操作系统上机实验指南

第四次作业说明

文件系统

单位：	南开大学机器智能研究所 Institute of Machine Intelligence. Nankai University
日期：	2015/12/30

目录

1 概述

本次作业共 8 道作业题，无需提交文档，仅提交源码。

本次作业的目的是实现一个简单的基于磁盘的文件系统，文件系统本身将按照微内核的模式（micro-kernel fashion）进行设计, 即位于内核之外,却有自己的用户空间。其他环境（进程）通过使用 IPC 请求来访问文件系统。

1.1 实验准备

同以往一样，使用 git 获得本次作业的源码，类似下图：

```
zhong@localhost:~/Desktop/lab$ git commit -am 'my solution to lab4'
nothing to commit (working directory clean)
zhong@localhost:~/Desktop/lab$ git pull
Already up-to-date.
zhong@localhost:~/Desktop/lab$ git checkout -b lab5 origin/lab5
Branch lab5 set up to track remote branch refs/remotes/origin/lab5.
Switched to a new branch "lab5"
zhong@localhost:~/Desktop/lab$ git merge lab4
Merge made by recursive.
 kern/env.c |   42 ++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
```

本次添加的源码主要是文件系统环境（file system environment）的部分，位于 fs 目录下，同时在 user 目录和 lib 目录也有一些新的源文件,特别是 lib/fsipc.c、 lib/file.c、lib/spawn.c 和新的全局头文件 inc/fs.h 和 inc/fd.h。

merge 之前的代码后，确保上次作业的 pingpong,primes,forktree 测试程序能够正常运行。为了运行这些测试程序，请先注释掉 ENV_CREATE(fs_fs)（in kern/init.c），因为 fs/fs 做了一些 目前 JOS 还不支持的 I/O 操作。同样地,暂时注释掉文件 lib/exit.c 中对 close_all()的调用,这个函数中调用的相关功能你将在本次实验中实现，现在调用会发生 panic。请确保刚才提到的测试程序能够运行。最后需要注意的是，当你开始下面的实验时，记得把刚才注释掉的代码解注释。

1.2 文件系统初步

本次试验中要实现的文件系统比大多数真实的文件系统要简单的多,但是已经足够支持基本的功能:在一个层次化的目录结构中创建、读取、写入和删除文件。现在我们仅仅是完成一个单用户的系统，可以提供足够的保护来捕捉 BUG，但是还不足以对多个相互独立的用户之间提供保护。因此,我们的系统就不支持 UNIX 中的文件所有者或是权限管理。当前的文件系统也不能像大多数 UNIX 文件系统那样支持硬连接(hard link)、符号链接（symbolic links）、时间戳(time stamps)或者其他的特殊设备文件。

1.3 磁盘上的文件系统（On-Disk File System Structure）

大多数的 UNIX 文件系统将可用磁盘空间分为两类主要的区域:inode 区域和数据（data）区域。 UNIX 文件系统将一个 inode 和文件系统的文件相关联;一个文件的 inode 保存着文件的关键数据,例如它的 stat 属性和指向数据块的指针. 数据区域则按照较大的数据块(data block) 进行划分,文件系统在这些地方保存着文件的数据以及目录的元数据 (meta data). 目录项（Directory entries）包括文件名和指向 inode 的指针; 仅当多个目录项都包含着一个文件的 inode 时，这个文件被称为硬连接(hard-linked)的。由 于我们的文件系统不支持硬连接,我们就不需要这个等级上的间接调用并给出一个简化设计: 我们的文件系统将不使用 inode,而是将文件或者子目录的元数据保存在其对应的目录项中, 而这个目录项有且只有一个。

文件和目录在逻辑上都包含着一系列数据块,就像是进程的虚拟内存地址可以被分散的映射在物理内存上一样,这些数据也可以零散的保存在磁盘上。文件系统环境（file system environment）隐藏了数据块布局的具体细节，只提供了允许用户程序读写文件任意偏移量数据的接口。文件系统保留了所有对目录进行修改的操作,例如文件创建和删除。 我们的文件系统确实允许用户程序直接的读取目录的数据(如 read),也就是说用户程序可以直接的完成扫描操作(如 ls 命令)而不是借助于文件系统的特殊调用。 这种设计的缺点,也是大多数现代 UNIX 操作系统不使用这种设计的原因就在于这将导致程序依赖于目录数据的格式, 使得程序很难在各种文件系统之间进行移植。

1.4 扇区和块（Sectors and Blocks）

大多数的磁盘不能在字节的粒度上进行读写,而只能按照扇区(sector)的大小进行操作,其一般的大小为 512 字节。文件系统实际分配和使用磁盘是按照扇区为单位的。请一定要弄明白这两个概念:扇区大小(sector size)是磁盘硬件的属性而块大小(block size)则是操作系统使用磁盘的单元。一个文件系统的块大小必定是一个扇区的整数倍。

