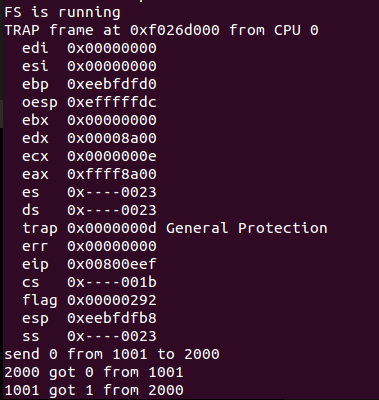
# Lab 5: File system, Spawn and Shell



## File system preliminaries

我们开发的是一个单用户的操作系统，只提供了足够的保护用于发现bug，但是并没有对恶意的用户之间进行隔离。因此我们的文件系统不支持UNIX中文件所有者或者权限这样的概念。同样，我们的文件系统也不支持一般UNIX文件系统中有的例如硬链接，软链接，特殊的设备文件这样的概念。

但是该文件系统能够提供基本的创建、 阅读、 写作、 和删除文件组织分层目录结构。

### On-Disk File System Structure

大多数UNIX文件系统而降磁盘空间氛围两块主要区域：inode区和数据区。

在UNIX文件系统中，每一个文件对应一个inode，每一个文件的inode中包含了关于这个文件重要的信息，比如stat特性，以及指向数据块的指针。

data region被分成很多个data blocks（大概每个8KB），其中存储了文件数据以及目录的元数据。Directory entry中保存了文件名以及指向文件对应的inode的指针。当一个文件的inode有多个directory entry指向它的时候，那我们说这个文件是硬链接的。因为我们的文件系统不会支持硬链接，所以我们不需要做的这么复杂，事实上，我们的文件系统根本不用inode。相反，我们将一个文件的所有元数据都存储在directory entry中。

文件和目录都由一系列的data block组成，它们分布在磁盘的各个地方，就像进程的虚拟地址空间的页面分布在物理内存当中一样。文件系统隐藏了block分布的细节，只提供了在文件任意位置进行读写的接口。文件系统处理了所有的对于目录的修改。我们的文件系统允许用户进程直接读取目录的元数据，因此用户进程能直接对目录进行扫描工作，而不需要对于文件系统进行额外的调用。但是这样做的缺陷是，这会让应用程序依赖与目录元数据的格式，从而很难在不改变，至少不重新编译应用程序的情况下，对文件系统的内部分布进行改变。

### Sectors and Blocks

block size ： 4096B = PGSIZE，操作系统使用磁盘层面

Sectors size ： 512B 磁盘的硬件属性

大多数磁盘都不支持以字节的粒度读写文件，而是以sector作为单位进行读写，通常一个sector为512字节。文件系统通常用block来申请和使用磁盘。需要进行区分的是，sector size是磁盘的特性，而block size是操作系统使用磁盘的一个概念。一个文件系统的block size必须是底层硬盘的sector size的整数倍。

在UNIX xv6中使用的block size是512字节，和底层磁盘的sector size相同。但是，许多现代的操作系统趋向于使用更大的block size，因为存储空间已经变得越来越便宜，而从更大的粒度对存储空间进行管理可以更加高效。因此我们的文件系统使用4096字节作为block size，刚好和处理器的页面大小相同

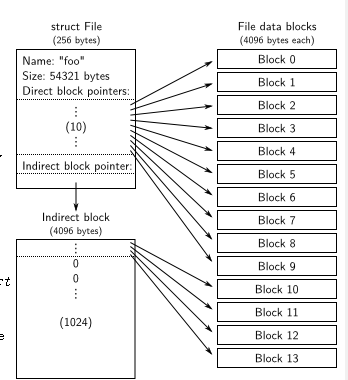
### Superblocks

文件系统通常会预留一些非常容易找到的block在磁盘中（例如磁盘的最前面或者最后面），用来存储一些用于描述整个文件系统的元数据，例如block size， 磁盘大小，找到根目录的所需的元数据，文件系统挂载的时间，文件系统最后一次进行错误检测的时间等等。而这些块统称为superblocks。

