Ausar的补充指导书----lab5

> 如发现指导书有错漏，请及时联系本人，或课程组内的老师或者助教

Exercise5.1

实验背景介绍

咱们的操作系统终于要到文件读写的部分啦！之前一直都是把程序什么的东西放在内存里，现在咱们要开始读写文件啦。回忆一下我们平时用的windows等操作系统，文件都是存放在磁盘上的对吧？因此，我们实验的第一步，就是去编写读写磁盘的驱动程序。

磁盘属于CPU的外设，一般来说，外设都是采用内存映射的方式挂在CPU上的。也就是说，CPU只要读写特定内存区域，实际上就是在对外设进行交互。如lab1中，我们的printf，实际上最终就是对一个内存地址进行读写，从而实现了在控制台上的输出字符。

既然对外设的控制，实际上是对内存某区域进行读写，所以第一步我们先封装好这个读写外设（内存）的函数吧。

实验内容

> 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

提示

- 要求中所提示的地址，都是物理内存地址。所以我们在代码中编写的时候，要写成对应的内核虚拟地址才能访问。还记得LAB2吗？
- kseg0和kseg1都是简单映射的方式去访问物理内存，但是kseg0是经过了缓存的，这可能导致读写的信息和外设实际上接到的信息不一致或者不同步。因此，我们应该**使用kseg1而非kseg0**
- 访问kseg1,直接把地址加上0xA0000000即可
- 建议采用bcopy函数进行数据复制，用memcpy可能出现一些奇怪的问题（感兴趣可以分析为什么）
- `lib/syscall_all.c`中编写的是系统调用的内核部分
- 还需要在`user/lib.h`中完成用户态下系统调用的声明（函数原型）
- 在`user/syscall_lib.c`中完成用户态下的系统调用实现

Exercise5.2

实验背景介绍

我们已经完成了对外设简单读写函数的设计，但是对IDE磁盘的操作，还需要遵循其厂家设计的交互方式。

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.

在我们的Gxemul模拟器中，IDE磁盘的物理地址为`0x13000000`

而相对这个物理地址不同偏移下的地址有着不同的功能。如上表所示。例如往`0x13000000+0x0000`的地方写入数据，会决定了之后读写数据相对于磁盘的偏移。可以通过写入不同的值来读写磁盘的不同位置。

读写磁盘实际上需要依赖一个512字节的缓冲区，读磁盘的数据会先读到缓冲区内，然后操作系统再读取这个缓冲区。而写入磁盘的时候，也是先写入这个缓冲区内，再让磁盘把缓冲区内数据写入磁盘中。

知道上述消息之后，现在请你完成磁盘读写驱动的编写。

实验内容介绍

> Exercise5.2 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

实验提示

- 无论是读写操作，都是以512字节（0x200）为单位进行的，因此如果要读写的数据比较多，需要分次完成
- 读写操作的时候，读写的是`[secno * 0x200,secno * 0x200 + nsecs * 0x200]`这区域的内容
- 读操作可以分为几个步骤：
  - 选择磁盘号
  - 设置磁盘读取位置的偏移
  - 写入0，代表开始读取
  - 读取磁盘操作结果
    - 0代表失败
    - 其他代表成功
  - 从磁盘缓冲区内读取结果
  - 循环以上步骤直到读取完全部所需要的数据
- 写操作可以分为以下几个步骤
  - 把所需要写入的内容写入缓冲区内
  - 选择磁盘号
  - 设置写入位置的偏移
  - 写入1，代表开始把缓冲区内内容写入磁盘

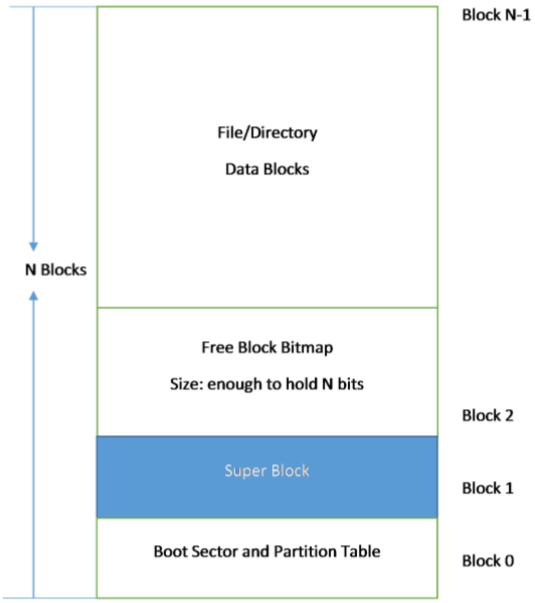
    - 读取写入结果
        - 0代表失败，其他代表成功
    - 继续循环，直到所有数据写入完毕
- 为了防止有些同学手滑打错地址，在这里给出一些大家可能用到的宏定义，可以自行取用

```
#define                   IDE_BEGIN_ADDR 0x13000000
#define IDE_OFFSET_ADDR   IDE_BEGIN_ADDR + 0x0000
#define IDE_OFFSETHI_ADDR IDE_BEGIN_ADDR + 0x0008
#define IDE_ID_ADDR       IDE_BEGIN_ADDR + 0x0010
#define IDE_OP_ADDR       IDE_BEGIN_ADDR + 0x0020
#define IDE_STATUS_ADDR   IDE_BEGIN_ADDR + 0x0030
#define IDE_BUFFER_ADDR   IDE_BEGIN_ADDR + 0x4000
#define IDE_BUFFER_SIZE 0x0200
```