UNIX 的 xv6 中的文件系统使用 512 字节的块大小,与扇区大小一致。大多数的现代文件系统使用更大的块大小,因为磁盘空间越来越便宜并且使用大粒度管理磁盘更有效率。我们的文件系统使用 4096 字节作为块大小,恰好和处理器页的尺寸一致。

1.5 超级块（Superblocks）

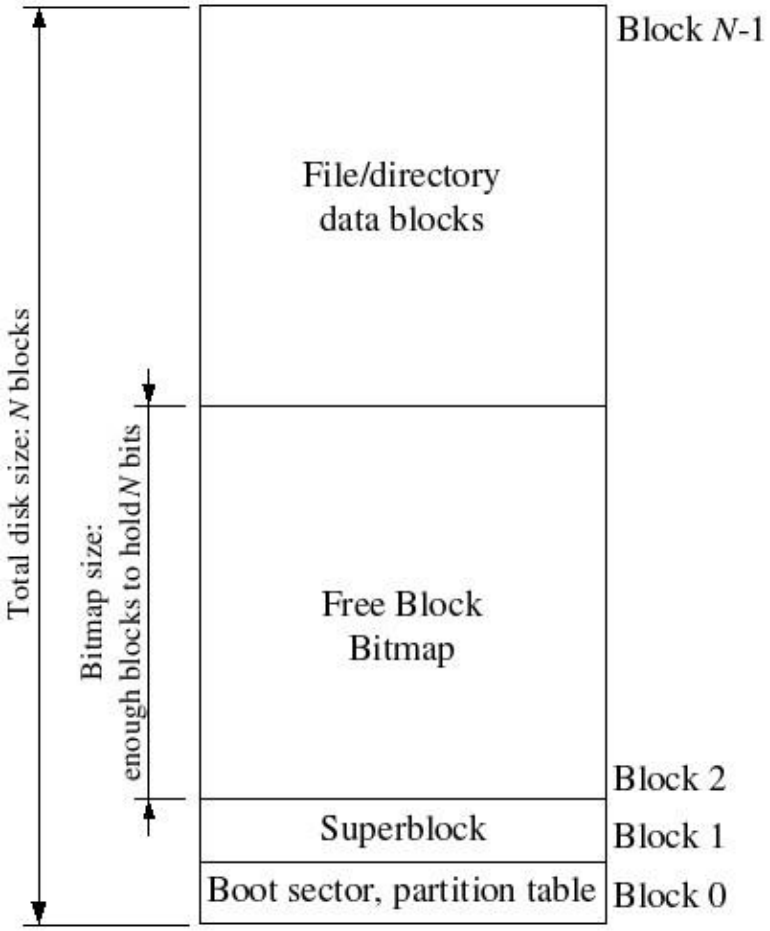
文件系统通常都会保留一定的磁盘块,在一些比较好找的地方上,例如在磁盘的最开始或者是结尾,用来保存描述整个文件系统的数据,例如块大小、磁盘大小、用于找到 root 目录的数据、上次挂载这个文件系统的时间、上次检查错误的时间等等。这些特殊的块被称 为超级块(Super Block)。

我们的文件系统只有一个超级块,总被放置在磁盘上的 Block 1 位置。它的布局是按照 inc/fs.h 中对于结构 struct Super 进行定义的。Block 0 通常被保留用来保存 boot loaders 和分区表,所以文件系统不会使用这一个磁盘块。大多数真实的文件系统维持多个冗余的超级块, 散布在磁盘的不同地方,一旦其中的一个出现了错误或者磁盘介质在这个地方出现了错误, 其他的超级块可以被找到以访问文件系统。

1.6 位图:管理空闲磁盘块（The Block Bitmap: Managing Free Disk Blocks）

和操作系统要求确保每个物理内存页同时只能有一个用途一样,文件系统必须要保证磁盘上的块同时只能有一个用途。在文件 pmap.c 中我们将所有空闲的物理内存页保存在一个链接表 page_free_list 上,以跟踪每一个空闲的物理页。在文件系统中,更多的是使用位图(bitmap)来跟踪每一个空闲的块,因为位图在存储上更有效率并且易于保持一致。在一个位图结构中寻找一个空闲磁盘块要比在空闲链表中删除头部元素耗费更多的 CPU 时间,但是