在我们的文件系统中只有一个superblock，并且它总是放在Block 1的位置。它的分布就是inc/fs.h中定义的struct Super的样子。Block 0通常被用于保存boot loader和partition tables，所以文件系统通常来说不会使用磁盘开头的这些blocks。许多真实的文件系统会保存多个superblock，重复地分布在磁盘的各个区域，因此当一个superblock损坏的时候，我们仍然能通过其他superblock访问文件系统。

#### 

### File Meta-data



描述文件的元数据的分布和inc/fs.h中的struct File一致。元数据包括文件名，大小，类型（普通文件或目录），以及指向block的指针。因为Jos没有inode，所以这些元数据都存放在磁盘的directory entry中。和真实的文件系统不同，为了简单起见，我们同时在磁盘和内存中用File表示文件的元数据。

在struct File中，数组f\_direct中存储了文件中前十个block的编号，我们称它为文件的direct blocks。当文件小于10\*4096=40KB的时候，这些block就够用了。当文件更大的时候，我们需要一个地方去保存文件剩下的block number。对于任何大于40KB的文件，我们需要另外分配一个disk block，叫做indirect block，用于存储其他剩余的4096/4=1024个block number。因此，我们的文件系统支持最多1034个block，大概为4M左右。为了支持更大的文件，真实的文件系统通常需要支持两级或者三级的indirect block。

### Directories versus Regular Files

File结构在我们的文件系统中既可以用来表示普通的文件，也可以用来表示目录。这两种类型的文件是通过file中的type字段来区分的。文件系统通常采用同样的方式对普通文件和目录进行管理，不同的是，它通常不会对普通文件的data blocks进行任何解析。相反，文件系统会将目录文件的内容解析为一系列的File结构用于描述文件和该目录的子目录。

我们文件系统的superblock的File文件中包含了文件系统根目录的元数据（struct Super中的root字段）。目录文件的内容就是一系列的用于描述文件和根目录的子目录的File结构。而根目录的子目录则反过来包含了更多用于代表子目录的File结构。

## The File System

这个实验的目的并不是为了实现整个操作系统，只要实现其中的关键组件即可。因此，我们需要实现的是(1)将block读入block cache并且将它们刷新回磁盘，(2)将file offset映射到磁盘的block中，(3)实现read，write，和open这些IPC interface。

### Disk Access

jos系统中的file system environment需要能够访问硬盘，但是我们的内核还并没有实现任何磁盘访问的功能。因此我们实现了一个IDE磁盘驱动作为user-level file system environment的一部分，而不是像传统的monolithic操作系统那样，采取往内核中添加一个IDE磁盘驱动，然后配合以必要的系统调用，从而能够让文件系统对它进行访问的方法。不过我们仍然需要对内核做一定的修改，从而使得file system environment有足够的权限能够自己实现对于磁盘的访问

当我们使用"programmed IO"（PIO）-based disk access而不使用disk interrupt的时候，我们就能比较容易地实现用户空间的磁盘访问。我们同样可以在用户空间实现interrupt-driven device drivers，但是这会比较困难，因为内核需要接收设备中断并且将它们分发到正确的用户空间进程。

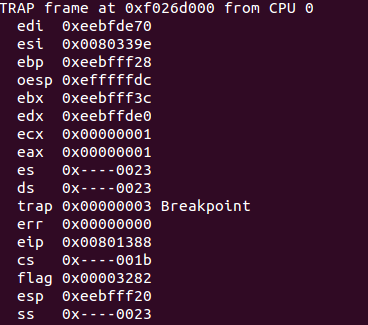
x86处理器通常使用EFLAGS处理器的IOPL位来决定是否允许处于保护模式状态下的代码是否能使用特殊的设备IO指令，例如IN和OUT。因为IDE硬盘所有的寄存器都在x86的IO空间中，而不是memory-mapped的，因此为了让文件系统访问这些寄存器，我们只需要将“IO privilege”赋予file system environment即可。事实上，EFLAGS中的IOPL位为内核提供了一种简单的“all-or-nothing”的方法决定用户空间的代码是否能够访问IO space。在我们的系统中，我们希望只有file system environment能够访问IO space。

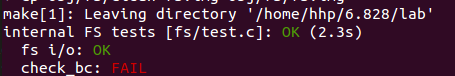
#### Exercise 1.

i386\_init identifies the file system environment by passing the

type ENV\_TYPE\_FS to your environment creation function, env\_create.

