Lab 5: File system, Spawn and Shell

Introduction

在这个lab，你将实现 `spawn` ，一个加载和运行磁盘上可执行文件的库调用。然后你将充实你的内核和库操作系统，足以在控制台运行一个shell。这些特性需要一个文件系统，所以这个lab引入了一个简单的*read/write file system* .

Getting Started

Use Git to fetch the latest version of the course repository, and then create a local branch called lab5 based on our lab5 branch, origin/lab5:

```
add git
git pull      # 这里若出错: server certificate verification failed.
CAfile: /etc/ssl/certs/ca-certificates.crt CRLfile: none
              # 需要导入环境变量 ·export GIT_SSL_NO_VERIFY=1·
# Already up-to-date
git checkout -b lab5 origin/lab5
# Branch lab5 set up to track remote branch refs/remotes/origin/lab5
# Switched to a new brach "lab5"
git merge lab4
```

这个lab的这个部分的主要部分是*文件系统环境* ，位于新的 `fs` 目录。扫描一下这个目录中的所有文件来感受一下什么是新的。当然，在 `user` 和 `lib` 目录中也有一些新的系统相关的文件。

fs/fs.c	**Code that mainipulates the file system's on-disk structure.**
fs/bc.c	A simple block cache built on top of our user-level page fault handling facility.
fs/ide.c	Minimal PIO-based (non-interrupt-driven) IDE driver code.
fs/serv.c	The file system server that interacts with client environments using file system IPCs.
lib/fd.c	Code that implements the general UNIX-like file descriptor interface.
lib/file.c	The driver for on-disk file type, implemented as a file system IPC client.
lib/console.c	The driver for console input/output file type.
lib/spawn.c	Code skeleton of the spawn library call.

你应该再次运行`pingpong`，`primes`，`forktree`测试例子，在合并完新的lab5代码之后。You will need to comment out the ENV_CREATE(fs_fs) line in `kern/init.c` because `fs/fs.c` tries to do some I/O, which JOS does not allow yet. Similarly, temporarily comment out the call to `close_all()` in `lib/exit.c`; this function calls subroutines that you will implement later in the lab, and therefore will panic if called. If your lab 4 code doesn't contain any bugs, the test cases should run fine. Don't proceed until they work. Don't forget to un-comment these lines when you start Exercise 1.

如果不能正常工作，使用`git diff lab4`来查看所有修改，确保没有任何lab4的代码在lab5中丢失。确保lab4仍然工作。

Lab Requirements

As before, you will need to do all of the regular exercises described in the lab and at least one challenge problem. Additionally, you will need to write up brief answers to the questions posed in the lab and a short (e.g., one or two paragraph) description of what you did to solve your chosen challenge problem. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called `answers-lab5.txt` in the top level of your *lab5* directory before handing in your work.

File system preliminaries

你将进行工作的文件系统比大多数的“真实”的文件系统更加简单，包括 xv6 UNIX 中的文件系统，但是它足够强大来提供基础的特性：*创建*，*读取*，*写入*，*删除* 组织在一个结构化目录结构的文件。

我们正在开发只有 *单个用户* 的操作系统，提供足够的保护来捕捉漏洞，*但是* 不保护多个来自互相的相互怀疑的用户。因此，我们的文件系统不支持UNIX中文件所有者和权限的想法。我们文件系统同时也不支持 *硬连接*，*符号连接*，*时间戳* 或是像大多数UNIX文件系统中特殊的设备文件。

On-Disk File System Structure

大多数的UNIX文件系统将可用的磁盘空间分为两个主要类型：***inode regions*** 和 ***data regions***。UNIX文件系统将一个 *inode* 分配给文件系统中的每个文件；一个文件的 *inode* 有着该文件关键的 *metadata*，比如其 *统计属性* 和 *指向其数据块的指针*。*data regions* 被分到更大的（一般为8KB或更大）数据块，文件系统在该数据块存储 *文件数据* 和 *目录元数据*。目录条目包含文件名和指向 *inode* 的指针；如果文件系统中多个目录条目都指向一个文件的 *inode*，则该文件被称为 *硬连接*。

因为我们的文件系统不支持硬连接，我们不需要这种级别的间接，因此可以做一个更便捷的简化：我们的文件系统将**不使用inode**，而是简单地在 *目录条目* 中存储所有文件的元数据，来描述文件。

文件 和 *目录* 在逻辑上都由一系列数据块组成，这些数据块可能散落在整个磁盘上，就像环境的虚拟地址空间pages可以散落在整个物理内存。*文件系统环境* 隐藏了块分布的细节，提供了 *读取* 和 *写入* 的接口，这两个操作可以读写文件中任意偏移的一序列字节。*文件系统环境* 内部处理所有对目录的修改，作为执行比如 *文件创建* 和 *文件删除* 动作的一部分。

我们的文件系统不允许用户环境直接读取目录元数据，直接读取意味着 *用户环境* 可以自己执行 *目录扫描* 操作（比如，实现 `ls` 程序），而不是必须依赖于另外的对文件系统的特殊调用。这种文件扫描的**缺点**，以及大多数UNIX变体都不支持的原因，就是它使得应用程序 *依赖* 于 *目录元数据* 的格式，让不修改或重新编译应用程序就能修改 *文件系统的内部布局* 变得很困难。

