**Lab 5**

**File system preliminaries**

**1.On-Disk File System Structure**

大多数UNIX文件系统将可用磁盘空间划分为两种主要类型的区域：inode区域和数据区域。UNIX 文件系统为文件系统中的每个文件分配一个inode，文件的 inode 保存有关文件的关键元数据。而数据区域被划分为更大（通常为 8KB 或更多）的数据块，文件系统在其中存储文件数据和目录元数据。

文件目录项包含文件名和指向inode的指针。

硬链接：文件系统中的多个目录项引用该文件的inode，则称这个文件是硬链接的。JOS中不支持硬链接，也不使用inode，只存储所有文件（或子目录）描述该文件的唯一的目录项中的元数据。

我们不必深究块布局的细节，文件系统都将其包装了起来，提供了用于读取和写入文件内任意偏移量的字节序列的接口。文件系统环境在内部处理对目录的所有修改，作为执行文件创建和删除等操作的一部分。我们的文件系统允许用户环境直接读取目录元数据（例如，使用read），这意味着用户环境可以自己执行目录扫描操作（例如，实现ls程序），而不必依赖额外的特殊调用。

**2.Sectors and Blocks**

大多数磁盘不能以字节为单位执行读取和写入，而是以扇区为单位执行读取和写入操作。在JOS中，扇区为512字节。文件系统实际上以块为单位分配和使用磁盘存储。请注意两个术语之间的区别：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。文件系统的块大小必须是底层磁盘扇区大小的倍数。

UNIX xv6文件系统使用512字节的块大小，与底层磁盘的扇区大小相同。我们的文件系统将使用4096字节的块大小，为了方便匹配处理器的页面大小。

**3.Superblocks**

文件系统通常将某些磁盘块保留在磁盘上的“易于查找”的位置（例如起始或最后），以保存描述整个文件系统属性的元数据，例如块大小，磁盘大小，找到根目录所需的任何元数据，文件系统上次挂载的时间，文件系统上次检查错误的时间等等。这些特殊块称为超级块。

我们的文件系统将只有一个超级块，它将始终位于磁盘上的块1。它由struct Super在inc/fs.h中定义。块0通常保留用于保存引导加载程序和分区表，因此文件系统通常不使用第一个磁盘块。许多“真正的”文件系统具有多个超级块，这几个副本在磁盘的几个广泛间隔的区域，以便如果其中一个被损坏或磁盘在该区域中产生媒体错误，则仍然可以找到其他超级块，并将其用于访问文件系统。图1是磁盘块的结构。

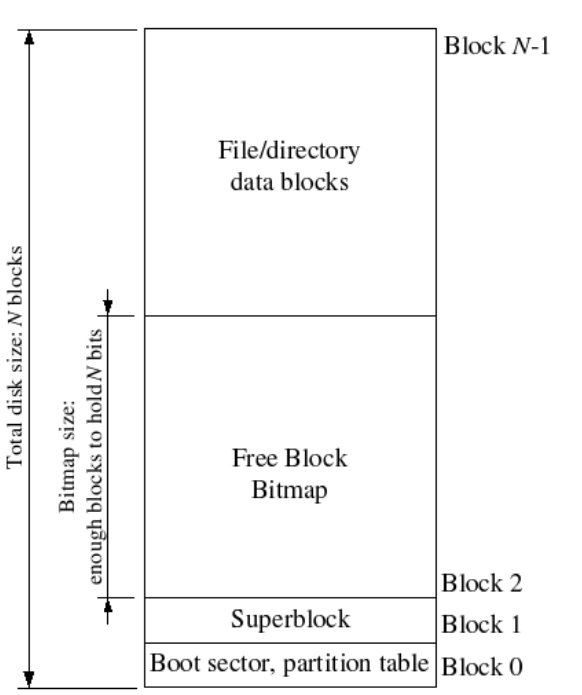


图1

**4.File Meta-data**

描述文件系统中的文件的元数据由inc/fs.h中的struct File定义。该元数据包括文件的名称，大小，类型（常规文件或目录）以及指向包含该文件的块的指针。如上所述，我们没有inode，所以元数据存储在磁盘上的目录条目中。

struct File中的f\_direct数组包含存储文件前10个（NDIRECT）块的块号的空间，这前10个块被称之为文件的直接块。对于大小为10 \* 4096 = 40KB的小文件，这意味着所有文件块的块号将直接适用于struct File本身。然而，对于较大的文件，我们需要一个地方来保存文件的其他块号。因此，对于大于40KB的任何文件，我们分配一个额外的磁盘块，称为文件的间接块，最多容纳4096/4 = 1024个附加块号。因此，我们的文件系统允许文件的大小可达1034个块，或者刚刚超过4MB大小。图2是文件的元数据的结构。

