文件信息块中记录的全部是数据块的偏移量
4. 使用 bitmap 标记哪些块可用，bitmap 大小固定（也就意味着磁盘大小固定）
5. bitmap 和根目录的大小和位置固定

Bitmap

空闲空间管理，像之前实现的物理空闲页本质上也是 bitmap, 因为它为每个物理页都维护了信息。 在动态内存分配中，常见空闲区块链表的设计，即一个链表结点记录连续空闲空间的开始和结束，并用链表串联起来， 这样节省了维护的空间开销，并且容易扩展，不过实现比较麻烦。 Bitmap 应该是最容易的管理方法了，但是它不易扩展且额外开销大。 在 bitmap 中，连续的 bit 表明连续的块是否已经被分配出去了。 如果真的以 bit 为单位记录，那么一个块大小的 bitmap 可以描述 2MB 磁盘空间（1024 进制），利用率就是 1 -

(512B / 2MB)。 如果你是位操作苦手，也可以直接以 char 作为标记。

分配块的过程，就是遍历 bitmap, 找到第一个为 0 的 bit 置 1, 获得它在 bitmap 中的偏移量，即块偏移量，乘以块大小，就是磁盘的字节偏移量。 删除块的过程，就是根据块偏移量找到对应的 bit 将其置 0.

一个支持 512MB 固定大小磁盘的 bitmap 结构定义参考如下：

```
struct bitmap {
  uint8_t mask[512 * 256];  // 512B ~ 2MB
};
```

根目录

你需要关心的是目录表项的设计，前面已经提到目录项需要记录的内容。最简单的情况下，根目录的大小是固定的（文件名设置最大长度）， 且根目录的位置在磁盘中也是固定的，即在 bitmap 的后面。

你可以为目录项中的偏移量指定一个特殊值，如 0 或者 -1，用于标记有效性。

只有一个块大小的，固定文件数量的根目录结构参考：

```
struct dirent {
  char     filename[24];
  uint32_t file_size;
  uint32_t inode_offset;
};  // sizeof(struct dirent) == 32

struct dir {
  struct dirent entries[512 / sizeof(struct dirent)];
};  // sizeof(dir) == 512, nr_entries == 16
```

文件信息块

由于根目录的目录项已经存储了文件名，所以这里你可以不再存储文件名，只记录数据块的块偏移量，这是直接索引法。

大小一般为一个块的大小，方便修改 bitmap.

参考结构体定义：

```
struct inode {
  uint32_t data_block_offsets[512 / sizeof(uint32_t)];
```

```
};
```

文件数据块

大小为一个块的大小，没有额外信息，512B 全部用来存储来自文件的数据。 由于我们是从根目录、文件信息块一路索引过来的，所以不会发生不知道这个数据块属于谁的问题。

提示

数据结构定义要严格地与磁盘的数据布局保持一致。一个隐性的修改结构体布局的行为就是结构体成员的地址自动对齐。 建议在定义这些结构体时保证它们被 `#pragma pack` 包含，即：

```
// pack
#pragma pack(0)
// definitions
#pragma pack()
// unpack
```

具体信息参考这里

格式化程序

格式化是使用特定文件系统格式对一块磁盘进行初始化的过程。

读取文件是我们的最基本要求，如果你们的游戏使用了外部素材，那么通过文件系进行读取应该是更容易组织的， 所以我们希望能在内核启动时就拥有一个已经按照设计的文件系统格式化的磁盘，所以我们需要额外编写一个格式化程序， 只不过这个格式化程序是带有初始文件数据的。

你可以用任何常见的语言来编写，不过还是推荐用 C, 因为它方便进行精确的位操作和字节操作，不过如何遍历文件目录，可能不是那么直截了当。 但是如果你不准备支持目录，那么将你要格式化的文件准备在同一个目录下，然后 `your-format-prog *`，之后便可以通过 `argc` 和 `argv` 遍历 命令行参数来获得所有文件名了。

建议你在这一环节里彻底忘掉你的内核，专心使用上述的**数据结构**来生成初始的数据磁盘，这个磁盘是个独立的文件，与内核无关。 如果按上面的最简单的参考设计，它作为一个文件的布局大概是这样的：

```
+-----------------+--------+--------+--------+--------+--------+
|  bitmap...      |  dir   |  inode |  data  |  ...   |  ...   |
+-----------------+--------+--------+--------+--------+--------+
```

然后，建议你再写一个配套的读取程序，用于将这个磁盘文件的内容提取出来。你可以先只用文本文件做测试，这样方便即时输出， 当然用二进制文件作为文件系统里根目录下的文件也是可以的，只不过要先保存，然后用外部程序验证是否和原来的一样。 这个程序的读取接口注意做好**抽象**和**封装**，这样它可以很容易地为内核所用（读写磁盘文件可以用 C 的 stdio 的 API, 注意使用 "rb", "rw" 方式打开文件并使用 fread 和 fwrite, stdio 流式处理文件，使用 fseek 和 SEEK_SET 方法获得随机定位的能力）。