Modify env\_create in env.c, so that it gives the file system environment I/O privilege, but never gives that privilege to any other environment.





Question

1. Do you have to do anything else to ensure that this I/O privilege setting is saved and restored properly when you subsequently switch from one environment to another? Why?

### The Block Cache

在我们的文件系统中，我们基于处理器的虚拟存储系统实现了一个简单的"buffer cache"（真的仅仅只是一个block cache）。

我们的文件系统将只能处理3GB以内的磁盘。我们会在file system environment地址空间中预留3GB的空间，范围是0x10000000（DISKMAP）到0xD0000000（DISKMAP+DISKMAX），作为磁盘的"memory mapped"。比如，磁盘的block 0映射到虚拟地址0x10000000，block 1映射到虚拟地址0x10001000等等。fs/bc.c的diskaddr函数实现了磁盘的block numbers到虚拟地址的映射。

因为我们的file system environment有它自己的虚拟地址空间，而它所需要做的唯一的事就是读取文件，因此将它大量的地址空间预留用作以上用途是合理的。但是如果要在真正的文件系统中，如果在32位的机器上，像这样实现文件的读取是不太可能的，因为磁盘的容量远远大于3GB。而如果是64位机，那么这样的buffer cache管理方法依旧是可行的。

当然，将整个硬盘读进内存是不合理的，所以我们实现了一种demand paging的方式，我们只为disk map region内的页分配页面，其他页面则在发生page fault的时候再读入相应的block。这样，我们就感觉好像整个磁盘都在内存中。

fs/fs.c中的fs\_init是如何使用block cache的一个主要的示例。在初始化block cache之后，全局变量super中存放了指向disk map region的指针。之后，我们可以直接从super结构中读取，好像它们已经在内存中一样，而我们的page fault handler会将它们从磁盘中读入，如果有必要的话。

#### Exercise 2.

Implement the bc\_pgfault and flush\_block functions

in fs/bc.c. bc\_pgfault is a page fault handler, just like the one your wrote in the previous lab for copy-on-write fork, except that its job is to load pages in from the disk in response to a page fault. When writing this, keep in mind that (1) addr may not be aligned to a block boundary and

(2) ide\_read operates in sectors, not blocks.

The flush\_block function should write a block out to disk if

necessary. flush\_block shouldn't do anything if the block isn't even in the block cache (that is, the page isn't mapped) or if it's not dirty. We will use the VM hardware to keep track of whether a disk block has been modified since it was last read from or written to disk. To see whether a block needs writing, we can just look to see if the PTE\_D "dirty" bit is set in

the uvpt entry

### The Block Bitmap

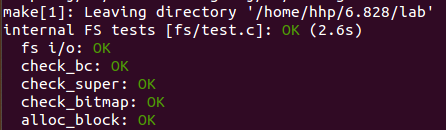
在fs\_init设置了bitmap指针以后，我们可以把bitmap看作一个位图，每一个元素代表磁盘中的一个block。例如，block\_is\_free这个函数能简单地检测指定的block是否在bitmap中被标为free。

#### Exercise 3.

Use free\_block as a model to implement alloc\_block, which should find a free disk block in the bitmap, mark it used, and return the number of that block. When you allocate a block, you should immediately flush the changed bitmap block to disk with flush\_block, to help file system consistency.