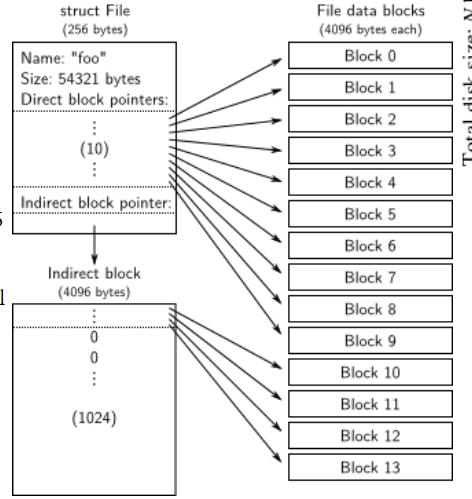


图2

**5.Directories versus Regular Files**

我们的文件系统中的struct File可以表示常规文件或目录;这两种类型的“文件”通过struct File中的类型字段进行区分。文件系统以完全相同的方式管理常规文件和目录文件，除了它不解释与常规文件相关联的数据块的内容，而文件系统将目录文件的内容解释为一系列描述目录中的文件和子目录的struct File。

我们的文件系统中的超级块包含一个struct File（struct Super中的根字段），它保存文件系统根目录的元数据。根目录文件的内容是描述位于文件系统根目录下的文件和目录的struct File序列。根目录中的任何子目录可以依次包含表示子子目录的更多的struct File，依此类推。

**The File System**

**Disk Access**

操作系统中的文件系统环境需要能够访问磁盘，但是我们还没有在内核中实现任何磁盘访问功能。JOS不采取传统的单内核操作系统的策略将IDE磁盘驱动器添加到内核中，允许文件系统以系统调用访问它，而是将IDE磁盘驱动器作为用户级文件系统的一部分。我们仍然需要稍微修改内核，以便设置文件系统环境具有实现磁盘访问所需的权限。

x86处理器使用EFLAGS寄存器中的IOPL位来确定是否允许保护模式代码执行特殊的设备I/O指令，如IN和OUT指令。由于我们需要访问的所有IDE磁盘寄存器位于x86的I/O空间中，而不是内存映射，因此为文件系统环境提供“I/O特权”是我们唯一需要做的，以便允许文件系统访问这些寄存器。实际上，EFLAGS寄存器中的IOPL位为内核提供了一种简单的“全或无”方法来控制用户态代码能否访问I/O空间。在我们的实现中，我们希望文件系统环境能够访问I/O空间，但是我们不希望任何其他环境能够访问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.

设置eflags寄存器的IOPL位即可，如图3所示。

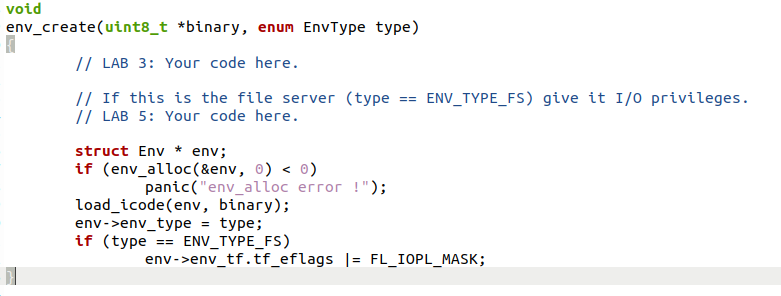


图3

**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?

不用，用户环境切换的时候，eflags寄存器的状态有CPU压入内核栈，最后由env\_pop\_tf的iret指令恢复eflags寄存器状态。

**The Block Cache**

在我们的文件系统中，我们将在处理器的虚拟内存系统的帮助下实现一个简单的“缓冲区缓存”（实际上只是块缓存）。

我们的文件系统将限于处理大小为3GB或更小的磁盘。我们保留一个大的，固定的3GB区域的文件系统环境的地址空间，从0x10000000（DISKMAP）~0xD0000000（DISKMAP + DISKMAX），作为磁盘的“内存映射”。例如，磁盘块0映射为虚拟地址0x10000000，磁盘块1映射到虚拟地址0x10001000，依此类推。fs/bc.c中的diskaddr函数实现了从磁盘块号到虚拟地址的转换。