Sectors and Blocks

大多数磁盘不能以字节为单位读写，而是以扇区为单位读写。在 `JOS` 中，每个扇区为512字节。文件系统实际上使用 *块* 来分配和使用磁盘存储。

要区分两个名词： *扇区大小* 是磁盘硬件的属性；而 *块大小* 是操作系统使用磁盘的一个方面。一个文件系统的 *块大小* 必须是底层磁盘扇区大小的倍数。

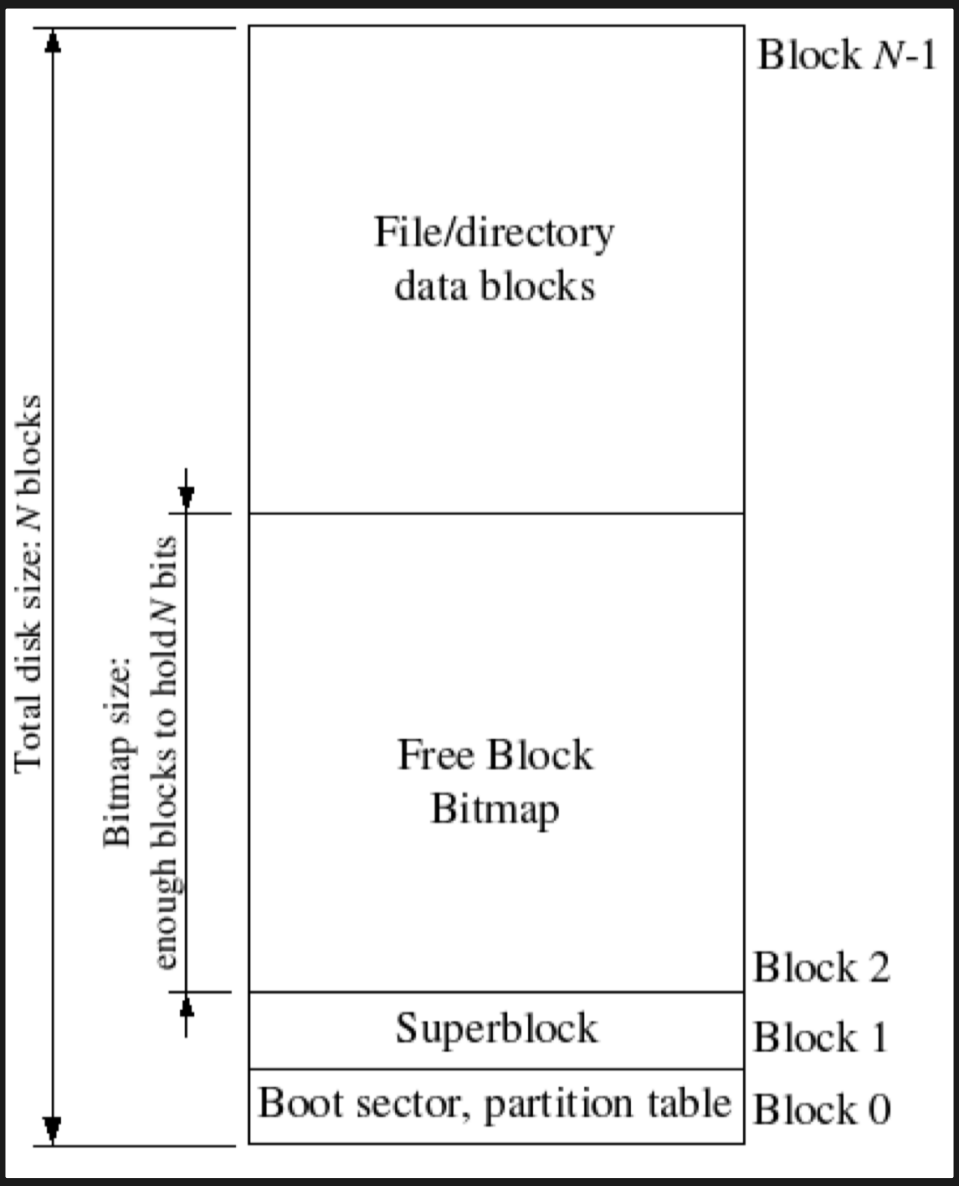
UNIX xv6 文件系统的 *块大小* 为512字节，和底层磁盘的扇区大小相同。大多数的现代文件系统使用更大的 *块大小* ，因为存储空间越来越便宜并且用更大粒度管理存储也更加高效。我们的文件系统使用大小为4096字节的块，方便和处理器的页大小匹配。

Superblocks

文件系统一般会在“容易找到”的位置保存特定的磁盘块（比如最开始和最后），来存储描述整个文件系统的元数据，比如 *块大小* ， *磁盘大小* 。任何元数据需要用来查找 *根目录* ， *文件系统最后被挂载的时间* ， *文件系统最后被检查错误的时间* 等等。这些特殊的块被称为 *superblocks* 。

我们的文件系统有一个superblock，总位于磁盘上的 *块1* 。它的布局在 `inc/fs.h` 中的 `struct Super` 中定义。 *块0* 通常保留用于存储 *boot loaders* 和 *partition tables* ，所以文件系统一般不使用第一个磁盘块。

很多“真实的”文件系统保存多个superblocks，在磁盘的几个广泛分布的区域中复制，所以如果其中一个superblock被损坏或者磁盘在该区域发出一个 *media error* ，其它的 *superblock* 仍可以被找到并用于访问文件系统。