以这个文件系统为例，创建文件的流程如下：

1. 找到根目录结构中无效的项，从 bitmap 里分配一个块，更新文件名和偏移量。
2. 在内存中创建一个 inode 结构体，以 512B 为单位循环读取输入文件，分配块，fseek 到那个块的位置写入数据。
3. 将 inode 写到目录项中记录的偏移处（注意块偏移和字节偏移的转换）。
4. 将 bitmap 和目录结构更新到磁盘文件中。

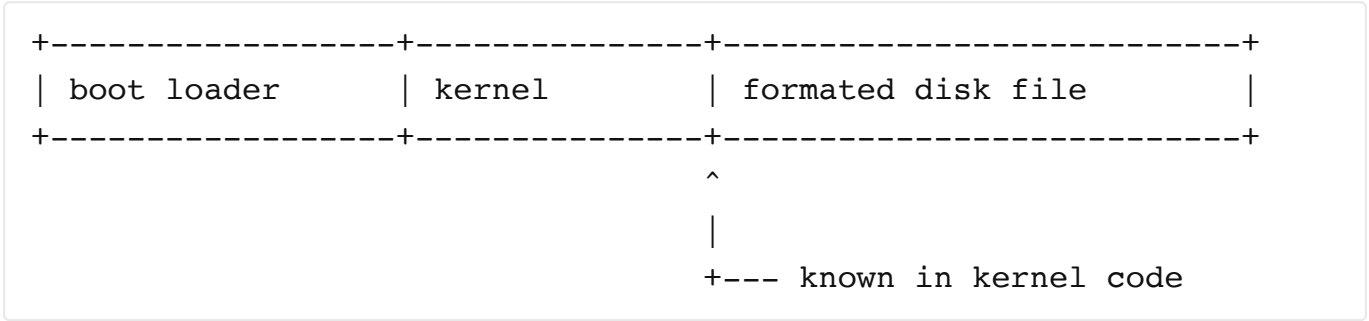
读写流程参考创建文件流程。

内核操作接口

完成格式化程序、验证格式化数据结构的正确性、实现读取函数后，你就可以将上面的成果引入内核。 相关的头文件和读取函数应该能够直接移植到内核代码里。

你需要修改内核初始化过程，在加载用户程序前，先要像之前的读取程序一样，读取 bitmap 和根目录内容，存放在内存中。 只不过这次你不能用 fread 直接读文件，而是使用磁盘的驱动函数，这在 bootloader 里已经使用多次，之后讲义也会提供读写的驱动代码。

关于布局问题，推荐 bootloader + kernel 与数据盘分离，而用户程序则格式化到磁盘中。在之前的实验中， 我们已经将 bootloader 和内核填充为固定的大小，然后再拼接用户程序，这样我们可以从确定的偏移量得到用户程序，现在，这个位置换成数据磁盘， 用户程序编译完后，要执行格式化程序将其格式化到数据盘中，之后将数据盘拼接到固定大小的内核之后，这样做比较简单，只需要记得所有与磁盘相关的偏移量 都要加上这个固定偏移，就像你们之前读取用户程序那样。这样一来 qemu 使用的磁盘布局如下：



用户操作接口

之前已经提到过需要暴露给用户的系统原语以及底层的数据结构。

FCB数组和文件描述符的维护分配与 PCB 如出一辙。PCB 中要增加对文件描述符的记录，固定大小的描述符数组即可（现实系统中一个进程的描述符数量也是有上限的）。

你可以在内核态通过页面映射，将文件的全部内容导入内存，这样方便读取，但是对扩张性的写操作可能不是很友好； 也可以维护一个固定大小的小缓冲区，根据偏移量进行缓冲区的换入和写回（对磁盘）操作。

这些接口最终通过系统调用交由内核完成，对于 open，就是从 FCB 数组里分配一个空闲 FCB 初始化、将对应的下标作为描述符返回并绑定到 PCB 上； 对于 read，就是通过描述符找到 FCB，完成读取后，移动偏移量；对于 write，就是通过描述符找到 FCB，完成写入后，移动偏移量；其他的依次类推。

这一部分的关键是对当前偏移量（以及缓冲区）的维护，并要注意 512B 对齐的问题。

对于删除，即访问内存中的 bitmap 和根目录结构，无效化对应的数据即可。

挑战内容

下面是增强上述最简文件系统的一些方向，需要你发挥自己的创造力来完成：

1. 支持多级目录（在 inode 或者 dirent 里要加 tag 来区分类型了）
2. 多级索引 -> 更大的文件体积
3. 可扩张的磁盘大小（在不修改数据结构重新编译的条件下，提供参数格式化指定大小的磁盘，不是在运行时扩充）

三个方向只要完成一个，即可获得基础得分 10% 加分，累计可获得 30% 加分。

JOS 框架基本涉及上面的提升内容，并且有其他一些设计特色，如果有同学基于 JOS 框架开发并完成与文件系统相关的 Lab5 部分， 则也可以获得 30% 加分（不再考察上面三点）。

磁盘驱动

磁盘驱动来自 JOS, 删除了不必要的文件，但是没有删除不必要的代码。

下载