由于我们的文件系统环境具有独立于系统中所有其他环境的虚拟地址空间的虚拟地址空间，文件系统环境唯一需要做的就是实现文件访问，因此保留了大部分文件系统环境的这个地址空间。

当然，将整个磁盘读入内存需要很长时间，因此我们将实现一种按需分页的形式，其中我们只在磁盘映射区域中分配页面，并从磁盘读取相应的块，以响应这个区域的页面错误。这样，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](http://pdos.csail.mit.edu/6.828/2011/readings/i386/s05_02.htm) of the 386 reference manual.) After writing the block to disk, flush\_block should clear the PTE\_D bit using sys\_page\_map.

首先是bc\_pgfault函数，根据注释，先分配页给硬盘，再将硬盘中的内容读进页中，需要注意要将addr对齐PGSIZE。代码如图4所示。

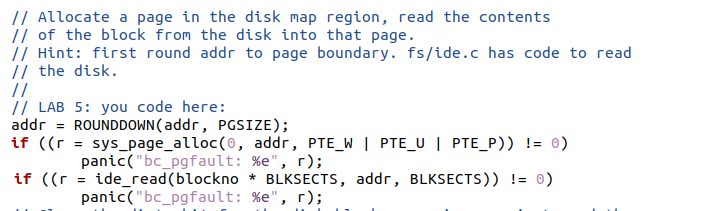


图4

然后是flush\_block函数，这个函数的作用是刷新块的内容，然后调用sys\_page\_map函数修改位，也需要页对齐。如图5所示。

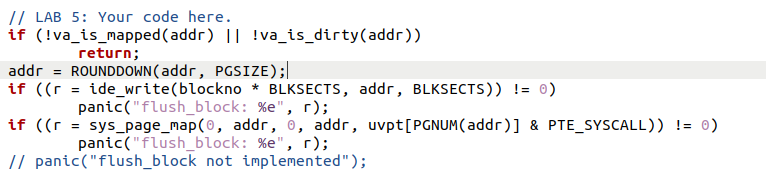


图5

**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.

实现alloc\_block函数对位图进行操作，为一个空闲的块查找一个位图并将其分配给它，将free\_block函数作为一个模板，然后分配完之后需要调用flush\_block对修改的块进行刷新，如图6所示

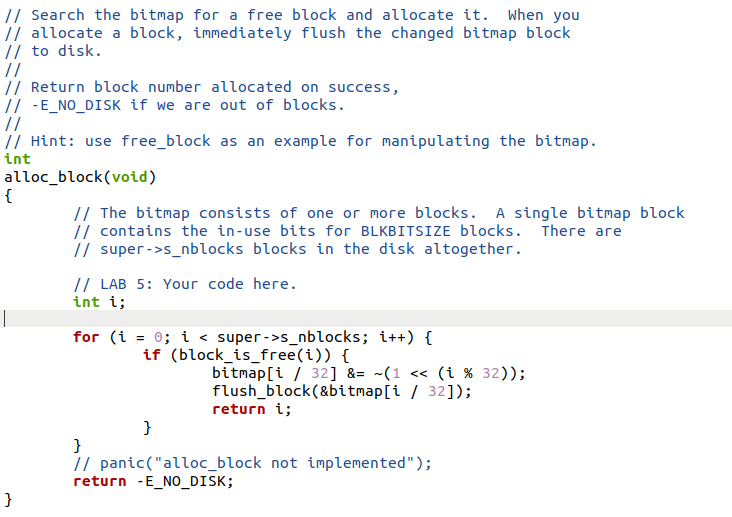


图6

**Exercise4.** 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.

首先是file\_block\_walk，这个函数的作用是在文件里面找到第filebno块对应的地址，储存到\*ppdiskbno，可能在f->f \_direct[]里面或者间接块里面，如果我们分配位置设置了，就分配一个块，同时在写的过程中需要进行一些错误检查，代码如图7所示