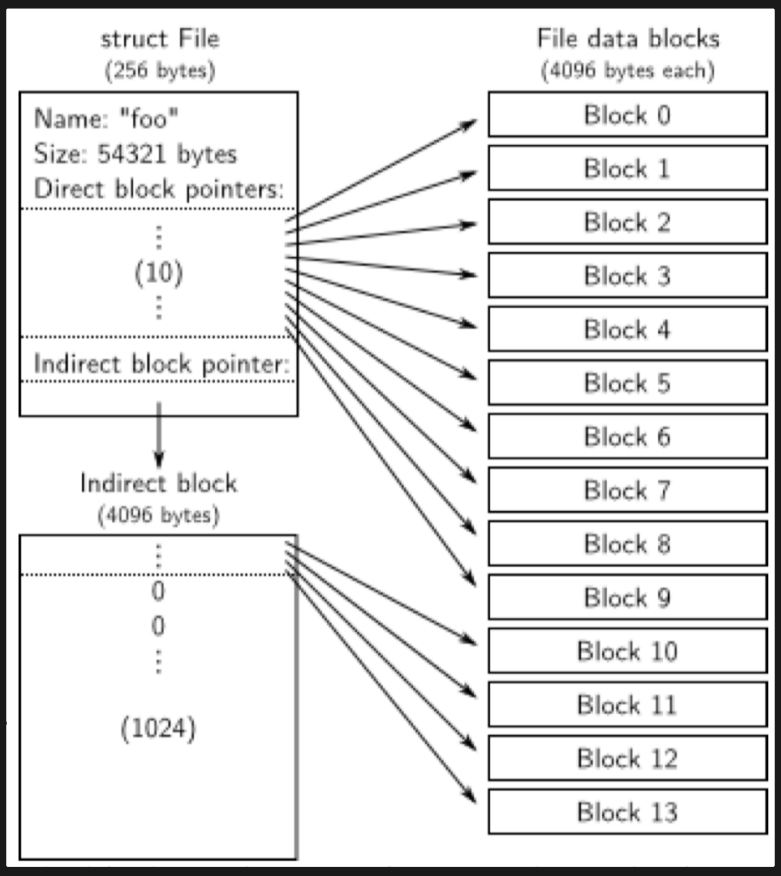


File Meta-data

描述一个文件的*meta-data*的布局在`inc/fs.h`的`struct File`中描述。这个*meta-data*包括文件的名字，大小，类型（一般是文件或是目录），和指向组成文件块的指针。就像上面所提到的，我们不使用*inodes*，所以这个*meta-data*被存储于磁盘上的目录条目。不像大多数"真实的"的文件系统，为了简单我们使用这一个`File`结构来表示文件*meta-data*，因为它出现在磁盘和内存中。

`struct File`中的`f_direct`包含空间来存储文件的前10个块，我们称之为文件的***直接块***。对于小文件，大小最大为104096=40KB的文件，这意味着文件所有块的块号将直接填入`File`结构；对于更大的文件，我们需要一个空间来保存剩下的文件块号。因此，任何文件大小大于40KB的文件，我们分配一个额外的磁盘块，称为文件的**间接块**，来保存最大到4096/4=1024个块数。

因此，我们的文件系统允许文件最多占用到1034个块，或刚超过4MB。为了支持更大的文件，"真实"文件系统一般支持*double- and triple-indirect blocks*。



Directories versus Regular Files

我们文件系统中的 `File` 结构可以表示一个 *常规文件* 或是一个 *目录* ； 这两个“文件”的类型通过 `File` 结构中的 `type` 字段来进行区分。文件系统以相同的方式管理 *常规文件* 和 *目录文件* ， 但是有一点不同：它根本不解释和 *常规文件* 关联的数据块内容，但是文件系统将 *目录文件* 中的内容解释为一系列 `File` 结构，这些结构描述在该目录中的文件和子目录。

我们文件系统中的superblock包含一个 `File` 结构（ `struct Super` 的 `root` 字段）保存文件系统 *根目录* 的元数据。该目录文件的内容是一系列 `File` 结构，描述了位于文件系统 *根目录的文件和目录* 。任何在根目录的 *子目录* 可能也会包含更多的 `File` 结构表示 *子目录* 等。

The File System

这个lab的目标不是让你实现整个文件系统，而是需要你实现特定的关键部分。特别的，你将负责将块读入 *block cache* ， 并将它们刷新回磁盘；分配磁盘块；映射文件偏移到磁盘块；实现在IPC接口的读取，写入，打开操作。因为你不用实现整个操作系统，熟悉所提供的代码和各种文件系统接口很重要。

Disk Access

我们的操作系统中的文件系统环境需要访问磁盘，但是我们没有在内核中实现任何磁盘访问功能。不采用传统的“整体”操作系统策略，即向内核加入一个 *IDE磁盘驱动器* 和必须的系统调用来允许系统来访问它，而是将 *IDE磁盘驱动器* 实现为 *用户级文件系统环境* 的一部分。我们将需要稍微修改内核，来修改一些东西使得文件系统环境有实现磁盘访问所需要的权限。

只要我们依赖于轮询，基于“编程I/O”（PIO）的磁盘访问并且不使用磁盘中断，在用户空间实现磁盘访问很简单。在用户模式实现 *中断驱动的设备驱动* 也是可能的，但是会更困难，因为内核必须现场中断设备，并将其分配到正确的 *用户模式环境* 。

x86处理器使用 `EFLAGS` 寄存器中的 `IOPL` 来决定 *保护模式* 代码是否允许执行特殊的设备I/O指令，比如 `IN` 和 `OUT` 指令。因为我们需要访问的所有IDE磁盘寄存器都位于x86的I/O空间，而不是被 *内存映射* ，所以我们**唯一**要做的就是赋予 *文件系统环境* "I/O privilege"来允许文件系统访问这些寄存器。实际上， `EFLAGS` 寄存器中的 `IOPL` 位提供给内核一个简单的“all-or-nothing"方法，来控制 *用户模式* 代码 *能否访问I/O空间* 。

在我们的例子中，我们想要 *文件系统环境* 能访问I/O空间，但是我们根本**不想要任何其它环境**能访问I/O空间。

> **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.
>
> Make sure you can start the file environment without causing a General Protection fault. You should pass the `"fs i/o"` test in make grade.

```
@@ -464,6 +464,9 @@ env_create(uint8_t *binary, enum EnvType type)
  }
  load_icode(e, binary);
  e->env_type = type;
+       if(ENV_TYPE_FS == type){
+               e->env_tf.tf_eflags |= FL_IOPL_MASK;
+       }
 }
```

Note that the GNUmakefile file in this lab sets up QEMU to use the file `obj/kern/kernel.img` as the image for disk 0 (typically "Drive C" under DOS/Windows) as before, and to use the (new) file `obj/fs/fs.img` as the image for disk 1 ("Drive D"). In this lab our file system should only ever touch disk 1; disk 0 is used only to boot the kernel. If you manage to corrupt either disk image in some way, you can reset both of them to their original, "pristine" versions simply by typing:

```
rm obj/kern/kernel.img obj/fs/fs.img
make
```

```
make clean
make
```

The Block Cache

在我们的文件系统中，我们将实现一个简单的"buffer cache"(really just a block cache)，在处理器虚拟内存系统的帮助下。针对block cache的代码在 `fs/bc.c` 中。

我们的文件系统被限制只能处理3GB或更少的磁盘大小。我们给文件系统环境地址空间保存一个大的固定的3GB的区域，从 *0x10000000(DISKMAP)* 到 *0xD0000000(DISKMAP + DISKMAX)* ，作为磁盘的一个“内存映射”版本。举个例子，磁盘块0被映射于虚拟地址 *0x10000000* ,磁盘块1被映射于 *0x10001000* ,以此类推。 `fs/bc.c` 中的 `diskaddr` 实现了从磁盘块号到虚拟地址的转换（和一些正常的检查）。