Exercise 5.3

实验背景介绍

我们已经完成了IDE读写驱动的设计。但是为了有效的组织文件，我们需要有文件系统。



上述是磁盘空间的基本布局。

关于超级块，文件系统结构等详细介绍，请参见标准参考书（贼长的那个）或者理论课程。

在我们的这一个练习中，我们重点关注的是`bitmap`的部分。即用来标注一个磁盘块是否被使用。1代表该磁盘块空闲，0代表该磁盘块被使用了。

我们需要完成的是free_block这个函数，用于释放blockno对应的块。其实也就只需要把bitmap对应的位置1即可

实验内容

> 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 ?

实验提示

许多同学第一次在做这个实验的时候，第一步非常困惑如何访问bitmap。

实际上，bitmap已经被映射为了一个指针，所以可以用指针去访问即可。

```
#include "fs.h"
#include <mmu.h>

struct Super *super;

u_int nbitmap;      // number of bitmap blocks
u_int *bitmap;      // bitmap blocks mapped in memory
```

因为bitmap指针是一个u_int型的，即32位。因此需要想想怎么计算`blockno`是在第几个32位中。然后可以把指针当数组用，来找到对应的uint。

计算完成之后，把对应的位数置1，可以尝试使用位运算

以把一个int变量a的第n位置为1为例，可以采用下面的代码

```
a |= (1<<n);
```

Exercise 5.4

实验背景

我们的IDE磁盘，实际上是一个磁盘镜像挂载上去的。想要我们的操作系统能正常管理IDE磁盘中的文件，那么IDE磁盘就必须要与我们的文件系统格式一致，因此我们需要把磁盘镜像中的文件系统格式给整理成我们操作系统兼容的格式。

这一个实验中，我们会用到一个**运行在linux上的程序**，**fs/fsformat.c**在这个程序中，会创建我们操作系统所挂载的fs.img，镜象文件。

在继续下面的内容之前，请**仔细阅读完整指导书上关于文件系统结构的介绍部分**，否则可能完全不知道要做什么。

下面给出了文件的结构

```
struct File {
    u_char f_name[MAXNAMELEN];  // 文件名
    u_int f_size;           // 文件大小，对于文件目录来说，他的f_size代表他的文件索引块所占用的大小
    u_int f_type;           // 是目录文件还是普通文件
    u_int f_direct[NDIRECT];//直接引用块
    u_int f_indirect;//非直接引用块

    struct File *f_dir;     // 仅在fsformat.c有效，在创建文件系统时，用于记录文件所在目录的结构体
    u_char f_pad[BY2FILE - MAXNAMELEN - 4 - 4 - NDIRECT * 4 - 4 - 4];//填充剩余空间，保证结构体大小为BY2FILE
};
/*文件目录的结构大概是这样的：
     * 目录型FIEL：
```

```
     *       --1024个指针，指向文件索引块
     *           --文件索引块里面有FILE2BLK个文件索引。也就是 BY2BLK/(sizeof (struct
File))=16个
     *               --每个索引文件对应着一个文件。存储着文件相关的信息
     *               --索引文件里面也有1024个指针，指向文件具体存储的内容
     */
```

目录型的文件大概是这样子的。

- 一个struct File，作为其索引，里面存放了这个是个目录文件，目录的名字之类的信息，最关键的是里面有10个直接引用指针，还有一个间接引用指针
    - 通过直接引用指针或者间接引用指针link到若干个block，每个block被分成了16个部分，16个部分都可以用来存储文件索引

我们需要完善的是这个函数`struct File *create_file(struct File *dirf)`

其目的是在给定的目录（dirf）中，创建一个新的文件指针。

这个函数做了下面的步骤

- 通过f_size/BY2BLK来计算出，dirf目前link着多少个block
- 如果目前一个block都没有，那么就利用**make_link_block**创建一个新的block，并且返回其首地址指针(其实也就是这个block的16个文件索引中的第一个)

- 如果目前存在着block，那么遍历**最后一块**blcok，查找还有没有空闲的文件索引（也就是文件名为空）
- 如果不能找到空闲的文件索引，那么就创建一个新的block

实验内容介绍

> 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

实验内容提示

- 需要判断一下nblk是否在直接引用块的范围内，如果是的话，要访问直接引用块，否则要访问间接引用块
- 会用到`make_link_block()`这个函数
- 会用到`disk`这个数组
- 一定要理清楚这个文件结构

Exercise 5.5

实验背景介绍

我们虽然已经完成了对IDE读写驱动的设计，但是就和我们平时写C代码时一样。我们希望能在内存中完成对内容的操作，然后再把操作好的内存数据写回文件中。因此，我们需要块缓存的设计。