Bitmap是一个压缩的表示block是否被占用的数组，在2号block处，每一位代表一个block，为1代表free。相应的查询方式为bitmap[blockno/32] & 1<<(blockno%32)。





### File Operations

在fs/fs.c中我们提供了大量的函数用于实现例如解析和管理File结构，扫描和管理整个目录文件，以及从根目录访问到一个绝对路径的操作

#### Exercise 4.

Implement file\_block\_walk and file\_get\_block. file\_block\_walk maps

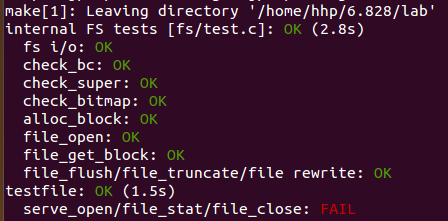
from a block offset within a file to the pointer for that block in the struct

File or the indirect block, very much like what pgdir\_walk did for page

tables. file\_get\_block goes one step further and maps to the actual disk block, allocating a new one if necessary.

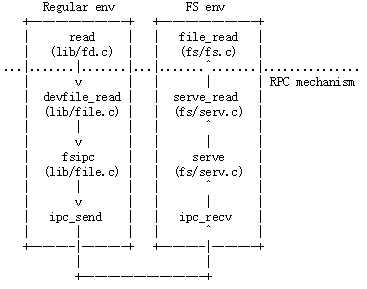
函数file\_block\_walk得到相应block号入口号的指针

函数file\_get\_block得到相应block首地址



### The file system interface

现在file system environment自身的功能已经具备了，接下来要做的就是让其他想要使用文件系统的environment能获取这些功能。因为其他environment不能直接调用file system environment中的函数，所以我们需要通过remote procedure call（PRC）把file system environment把访问权限暴露出去，利用JOS之上的IPC机制。调用过程如下图所示：



虚线以下的所有工作都是为了让file system environment从普通的environment中获取读请求。首先，read作用在一个file descriptor上，然后被简单地分发到合适的device read function，在这个例子中是devfile\_read(我们可以有更多的设备类型，例如管道)。devfile\_read是专门用来读磁盘文件的。在lib/file.c中的所有devfile\_\*函数都用来实现FS操作的客户端的，它们的工作方式基本相同，都是将参数打包进一个request structure，调用fsipc去传递一个IPC请求，最后再解压并返回结果。fsipc函数简单地处理一些发送请求共同的细节，然后获取请求的结果。

文件系统server端的代码可以在fs/serv.c中找到。它就是在serve函数中无限循环，不断地通过IPC接收请求，将请求分发到相应的处理函数，并且通过IPC将结果返回。在read中，serve会分发到serve\_read函数，它会负责一些具体的工作，例如解压request structure以及最终调用file\_read函数实现读文件的操作。

在JOS中IPC操作，让一个environment发送一个32位的数，还可以选择共享一个页面。在从client端向客户端发送请求时，我们使用一个32位的数字作为一个request type，并且将带有请求参数的union Fsipc放在通过IPC共享的页面上。在client端，我们总是在fsipcbuf共享页面，而在server端，我们将我们将到来的请求页映射在fsreq(0x0ffff000)。

server同样需要通过IPC返回response。我们用一个32位的数字表示函数的返回值。对于大多数的RPC，这就是它们返回的所有的东西了。不过FSREQ\_READ和FSREQ\_STAT还会返回数据，其实它们做的也仅仅只是写client发过来的页面。在返回的response IPC中，我们不用对页面进行传送，因为client和file system server在一开始就对它进行了共享。不过，对于FSREQ\_OPEN，server会和client共享一个新的"Fd page"。

#### Exercise 5

Implement serve\_read in fs/serv.c.

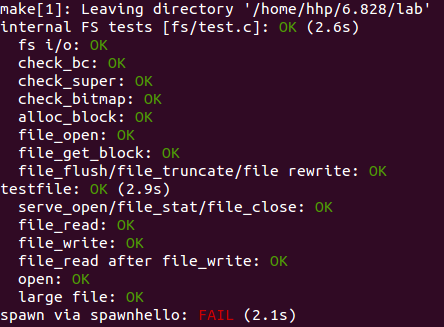
#### Exercise 6

Implement serve\_write in fs/serv.c and devfile\_write in lib/file.c.