`diskaddr()` in `fs/bc.c`

```
// Return the virtual address of this disk block.
void*
diskaddr(uint32_t blockno)
{
  if (blockno == 0 || (super && blockno >= super->s_nblocks))
    panic("bad block number %08x in diskaddr", blockno);
  return (char*) (DISKMAP + blockno * BLKSIZE);
}
```

因为我们的文件系统环境有其自己的虚拟地址空间，独立于系统中其它环境的的虚拟地址空间。文件系统环境唯一要做的就是实现文件访问，用这种方式保留大多数的文件系统环境地址空间是合理的。

对于一个“真实的”文件系统在32-bit的机器上来做这个是 *不好处理* 的，因为现代磁盘大小都要比3GB更大。但是一个 *buffer cache* 管理方法在一个64-bit地址空间机器上可能仍是合理的。

当然，将整个磁盘读入内存会耗费很长时间，所以我们将实现一种形式的 `demand paging` ，这样我们只要在磁盘映射区域分配页，然后从磁盘读取对应块，来回应在该区域的 `page fault` 。用这种方式，我们可以假装整个磁盘都在内存中。

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 `PTE_D` bit is set by the processor in response to a write to that page; see 5.2.4.3 in chapter 5 of the 386 reference manual.) After writing the block to disk, `flush_block` should clear the `PTE_D` bit using `sys_page_map` .

Use `make grade` to test your code. Your code should pass "check_bc", "check_super", and "check_bitmap".

`bc_pgfault` in `fs/bc.c`

```
// Fault any disk block that is read in to memory by
// loading it from disk.
static void
bc_pgfault(struct UTrapframe *utf)
{
  void *addr = (void *) utf->utf_fault_va;
  uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
```

```
  int r;

  // Check that the fault was within the block cache region
  if (addr < (void*)DISKMAP || addr ≥ (void*)(DISKMAP +
DISKSIZE))
    panic("page fault in FS: eip %08x, va %08x, err %04x",
          utf→utf_eip, addr, utf→utf_err);

  // Sanity check the block number.
  if (super && blockno ≥ super→s_nblocks)
    panic("reading non-existent block %08x\n", blockno);

  // Allocate a page in the disk map region, read the contents
  // of the block from the disk into that page.
  // Hint: first round addr to page boundary. fs/ide.c has code to
read
  // the disk.
  //
  // LAB 5: you code here:
// void* start = (void*)ROUNDDOWN((uint32_t)addr, PGSIZE);
  addr = (void*)ROUNDDOWN((uint32_t)addr, PGSIZE);
  if((r = sys_page_alloc(0, addr, PTE_P | PTE_W | PTE_U)) ≠ 0){
    panic("bc_pgfault: %e", r);
  }
  if((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) ≠ 0){
    panic("bc_pgfault: ide_read, %e", r);
  }

  // Clear the dirty bit for the disk block page since we just
read the
  // block from disk
  if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
PTE_SYSCALL)) < 0)
    panic("in bc_pgfault, sys_page_map: %e", r);

  // Check that the block we read was allocated. (exercise for
  // the reader: why do we do this *after* reading the block
  // in?)
  if (bitmap && block_is_free(blockno))
    panic("reading free block %08x\n", blockno);
```

```
}
```

`flush_block()` in `fs/bc.c`

```
// Flush the contents of the block containing VA out to disk if
// necessary, then clear the PTE_D bit using sys_page_map.
// If the block is not in the block cache or is not dirty, does
// nothing.
// Hint: Use va_is_mapped, va_is_dirty, and ide_write.
// Hint: Use the PTE_SYSCALL constant when calling sys_page_map.
// Hint: Don't forget to round addr down.
void
flush_block(void *addr)
{
  uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;

  if (addr < (void*)DISKMAP || addr ≥ (void*)(DISKMAP +
DISKSIZE))
    panic("flush_block of bad va %08x", addr);

  // LAB 5: Your code here.
  // panic("flush_block not implemented");

  // round addr down
  addr = (void*)ROUNDDOWN((uint32_t)addr, PGSIZE);
int r;
  if(va_is_mapped(addr) && va_is_dirty(addr)){
    if((r = ide_write(blockno * BLKSECTS, addr, BLKSECTS)) ≠ 0){
      panic("flush_block, ide_write: %e", r);
    }
    if((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
PTE_SYSCALL)) ≠ 0){
      panic("flush_block, sys_page_map: %e", r);
    }
  }else{
    return;
  }
}
```

`fs/fs.c` 中的 `fs_init` 函数是如何去使用 *block cache* 的一个很好的例子。在初始化完 *blcok cache* 之后，它将指向磁盘映射区域的指针存入 `super` 全局变量。在这之后，我们可以简单地从 `super` 结构中读取就好像他们已经在内存中，然后我们的页错误处理程序将会从磁盘中读取它们当必要的时候。

The Block Bitmap

在 `fs_init` 设置完 `bitmap` 指针之后，我们可以将 `bitmap` 视为包装好的bits数组，每个磁盘上的块为1。看 `block_is_free` 这个例子，它检查一个给定的块是否在 `bitmap` 中被标记为 *free* 。

`block_is_free()` in `fs/fs.c`

```
// Check to see if the block bitmap indicates that block 'blockno' is
free.
// Return 1 if the block is free, 0 if not.
bool
block_is_free(uint32_t blockno)
{
  if (super == 0 || blockno >= super->s_nblocks)
    return 0;
  if (bitmap[blockno / 32] & (1 << (blockno % 32)))
    return 1;
  return 0;
}
```

`free_block()` in `fs/fs.c`

```
// Mark a block free in the bitmap
void
free_block(uint32_t blockno)
{
  // Blockno zero is the null pointer of block numbers.
  if (blockno == 0)
    panic("attempt to free zero block");
  bitmap[blockno/32] |= 1<<(blockno%32);
}
```

Exercise 3.

Use `free_block` as a model to implement `alloc_block` in `fs/fs.c`, 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.

Use `make grade` to test your code. Your code should now pass "alloc_block".