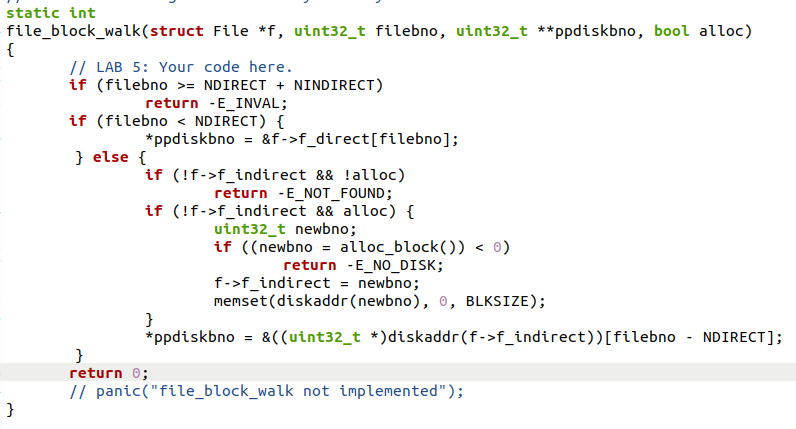


图7

file\_get\_block函数的作用是设置\*bLk为文件f的第fileno个块的地址，显然是要用到上面一个函数，如果pdiskbno不存在就去分配一个新块，如图8所示

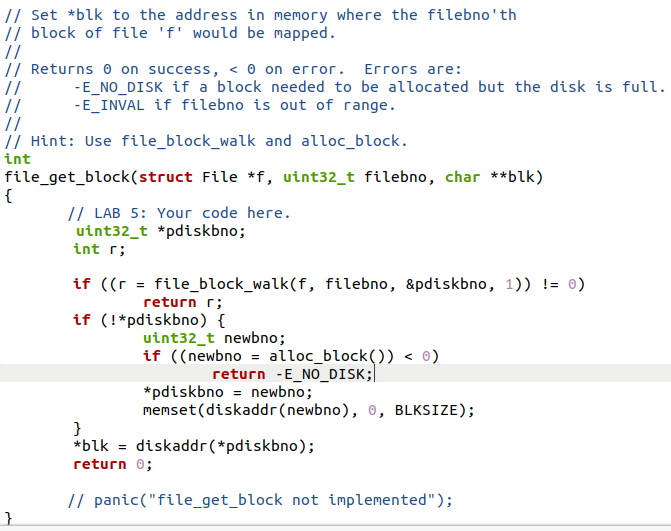
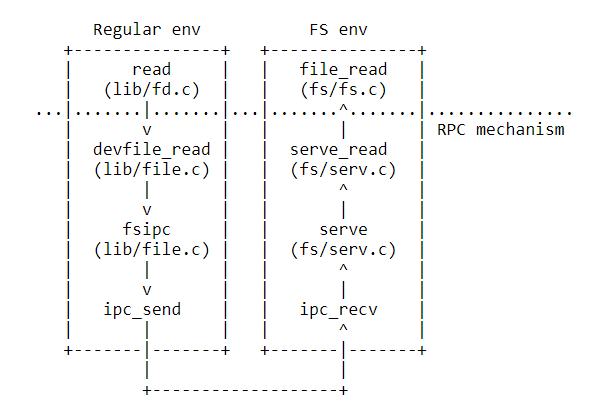


图8

现在我们的文件系统环境已经具有必要的功能，我们必须让其他希望使用文件系统的用户环境可以访问文件系统。由于其他用户环境不能直接调用文件系统环境中的函数，因此我们将通过远程过程调用（RPC）即在JOS的IPC机制上封装构建的对文件系统环境的访问接口。如下图所示，对文件系统服务器的调用：



虚线以下的所有内容都只是从常规用户环境到文件系统环境的读取请求的机制。从最开始，read适用于任何文件描述符，并且简单地分发到适当的设备读取函数，在这种情况下我们可以有更多的设备类型，如管道。devfile\_read专门实现为读取磁盘文件。lib/file.c中的devfile\_read和其他devfile\_\*函数实现了客户端FS操作，并且都以大致相同的方式工作，在request结构体中封装参数，调用fsipc发送IPC请求，并解包和返回结果。fsipc函数简单地处理向服务器发送请求并接收回复的常见细节。