这一句话的差别导致错误



## Spawning Processes

在lib/spawn.c中的spawn函数能够创建一个新的environment，从文件系统中加载程序镜像，接着让child environment自己运行这个程序。之后，父进程再独立于子进程单独运行。spawn函数很像UNIX中执行了一个fork，然后马上在子进程中调用exec。

我们并不是通过UNIX风格的exec来实现spawn的，因为在微内核中，在用户空间实现spawn更加容易

#### Exercise 7.

实现文件 kernel/syscall.c中的sys\_env\_set\_trapframe系统功能调用，为新创建的进程初始化状态。



### Sharing library state across fork and spawn

在UNIX中file descriptor是一个一般的概念，其中包含了pipes，console IO等等。每一种设备的类型都有一个对应的struct Dev，其中包含了指向对应设备类型的read/write等函数的指针。lib/fd.c在这之上实现了类UNIX的file descriptor接口。每个struct Fd中都包含了对应的设备类型，而lib/fd.c中的大多数函数都只是简单地将操作分发到struct Dev中的各个函数。

lib/fd.c同时在每个application environment的地址空间中维护了一个file descriptor table，从FSTABLE开始。这个表有一个页的大小，其中存放了一个应用一次最多能打开的MAXFD(现在是32)个file descriptor。在任意时间，一个特定的file descriptor table当且仅当对应的file descriptor在使用的时候才会被映射。每个file descriptor还有一个可选的"data page"，对应的区域从FILEDATA开始，设备可以选择使用它。

我们希望在fork和spawn之后能共享file descriptor state，但是现在file descriptor是保存在用户空间的内存中的。现在，在fork以后，内存会被标记为copy-on-write，所以状态是被拷贝的而不是共享的。（这意味着environment不能seek那些不是它自己打开的文件，pipe也不能在fork之后使用）。在spawn中，内存根本就不会被拷贝。（事实上，spawn得到的environment根本就没有打开的file descriptor）。

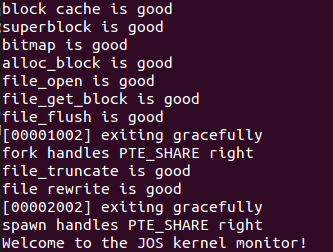
我们现在要改变fork的实现，从而能让特定区域的内存被"library operating system"使用，从而总是能被共享。我们会利用起page table entry中那些未被使用的位（就像PTE\_COW一样），而不是对于某些区域直接做hard-code。

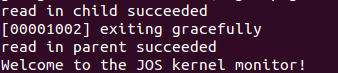
我们已经在inc/lib.h中定义了PTE\_SHARE。这个标志位是在AMD和Intel手册中三个被标记为"available for software use"的PTE标记位。如果一个page table entry的该位被标记了，那么该PTE将直接从父进程拷贝到子进程，不论是在fork还是在spawn中。

#### Exercise 8.

Change duppage in lib/fork.c to follow the new convention. If the page table entry has the PTE\_SHARE bit set, just copy the mapping directly. (You should use PTE\_SYSCALL, not 0xfff, to mask out the relevant bits from the page table entry. 0xfff picks up the accessed and dirty bits as well.)

Likewise, implement copy\_shared\_pages in lib/spawn.c. It should loop through all page table entries in the current process (just like fork did), copying any page mappings that have the PTE\_SHARE bit set into the child process.