`alloc_block()` in `fs/fs.c`

```
// Search the bitmap for a free block and allocate it.  When you
// allocate a block, immediately flush the changed bitmap block
// to disk.
//
// Return block number allocated on success,
// -E_NO_DISK if we are out of blocks.
//
// Hint: use free_block as an example for manipulating the bitmap.
int
alloc_block(void)
{
  // The bitmap consists of one or more blocks.  A single bitmap
block
  // contains the in-use bits for BLKBITSIZE blocks.  There are
  // super->s_nblocks blocks in the disk altogether.

  // LAB 5: Your code here.
  // panic("alloc_block not implemented");
uint32_t blockno;
  for(blockno = 0; blockno < super->s_nblocks; ++blockno){
    if(block_is_free(blockno)){
      bitmap[blockno/32] &= ~(1 << (blockno % 32));
      flush_block(bitmap);
      return blockno;
    }
  }
  return -E_NO_DISK;
}
```

File Operations

我们已经在 `fs/fs.c` 中提供了很多的方法来实现基本的功能，你将需要解释和管理 `File` 结构，扫描和管理 *目录文件* 的条目，并从根目录遍历来解析一个 *绝对路径* 。通读 `fs/fs.c` 中的所有代码，确保你理解每个方法的作用，在使用之前。

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.

Use `make grade` to test your code. Your code should pass "file_open", "file_get_block", and "file_flush/file_truncated/file rewrite", and "testfile".

`file_block_walk()` in `fs/fs.c`

```
// Find the disk block number slot for the 'filebno'th block in
file 'f'.
// Set '*ppdiskbno' to point to that slot.
// The slot will be one of the f->f_direct[] entries,
// or an entry in the indirect block.
// When 'alloc' is set, this function will allocate an indirect
block
// if necessary.
//
// Returns:
//  0 on success (but note that *ppdiskbno might equal 0).
//  -E_NOT_FOUND if the function needed to allocate an indirect
block, but
//    alloc was 0.
//  -E_NO_DISK if there's no space on the disk for an indirect
block.
```

```
//  -E_INVAL if filebno is out of range (it's ≥ NDIRECT +
NINDIRECT).
//
// Analogy: This is like pgdir_walk for files.
// Hint: Don't forget to clear any block you allocate.
static int
file_block_walk(struct File *f, uint32_t filebno, uint32_t
**ppdiskbno, bool alloc)
{
    // LAB 5: Your code here.
  // panic("file_block_walk not implemented");
int r;
uint32_t disk_slot;

  // assert((f->f_size % BLKSIZE) == 0);    ⚠️ 纯纯画蛇添足，浪费大量
时间
  if(NDIRECT > filebno){
    *ppdiskbno = &(f->f_direct[filebno]);
  }else{
    if((NDIRECT + NINDIRECT) ≤ filebno){
      return -E_INVAL;
    }
    if(!f->f_indirect){
      if(alloc == 0){
        return -E_NOT_FOUND;
      }
      if((disk_slot = alloc_block()) < 0){
        return disk_slot;
      }
      memset(diskaddr(disk_slot), 0, BLKSIZE);
      flush_block(diskaddr(disk_slot));
      f->f_indirect = disk_slot;
    }
    *ppdiskbno = &(((uint32_t*)(diskaddr(f->f_indirect)))[filebno
- NDIRECT]);
  }
  return 0;
}
```

`file_get_block()` in `fs/fs.c`

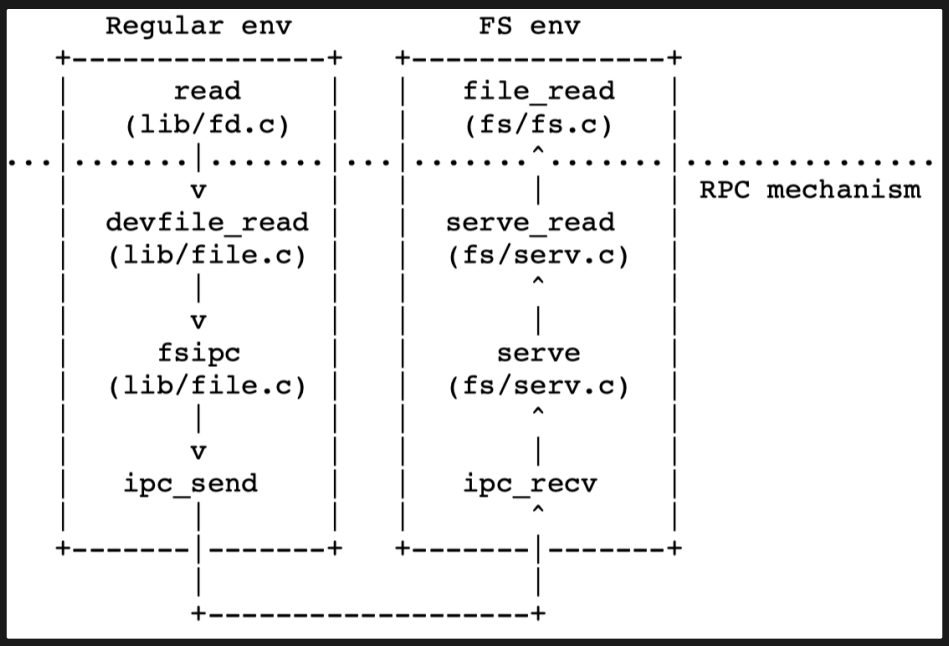
```
// Set *blk to the address in memory where the filebno'th
// block of file 'f' would be mapped.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_NO_DISK if a block needed to be allocated but the disk is
full.
//  -E_INVAL if filebno is out of range.
//
// Hint: Use file_block_walk and alloc_block.
int
file_get_block(struct File *f, uint32_t filebno, char **blk)
{
    // LAB 5: Your code here.
 //    panic("file_get_block not implemented");
uint32_t* ppdiskbno;
int r;

  if((r = file_block_walk(f, filebno, &ppdiskbno, true)) != 0){
    return r;
  }
  if(*ppdiskbno == 0){
    if((r = alloc_block()) < 0){
      return r;
    }
    *ppdiskbno = r;
    memset(diskaddr(r), 0, BLKSIZE);
    flush_block(diskaddr(r));
  }
  *blk = diskaddr(*ppdiskbno);
  return 0;
}
```

`file_block_walk` 和 `file_get_block` 是文件系统的主力。举个例子，`file_read` 和 `file_write` 只不过是在 `file_get_block` 上做一些简单的记录和处理（bookkeeping，簿记），用于在分散的块和连续缓冲区之间复制字节。