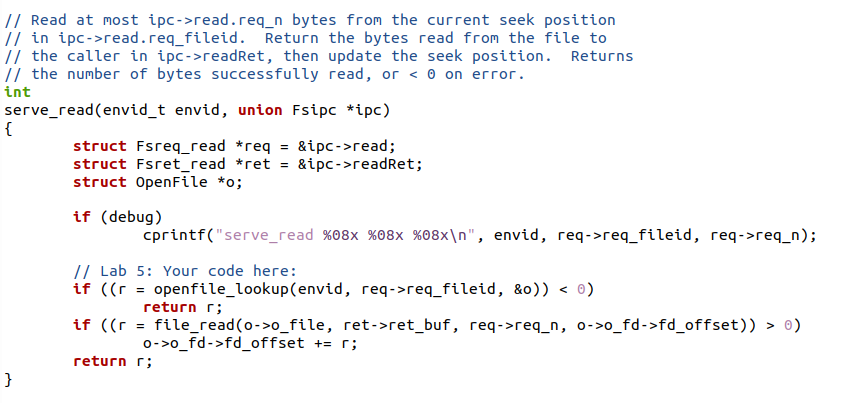
文件系统服务器代码可以在fs/serv.c中找到。它在serve函数中循环，无休止地通过IPC接收请求，将该请求分发到适当的处理函数，并通过IPC发送结果。在read示例中，serve函数将请求分发到serve\_read，它将处理read请求指定的IPC细节，例如解包request结构体，最后调用file\_read来实际执行文件读取。

JOS的IPC机制允许用户环境发送一个32位数字，并且可选地共享一个页面。要从客户端发送请求到服务器，我们使用32位数字作为请求类型（文件系统服务器RPC也是像系统调用编号那样编号），并将参数存储在通过IPC共享页面的fsipc联合类型中的request结构体中。在客户端，我们总是在fsipcbuf共享页面;在服务器端，我们将传入请求页映射到fsreq（0x0ffff000）。

服务器还通过IPC发回响应。我们使用32位数字作为函数的返回码。对于大多数RPC，这些是他们返回的全部内容。FSREQ\_READ和FSREQ\_STAT也返回数据，它们只是将数据写入客户端发送请求的页面。无需在响应IPC中发送此页面，因为客户端首先与文件系统服务器共享。此外，在其回复中，FSREQ\_OPEN与客户端共享一个新的Fd page。

Exercise 5和6是实现函数serve\_read和serve\_write，devfile\_write

Serve\_read首先查找打开的文件，然后调用read函数读取n个字节，注意一下读取的文件和需要读取到的目标buffer即可，serve\_write和serve\_read类似，如图9