这样，我们之后的操作流程均是

- 从磁盘中读取内容到内存

- 在内存中进行修改
- 把内存中数据写回磁盘

我们将磁盘中的数据映射到`[DISKMAP,DISKMAX]`对应的内存区间中

因此，我们需要一个函数来找到blockno指向的磁盘块在内存中对应的位置

实验内容介绍

> 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）

实验提示

这个函数非常简单，只需要做两步

- 确认blockno是否在超级块中的`s_nblocks`范围内(超级块可以直接用指针`super`访问)
- 返回DISKMAX加上对应的片移即可
- 一个块的大小，可以用宏`BY2BLK`表示

Exercise 5.6

实验背景介绍

我们访问一个磁盘块的时候，分为几个步骤：

- 找到磁盘块对应的内存地址
- 看看这这磁盘块有没有被映射到内存中
- 如果没有，那么需要map过去
- 操作完成后，ummap这一个内存

所以，我们需要map和unmap相关的函数

实验内容介绍

> 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）。

实验内容提示

map_block函数：

该函数非常简单，主要有两步骤：

- 调用`block_is_mapped`来查看是否已经map过了
- 如果已经map了，那直接返回即可
- 如果没map，那么在对应的地址上调用`syscall_mem_alloc`分配一页空间即可

unmap_block 函数：

- 调用`block_is_mapped`来查看是否已经map过了
    - 如果没map，那直接返回就好
- 用block_is_free，block_is_dirty两个函数来对这一个磁盘块进行检查
    - 如果这一个块不是free的，且dirty，那么需要用write_block写回数据
- 最后用syscall_mem_unmap来释放对应的内存区域

Exercise 5.7

实验背景介绍

在使用文件系统的时候，很重要的一个功能就是查找文件。

我们也需要完成相关的函数。

实验内容介绍

> 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）。

实验内容提示

查找文件分三步：

- 第一步，通过f_size和BY2BLK算出这个目录下有多少个磁盘块
- 遍历目录下所有的block
- 通过file_get_block把这一块磁盘块读到内存中
- 查找这一块中的所有文件，是否存在文件名一致的文件。

Exercise 5.8

实验背景介绍

刚才完成了文件的查找，现在需要对文件进行打开。

此时我们就涉及一个概念Fd，即文件描述符，用于对文件进行管理。

实验内容介绍

> 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）。

实验内容提示

- 首先，我们需要用`fd_alloc`来分配一个新的文件描述符（fd）
- 然后调用`fsipc_open`来打开一个文件，并且绑定到刚刚得到的fd上
- 获取va,ffd,,size,fileid,等信息
- 遍历整个文件大小
    - 用fsipc_map来把对应的文件内容给map到内存中
- 用fd2num返回fd对应的信息

Exercise 5.9

实验背景介绍

一个大文件，每次都整个加载到内存中，这个消耗太大了。因此我们希望能有一个方式，只读写其中的一小部分。

因此，操作系统中设计了read和write两个函数。从fd->fd_offset开始，读写若干字节。

这样子就可以比较灵活的对文件进行读写。

其中，write已经写好，而read格式与write非常像，请你完成一下read函数

实验内容介绍

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

实验内容提示

主要有下面几个步骤：

- 通过fdnum,用`fd_lookup()`查找fd
- 通过`dev_lookup`查找dev（实际上就是找到我们之前弄的IDE读写驱动）
- 检查文件模式是否为只可写，不可读。如果是的话，就报错
- 通过dev->dev_read来进行文件的读取
- 更新fd->fd_offset。因为我们已经继续读取了一段内容，所以偏移也要更新。
- 给buf后面加上`\0`

这些步骤都与write非常相似，大家基本上可以对着write写一写

Exercise5.10

如果每个想读写文件的程序，都要把上面的一大堆函数给编译进去，那么太臃肿了。

因此我们所设计的文件系统相关的操作，最后会封装成一个进程，叫做文件系统服务。

其他程序想要读写文件的时候，就可以通过IPC的形式来与文件系统服务交互，从而完成文件的读写。

因此，我们需要完善相关的工作

实验内容介绍

> Exercise5.10 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

实验内容提示

fsipc_remove：

- 首先，检查一下路径是否为空，或者路径是否超过最大长度（`MAXPATHLEN`），如果是的话,返回`-E_BAD_PATH`
- 建立一个结构体`struct Fsreq_remove`的指针,叫做req，并把指针头指向fsipcbuf
- 把path字符串拷贝到req->req_path中去。
- 用如下命令发送删除指令`fsipc(FSREQ_REMOVE,req,0,0)`

remove:

- 这函数非常简单，就一行，调用fsipc_remove进行path的移除

serve_remove：

- 用user_bcopy把`rq->req_path`给拷贝到path里面
    - 注意，一定要在末尾加`\0`，否则可能出现无限长的字符串现象

- 用file_remove来把path移除

恭喜你，完成了LAB5的所有任务

> 写了好久的指导书，终于可以吃饭去了