The file system interface

现在在文件系统环境本身中已经有了必要的功能，我们必须使其能够访问其他想使用文件系统的环境。因为其它环境在文件系统环境中不能直接调用函数，我们将通过*remote procedure call, RPC* 来暴露对文件系统环境的访问， *RPC* 建立在 `JOS` 的IPC机制。如下图所示，是一个对文件系统服务器的调用。



所有在点线之下的就是从一般的环境向文件系统环境发出读请求的机制。从一开始， *读取动作*（我们已经提供）在任何文件描述符上工作，然后转到合适的设备读取方法，本例为 `devfile_read()` （我们可以有更多的设备类型，如pipes）。 `devfile_read` 针对在磁盘上的文件实现 *read* 。该方法和其它在 `lib/file.c` 中的 `devfile_*` 文件实现了FS操作的客户端，并且所有的工作方式都差不多--在需求结构中绑定参数；调用 `fsipc` 来发送IPC需求；解压并返回结果。 `fsipc` 方法简单地处理向服务器发送一个需求和接收一个回复的共同细节。

文件系统服务端代码可以在 `fs/serv.c` 中被找到。它在 `serve` 函数中无尽循环来在IPC机制上接收一个请求；将请求分配给合适的处理程序函数；通过 `IPC` 将结果发送回。 *在读取例子中* ，服务端将转向 `serve_read` ，该方法将关注指定的读取需求时的IPC细节，比如unpack需求结构并最后调用 `file_read` 来执行文件读取动作。

回想一下， `JOS` 的IPC机制让一个环境发送一个单个的32位数字，和选择性地分享一个页。为了从客户端向服务端发送一个请求，我们使用32位的数字来表示请求类型（文件系统服务端RPCs被数字化，就像系统调用被数字化那样）并且将请求的参数存入通过IPC分享的页上的 `union Fsipc` 。 *在客户端* ，我们总是在 `fsipcbuf` 分享页； *在服务端* ，我们将到来的请求页映射于 `fsreq` 。

服务端也通过IPC发送回复返回。我们使用32位数字作为函数的返回代码。对于大多数的 `RPCs` ，这就是它们返回的所有东西。 `FSREQ_READ` 和 `FSREQ_STAT` 也返回数据，该数据是写入到 *客户端发送其请求的页* 的数据。没有必要在IPC回复中发送该页，因为客户端首先在文件系统服务器共享它。当然，在它的响应中， `FSREQ_OPEN` 向客户端分享一个新的"Fd page"。我

们将很快返回到 *文件描述符页* 。

Exercise 5.

Implement `serve_read` in `fs/serv.c`.

`serve_read`'s heavy lifting will be done by the already-implemented `file_read` in `fs/fs.c` (which, in turn, is just a bunch of calls to `file_get_block`). `serve_read` just has to provide the RPC interface for file reading. Look at the comments and code in `serve_set_size` to get a general idea of how the server functions should be structured.

Use `make grade` to test your code. Your code should pass "serve_open/file_stat/file_close" and "file_read" for a score of 70/150.

`serve_rade()` in `fs/serc.c`

```
// Read at most ipc->read.req_n bytes from the current seek
position
// in ipc->read.req_fileid.  Return the bytes read from the file
to
// the caller in ipc->readRet, then update the seek position.
Returns
// the number of bytes successfully read, or < 0 on error.
int
serve_read(envid_t envid, union Fsipc *ipc)
{
  struct Fsreq_read *req = &ipc->read;
  struct Fsret_read *ret = &ipc->readRet;

  if (debug)
    cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid,
req->req_n);

  // Lab 5: Your code here:
int r;
struct OpenFile *of;
  if((r = openfile_lookup(envid, req->req_fileid, &of)) != 0){
    return r;
```

```
    }
    if((r = file_read(of->o_file, ret->ret_buf, req->req_n, of-
  >o_fd->fd_offset)) > 0){
      of->o_fd->fd_offset += r;
    }

    return r;
  }
```

这里有必要提一下，`serve_read`本身实现并不难，但由于之前lab中的实现有bug，所以耗费了我比较长的时间。在`openfile_lookup()`函数中需调用`pageref()`来检查该物理页是否可访问或被使用，具体代码如下：

`pageref()`

```
int
pageref(void *v)
{
  pte_t pte;

  if (!(uvpd[PDX(v)] & PTE_P))
    return 0;
  pte = uvpt[PGNUM(v)];
  if (!(pte & PTE_P))
    return 0;

  return pages[PGNUM(pte)].pp_ref;
}
```

可以看到在最后返回时，访问了`pages`这个数组，也就是问题所在！花了很长时间都没有检查出来为什么一访问`pages`就出错，最后发现，是我在**lab2**中对`pages`进行映射时发生了权限错误！！*没有给它`PTE_U`权限*

```
diff --git a/kern/pmap.c b/kern/pmap.c
index 62221fb..0179971 100644
--- a/kern/pmap.c
+++ b/kern/pmap.c
@@ -204,7 +204,7 @@ mem_init(void)
   // change the permission of the new image at UPAGES
   // REMIND: pages is also VIRTUAL ADDRESS, need to use 'PADDR()'
   // WARNING: the permission here may be wrong
-    boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages *
sizeof(struct PageInfo), PGSIZE), PADDR(pages), PTE_P | PTE_W);
+    boot_map_region(kern_pgdir, UPAGES, ROUNDUP(npages *
sizeof(struct PageInfo), PGSIZE), PADDR(pages), PTE_P | PTE_W |
PTE_U);
     // WARNING: the permission here may be wrong
     kern_pgdir[PDX(UPAGES)] = PTE_ADDR(kern_pgdir[PDX(UPAGES)]) |
PTE_P | PTE_U;
```

甚至当时在实现的时候我还提醒自己这里的权限可能出错哈哈哈。

Exercise 6.

Implement `serve_write` in `fs/serv.c` and `devfile_write` in `lib/file.c`.

Use `make grade` to test your code. Your code should pass "file_write", "file_read after file_write", "open", and "large file" for a score of 90/150.