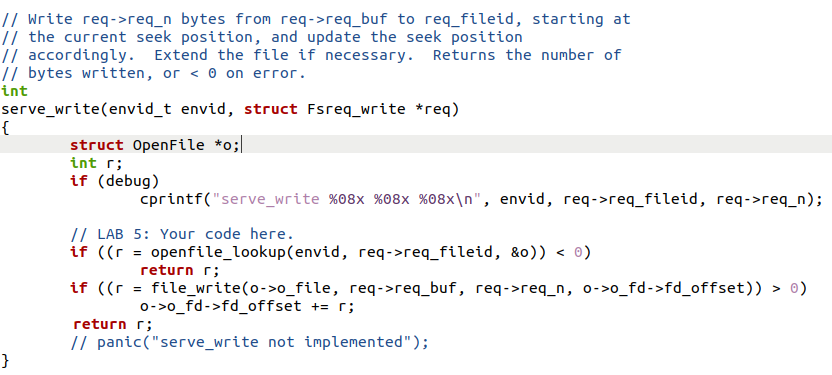


图9

devfile\_write是为write函数准备的，和devfile\_read类似，写入数据，调用fsipc即可，如图10

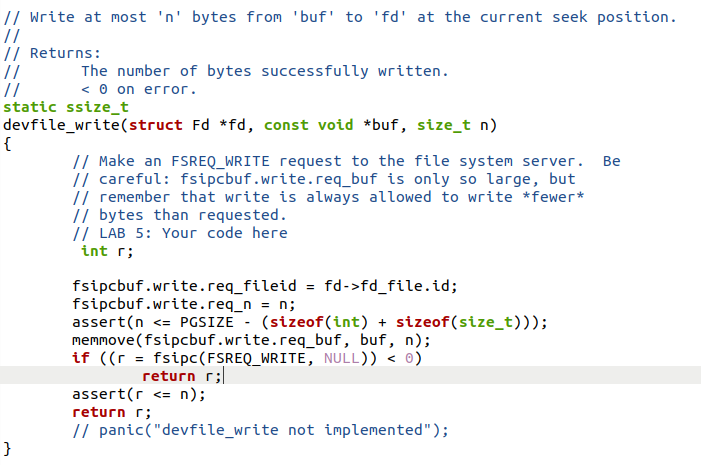


图10

**Spawning Processes**

实验已经给我们提供了生成一个新的用户环境spawn的代码（参见lib/spawn.c），将文件系统中的程序镜像加载到用户环境中，然后启动运行此程序的子环境。然后，父进程继续独立于该子进程运行。spawn函数就像UNIX中fork后面紧接着在子进程中执行exec。

**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()).

对对于应的env设置权限，开启中断，然后设置IOPL of 0，也就是清楚之前设置的IOPL，代码如图11所示

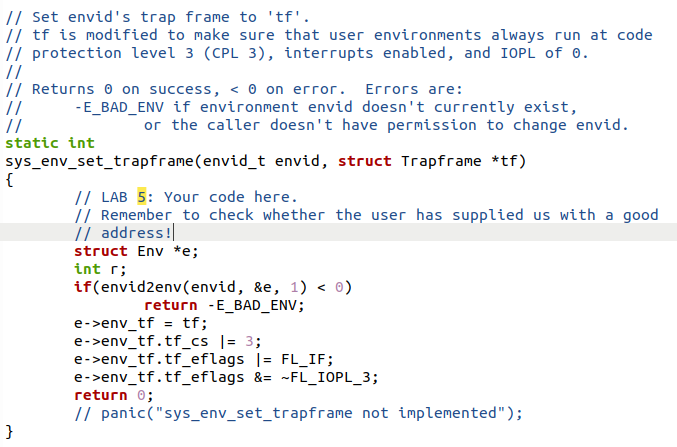


图11

UNIX文件描述符是一个普遍的概念，它包括管道，控制台I/O等。在JOS中，每一个这样的设备类型的都具有相应的struct Dev，该结构体中有指向实现该设备类型读/写等操作的函数指针。lib/fd.c在struct Dev之上实现了一般的类UNIX文件描述符接口。每个struct Fd表示其设备类型，并且lib/fd.c中的大多数函数简单地将操作分发到合适的struct Dev中的函数。

lib/fd.c还在每个应用程序环境的地址空间虚拟地址FDTABLE开始中处护文件描述符表区域。该区域为应用程序可以一次能最多打开MAXFD（当前为32）个文件描述符的每个文件描述符保留一个页面（4KB）大小的地址空间。在任何时候，当且仅当相应的文件描述符被使用时，文件描述符表页会被映射。每个文件描述符还有一个在FILEDATA开始的区域可选的“数据页”，如果选择了数据区域，设备就可以使用该区域。

我们想在fork和spawn之间共享文件描述符状态，但文件描述符状态保存在用户空间内存中。现在，fork完成之后，内存将被标记为copy-on-write，所以状态将被重复而不是共享。 （这意味着环境将无法在他们没有打开自己的文件中查找，并且管道将在fork之后也不能运行。）spawn完成之后，内存将被丢弃，完全不会被复制。（有效地，spawn的用户环境在没有打开的文件描述符状态下开始运行。）

我们将更改fork，让fork时了解某些区域的内存是由“库操作系统”使用，应始终共享。而不是在某个地方硬编码一个区域列表，我们将在页表项中设置一个未使用的位（就像我们在fork中使用PTE\_COW位一样）来确定共享区域。

我们已经在inc/lib.h中定义了一个新的PTE\_SHARE位。该位是Intel和AMD手册中标记为“可用于软件使用”的三个PTE位之一。我们约定，如果一个页表项中该位置位，应该在fork和spawn中直接将该PTE从父对象复制到子环境。请注意，这不同于标记PTE为copy-on-write：如第一段所述，我们要确保共享页面更新。