对于文件系统而言这并不是个问题,因为在找到空闲磁盘块后花费在 I/O 上的时间更决定着整体的性能。



为了设置一个空闲块的位图结构,我们保留一部分连续的足够大的磁盘空间来为每一个磁盘块分配一个 bit。例如,我们的文件系统使用 4096 字节的块,每个块有 4096*8=32768 个 bit,也就是说可以描述 32768 个磁盘块。或者说,对于每 32768 个磁盘块,就需要一个额外的磁盘块来保存位图。在位图中的某一位如果被置位,就是说对应的块是空闲的,如果被清空就说明是正在使用的。在我们的文件系统中,位图从磁盘的 Block 2 开始,紧跟着超级块。出于简化的目的,我们将保留足够的位图块以为整个磁盘中的每一个块分配一个 bit, 包括包含着超级块和位图结构的块。我们将简单的确保位图中这些特殊区域所对应的 bit 总是清空的,意即在使用中。

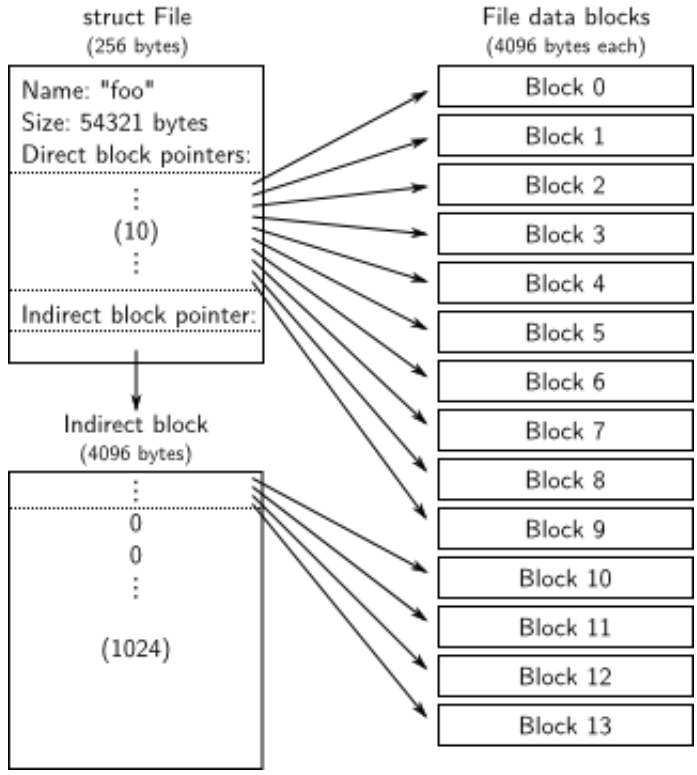
1.7 文件元数据（File Meta-data）

文件的元数据(Meta-data)是描述文件的基本属性的，其结构定义在 inc/fs.h 中的结构 struct File 中。这些元数据包括文件名、大小、类型(普通文件还是目录)以及指向文件数据块的指针。如上所述，我们没有 inodes,所以文件元数据存储在磁盘的目录项（directory

entry）中。不像大多数真实的文件系统，出于简便的考虑，我们将同时使用这个 File 结构来在磁盘和内存中描述文件的元数据。

在结构 File 中的 block 数组保存着该文件的前 NDIRECT(当前为 10)个磁盘块的块号，我们称之为文件的直接块(direct block)。对于不大于 10*4096=40KB 的小文件，这就意味着文件的所有的数据块号都被保存在这个结构中。对于较大的文件，就需要其他的地方来保存剩余的文件块号。所以，对于大于 40KB 的文件，我们分配一个额外的磁盘块，称为文件的间接块(indirect block)，来保存最多 4096/4=1024 个额外的磁盘块号。我们的文件系统允许文件最多为 1034 个块 。为了支持更大的文件，真实的文件系统使用双重间接块

(double-indirect blocks)和三重间接块(triple-indirect blocks)。



1.8 目录 VS 普通文件（Directories versus Regular Files）

在我们的文件系统中，一个 File 结构既可以是目录，也可是普通文件；文件的这两种类型由 File 结构中的 type 域区分。文件系统管理普通文件和目录文件使用完全一样的方式，除了它不去解释普通文件中数据块的内容，而将目录文件解释为一系列的描述着文件和子目录的 File 结构体。

超级块包含一个 File 结构体(在 Super 中的 root)，该结构体保存着文件系统的 root 目录的元数据。这个目录文件的内容为一系列的 File 结构，描述着位于 root 目录下的文件和目录。任一位于 root 目录下的子目录可能包含着更多的 File 结构来保存着它自己的子目录。

2 文件系统

本次实验的目的并不是让你实现整个文件系统，但是要实现一些核心模块。特别是你要实现读 block 到 block 缓存中并且写入磁盘；分配磁盘 block；映射文件索引到磁盘 block；并在 IPC 接口中实现读、写、打开。因为你不需要实现整个文件系统，所以要求你自己要熟悉提供的代码和各种文件系统接口。