`serve_write` in `fs/serv.c`

```
// Write req->req_n bytes from req->req_buf to req_fileid,
starting at
// the current seek position, and update the seek position
// accordingly.  Extend the file if necessary.  Returns the number
of
// bytes written, or < 0 on error.
int
serve_write(envid_t envid, struct Fsreq_write *req)
```

```
{
  if (debug)
    cprintf("serve_write %08x %08x %08x\n", envid, req-
>req_fileid, req->req_n);

  // LAB 5: Your code here.
  // panic("serve_write not implemented");
int r;
struct OpenFile *of;
  if((r = openfile_lookup(envid, req->req_fileid, &of)) != 0){
    return r;
  }
  if((r = file_write(of->o_file, req->req_buf, req->req_n, of-
>o_fd->fd_offset)) > 0){
    of->o_fd->fd_offset += r;
  }
  return r;
}
```

`devfile_write` in `lib/file.c`

```
// Write at most 'n' bytes from 'buf' to 'fd' at the current seek
position.
//
// Returns:
//   The number of bytes successfully written.
//   < 0 on error.
static ssize_t
devfile_write(struct Fd *fd, const void *buf, size_t n)
{
  // Make an FSREQ_WRITE request to the file system server.  Be
  // careful: fsipcbuf.write.req_buf is only so large, but
  // remember that write is always allowed to write *fewer*
  // bytes than requested.
  // LAB 5: Your code here
  // panic("devfile_write not implemented");
int r;

  fsipcbuf.write.req_fileid = fd->fd_file.id;
  fsipcbuf.write.req_n = n;
```

```
    assert(n ≤ (PGSIZE - sizeof(int) - sizeof(size_t)));
    memmove(fsipcbuf.write.req_buf, buf, n);
    if((r = fsipc(FSREQ_WRITE, NULL)) < 0){
      return r;
    }
    assert(r ≤ n);
    assert(r ≤ PGSIZE);
    return r;
  }
```

Spawning Processes

我们已经提供给你了 `spawn` 的代码(see `lib/spawn.c`),该方法创建了一个新的环境，从文件系统中加载一个程序镜像，然后开始一个运行该程序的 *子环境* 。*父进程* 继续独立于 *子进程* 运行。 `spawn` 函数就像UNIX中的 `fork` 函数，在 *子进程* 中立即进行。

我们实现 `spawn` 而不是UNIX风格的 `exec` ，因为 `spawn` 更容易从 *用户空间* 以"exokernel fashion"的形式实现，而不需要kernel的帮助。考虑如果要在 *用户空间* 实现 `exec` ，你将需要做些什么，确保你理解为什么那样更困难。

Exercise 7.

`spawn` relies on the new syscall `sys_env_set_trapframe` to initialize the state of the newly created environment. Implement sys_env_set_trapframe in

`kern/syscall.c` (don't forget to dispatch the new system call in syscall()).

Test your code by running the `user/spawnhello` program from `kern/init.c` , which will attempt to spawn `/hello` from the file system.

Use make grade to test your code.

`sys_env_set_trapframe()` in `kern/syscall.c`

```
 // Set envid's trap frame to 'tf'.
```

```
// tf is modified to make sure that user environments always run
at code
// protection level 3 (CPL 3), interrupts enabled, and IOPL of 0.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//    or the caller doesn't have permission to change envid.
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
  // LAB 5: Your code here.
  // Remember to check whether the user has supplied us with a
good
  // address!
  // panic("sys_env_set_trapframe not implemented");
int r;
struct Env* e;

  if((r = envid2env(envid, &e, 0)) ≠ 0){
    return r;
  }
  user_mem_assert(e, tf, sizeof(struct Trapframe), PTE_W);
  tf→tf_cs ⊨ 3;
  tf→tf_ss ⊨ 3;
  tf→tf_eflags ⊨ FL_IF;
  tf→tf_eflags &= ~FL_IOPL_3;
  e→env_tf = *tf;
  return 0;
}
```

Sharing library state across fork and spawn

UNIX文件描述符是一个通用概念，包含了*pipes*，*console I/O*等。在`JOS`中，每个这种设备类型都有一个对应的`struct Dev`，其中含有指向实现*read/write/etc.*操作的函数。`lib/fd.c`在这之上实现了通用的UNIX类文件描述符接口。每个`struct Fd`指明了他的设备类型，并且`lib/fd.c`中的大多数函数就是将操作转向合适的`struct Dev`中的函数。

`lib/fd.c` 也在每个应用环境的地址空间维护 *文件描述符表区域*，开始于 `FDTABLE`。该区域保留了4KB的地址空间给最多 `MAXFD` （目前是32）个文件描述符（应用一次性可以打开的数量）。在任何给定时间，一个指定的 *文件描述符表页* 被映射**当且仅当**对应的文件描述符正在使用。每个文件描述符也有一个可选择的“数据页”，位于开始于 `FILEDATA` 的区域，设备可以选择是否使用。

我们想在 `fork` 和 `spawn` 之间共享 *文件描述符状态*，但是 *文件描述符状态* 被保存于用户空间内存。现在，（对于 `fork`），内存将会被标记为 `copy-on-write`，所以状态会被复制而不是被共享。（这意味着环境不能在它们自己没有打开的文件中 *seek* 并且 *pipes* 不能夸 `fork` 工作；（对于 `spawn`），内存将会被留下，完全不复制。（实际上，the spawned environment 在开始时没有打开的文件描述符）。

我们将让 `fork` 了解内存中特定的区域被 *library operating system* 所使用并应该被分享。我们将在 *page table entries* 设置一个 *otherwise-unused* 位（就像在 `fork` 中使用 `PTE_COW` 位），而不是在某个地方硬编码一个区域的链表。

我们已经在 `inc/lib.h` 中定义了一个新的 `PTE_SHARE` 位。这个位是三个PTE位之一，这三个PTE位在Intel和AMD手册上表示"available for software use"。我们将建立一个共识，如果一个 *page table entry* 设置了 `PTE_SHARE`，该PTE应该被直接从父进程复制到 `fork` 和 `spawn`。注意，这不同于将其标记为 `copy-on-write`，就像在第一段描述的一样，我们想确保 *共享* 对页的更新。

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.