**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.

第一个是修改duppage函数，对是否设置了PTE\_SHARE进行判断，如果是就进行映射，如图12

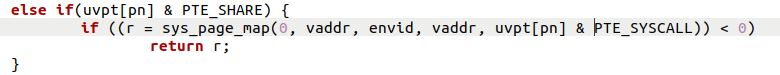


图12

第二个实现copy\_shared\_pages函数循环遍历所有页表项，对PTE\_SHARE位进行判断，然后进行映射，和上面的操作类似，如图13

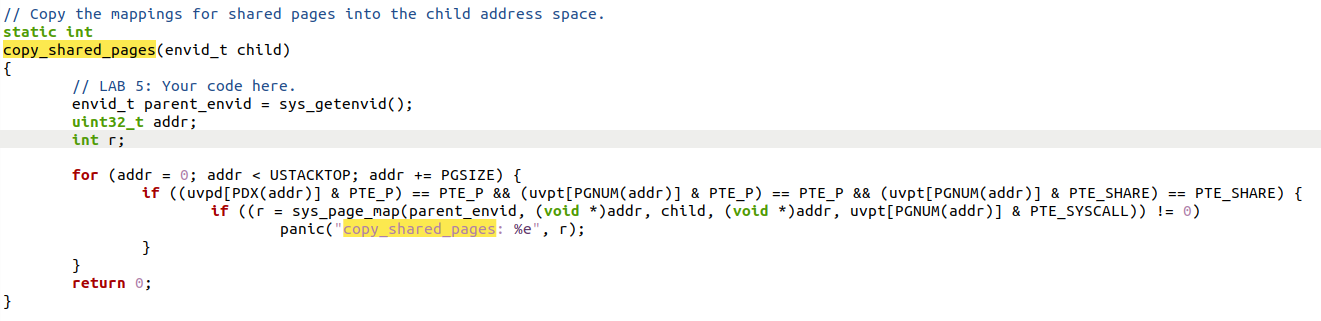


图13

Exercise 9是在trap\_dispatch函数中添加对keyboard and serial interrupts的处理，和之前一样，在此不再赘述。

**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.

实现输入重定向，仿造给定的输出重定向即可，如图14

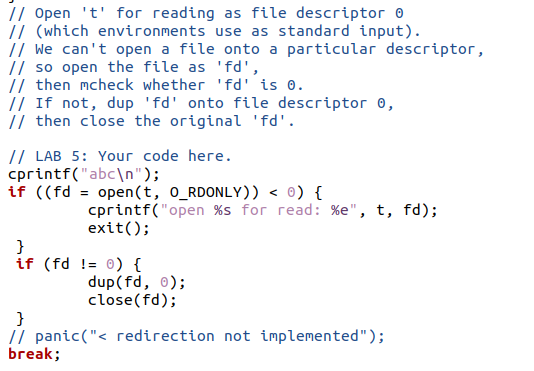


图14

最后一个测试testshell不知为何总是通不过，debug弄了几天都没搞懂，结果如图15所示

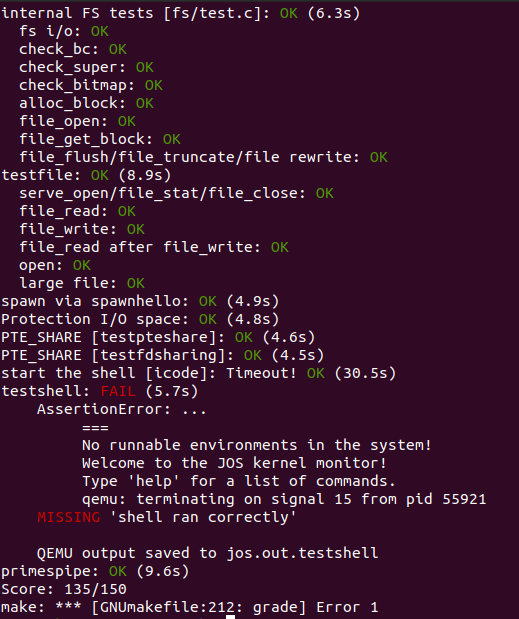


图15