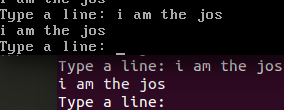
## The keyboard interface

为了让shell能够工作，我们需要能够往里面输送信息。QEMU已经能够将我们输入的东西输出到CGA display和serial port中了，但是我们现在只能在kernel monitor中进行输入。在QEMU中，键盘的输入被显示在图形窗口中，同时，console中的输入作为serial port中字符的显示。kern/console.c已经包含了键盘和serial 的驱动，这我们从lab1就开始使用了，但是现在我们需要将它们和系统的剩余部分连接起来。

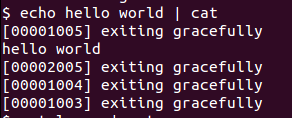
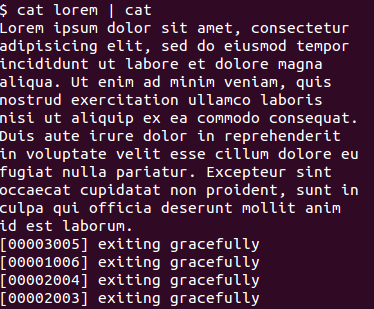
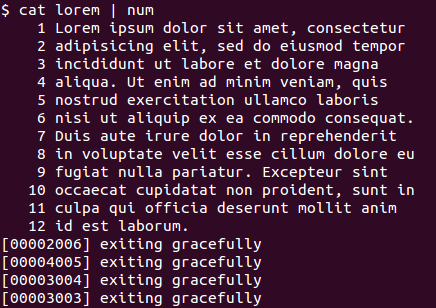
#### Exercise 9

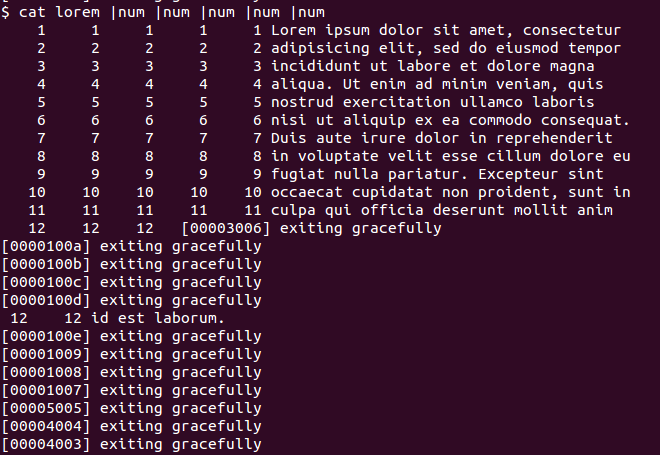
In your kern/trap.c, call kbd\_intr to handle trap IRQ\_OFFSET+IRQ\_KBD and serial\_intr to handle trap IRQ\_OFFSET+IRQ\_SERIAL

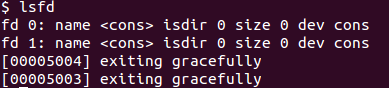
在lib/console.c中，我们已经实现了console的输入输出的file type。kbd\_intr和serial\_intr会根据最近的输入填充buffer，而console file type则会消耗buffer。(console file type通常作为默认的stdin/stdout，除非用户重定向它们)



## The Shell

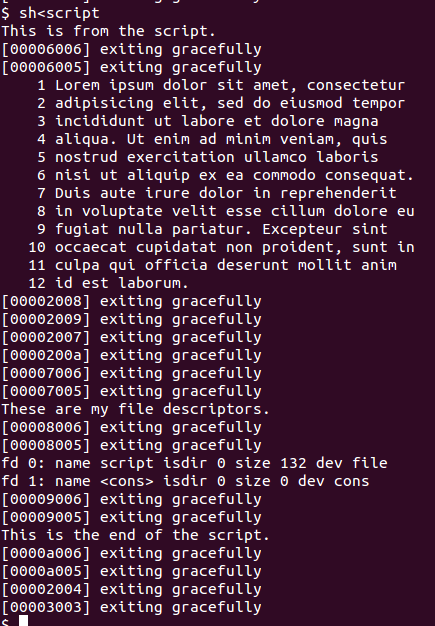




#### Exercise 10.

The shell doesn't support I/O redirection. It would be nice to run sh <script instead of having to type in all the commands in the script by hand, as you did above. Add I/O redirection for < to user/sh.c.

Test your implementation by typing sh <script into your shell



控制流：   