`duppage` in `lib/fork.c`

```
// Map our virtual page pn (address pn*PGSIZE) into the target
envid
// at the same virtual address.  If the page is writable or copy-
on-write,
```

```
// the new mapping must be created copy-on-write, and then our
mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to
mark ours
// copy-on-write again if it was already copy-on-write at the
beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
static int
duppage(envid_t envid, unsigned pn)
{
  int r;

  // LAB 4: Your code here.
void* addr = (void*)(pn * PGSIZE);
pte_t pte = uvpt[pn];
  if((pte & PTE_SHARE) == PTE_SHARE){
    if((r = sys_page_map(0, addr, envid, addr, (pte &
PTE_SYSCALL))) != 0){
      panic("duppage: %e", r);
    }
  }else if((pte & PTE_W) == PTE_W || (pte & PTE_COW) == PTE_COW){
    if((r = sys_page_map(0, addr, envid, addr, PTE_COW | PTE_P |
PTE_U)) != 0){
      panic("duppage: %e", r);
    }
    if((r = sys_page_map(0, addr, 0, addr, PTE_COW | PTE_P |
PTE_U))){
      panic("duppage: %e", r);
    }
  }else{
    if((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_W |
PTE_U)) != 0){
      panic("duppage: %e", r);
    }
  }
```

`copy_shared_pages` in `lib/spawn.c`

```
// Copy the mappings for shared pages into the child address
space.
static int
copy_shared_pages(envid_t child)
{
  // LAB 5: Your code here.
int r, pn;
uint32_t addr;
struct Env* e;
envid_t parent_id = sys_getenvid();

  for(addr = 0;addr < USTACKTOP; addr += PGSIZE){
    if((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P)){
      if((uvpt[PGNUM(addr)] & PTE_SHARE) == PTE_SHARE){
        if((r = sys_page_map(0, (void*)addr, child, (void*)addr,
(uvpt[PGNUM(addr)] & PTE_SYSCALL)))){
          return r;
        }
      }
    }
  }
  return 0;
}
```

Use `make run-testpteshare` to check that your code is behaving properly. You should see lines that say "*fork handles PTE_SHARE right*" and "*spawn handles PTE_SHARE right*".

Use `make run-testfdsharing` to check that file descriptors are shared properly. You should see lines that say "*read in child succeeded*" and "*read in parent succeeded*".

The keyboard interface

为了让 `sheel` 工作，我们需要一种方式在其中输入。 `QEMU` 一直显示我们写入 `CGA` 和 `serial port` 的输出，但是到目前为止我们只在 *kernel monitor* 中用过输入。在 `QEMU` 中，输入在图形窗口的输入显示为从键盘到 `JOS` 的输入；而键入到控制台的输入显示为串口上的字符。 `kern/console.c` 已经包含了 *keyboard* 和 *serial drivers*， *keyboard* 和 *serial drivers* 从lab1就一直被 *kernel monitor* 使用，但是现在你需要将这些附加到系统的其它部分。

> 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`.

```
@@ -278,6 +278,12 @@ trap_dispatch(struct Trapframe *tf)
             // case SYS_yield:
             //      sys_yield();
             //      return;
+            case IRQ_OFFSET + IRQ_KBD:
+                    kbd_intr();
+                    return;
+            case IRQ_OFFSET + IRQ_SERIAL:
+                    serial_intr();
+                    return;
```

我们为你实现了 *控制台输入/输出文件类型*，位于 `lib/console.c` 中。 `kbd_intr` 和 `serial_intr` 用最近读到的输入填充缓冲区，而 *控制台文件类型* 耗尽缓冲区（*控制台文件类型* 默认用于 *stdin/stdout*，除非用户将它们重定向。

Test your code by running `make run-testkbd` and type a few lines. The system should echo your lines back to you as you finish them. Try typing in both the console and the graphical window, if you have both available.

The Shell

Run `make run-icode-nox` or `make run-icode-nox` 。这将运行你的内核并且开始 `user/icode` 。 `icode` 执行 `init` ， `init` 将控制台设置为 *文件描述符0和1* （标准输入和标准输出）。然后，它将*spawn* `sh` ， *the shell* 。你应该能够运行下面的命令：

```
echo hello world | cat
cat lorem |cat
cat lorem |num
cat lorem |num |num |num |num
lsfd
```

注意，用户库例程 `cprintf` 直接打印到控制台，不使用 *文件描述符* 代码。这对调试来说很好，但对于管道传输到其它程序不好。为了打印输出到一个指定的 *文件描述符* （比如，1，标准输出），使用 `fprintf(1, "...", ...).printf("...", ...)` 是打印到 `FD 1` 的快捷操作。See `user/lsfd.c` for examples.

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

Run `make run-testshell` to test your shell. `testshell` simply feeds the above commands (also found in `fs/testshell.sh`) into the shell and then checks that the

output matches `fs/testshell.key` .

注意：在 `lib/fork.c` 中的 `dippage()` 再次发生权限错误。

```
@@ -83,7 +83,7 @@ pte_t pte = uvpt[pn];
                        panic("duppage: %e", r);
                }
        }else{
-               if((r = sys_page_map(0, addr, envid, addr, PTE_P |
PTE_W | PTE_U)) < 0){
+               if((r = sys_page_map(0, addr, envid, addr, PTE_P |
PTE_U)) < 0){
                        panic("duppage: %e", r);
                }
        }
```

`runcmd()` in `user/sh.c`

```
diff --git a/user/sh.c b/user/sh.c
index 26f501a..04f1f59 100644
--- a/user/sh.c
+++ b/user/sh.c
@@ -55,7 +55,15 @@ again:
                        // then close the original 'fd'.

                        // LAB 5: Your code here.
-                       panic("< redirection not implemented");
+                       // panic("< redirection not implemented");
+                       if((fd = open(t, O_RDONLY)) < 0){
+                               cprintf("open %s for write: %e",
t, fd);
+                               exit();
+                       }
+                       if(fd != 0){
+                               dup(fd, 0);
+                               close(fd);
+                       }
                        break;
```

Your code should pass all tests at this point. As usual, you can grade your submission with `make grade` and hand it in with `make handin`.

This completes the lab. As usual, don't forget to run `make grade` and to write up your answers and a description of your challenge exercise solution. Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab5.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 5'`, then `make handin` to submit your solution.