2.1 磁盘访问（Disk Access）

文件系统需要能够访问磁盘，但是现在还没有任何的磁盘访问函数被实现。我们将 IDE 设备的驱动程序实现为用户级文件系统的一部分，而不是采取传统的将设备驱动和对应的系统调用放置在内核中的模式。我们还需要稍微的修改内核，使得文件系统拥有所需的权限来自己实现磁盘访问。

在用户空间实现磁盘访问是比较简单的，因为我们使用轮询(polling)式的“programmed I/O”（PIO）而不是磁盘中断。在用户模式实现中断驱动的设备驱动程序也是可以的(前几次作业中就是如此)，但是这个难度比较大，因为内核必须能够处理这些中断并将其分配给对应的用户程序。

X86 的处理器使用 EFLAGS 寄存器中的 IOPL 位来决定在保护模式下是否允许产生诸如 IN 和 OUT 之类的特殊设备 I/O 指令。由于所有需要访问的 IDE 磁盘寄存器都位于 x86 处理器的 I/O 映射空间而不是内存映射空间，因此为使文件系统可以访问这些寄存器，我们只需要给文件系统进程赋予“I/O privilege”。实际上，在 EFLAGS 寄存器中的 IOPL 位提供的是一种简单的“all-or-nothing”方式来决定用户级的代码是否可以访问 I/O 空间。在本次试验中，文件系统进程可以访问 I/O 地址空间，而任何其他的进程都不应该访问这部

分空间。处于简化设计的考虑，从现在开始我们设定文件系统进程将永远运行在 environment
1(environment 0 被用于 idle 了)。在下面的测试程序中，如果某个测试失败了，文件
obj/fs/fs.img 就有可能被留存下来。
在运行 make grade 或者 make bochs 之前删除这个文件。

作业 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.

阅读在目录 fs 下的代码树。文件 fs/ide.c 实现了最基本的基于 PIO 的设备驱动程序。文件 fs/serv.c 包含着用于文件系统服务的 umain 函数。

要注意的是，新的.bochsrc 文件设置本次试验中的 Bochs 使用 kern/bochs.img 作为 disk0 的镜像(类似于 Windows 中的 C 盘)，使用新的文件 obj/fs/fs.img 作为 disk1(好比是 D 盘)。本次试验中文件系统只应当访问 disk1；disk0 只用来启动内核。如果你破坏了这两个镜像文件，你可以将各自还原到原始的状态： rm obj/kern/kernel.img obj/fs/fs.img make
或者：

make clean
make

2.2 块缓存（The Block Cache）

在我们的文件系统中，根据处理器的虚拟内存机制，将实现一个简化的“缓存缓冲”(buffer cache)机制，block cache 的代码在 fs/bc.c 中。

文件系统所支持的磁盘大小将被限制在 3GB 以内。我们在文件系统进程的虚拟内存空间中保留一块定长的 3GB 区域，从 0x10000000(DISKMAP)到 0xD0000000(DISKMAP+DISKMAX)，

来映射对应的在内存中的磁盘块。在这个区域中的虚拟内存页中其对应的磁盘块没有被读入内存时就不被映射。例如，当磁盘块 block 0 被读入时，被映射在虚拟地址 0x10000000。磁盘块 block 1 被映射至 0x10001000。可以通过检查 vpt 来确认某个磁盘块是否被映射。

由于我们的文件系统进程拥有与其他的进程完全独立的虚拟地址空间，并且文件系统唯一要做的事情就是实现文件访问，这就使得使用这种方式来保留文件系统进程的地址空间是合理的。当然，在 32 位计算机上真实的文件系统中这样的实现是有问题的，因为现在绝大多数磁盘都大于 3GB 了。不过，这种方式在 64 位的计算机上是可以使用的。

当然，读整个磁盘到 memory 中是不合理的，所以我们实现一个 demand paging 的表单，为了应对缺页错误，我们只在磁盘映射区域分配页，并且从磁盘读相应的 block。通过这种方式，就可以假设整个磁盘在 memory 中了。

作业 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 vpt 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".

2.3 The Block Bitmap

在 fs_init 设置 bitmap 指针之后，我们将 bitmap 看做是一个打了包的 bits 的数组，每一个代表磁盘上的每个 block。例如，block_is_free，只是简单的检查了是否一个给定的 block 在 bitmap 中标定为 free 了。

作业 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.

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

2.4 文件操作（File Operation）

我们已经在文件 fs/fs.c 中提供了一系列的函数来管理和解释 File 结构、分配、寻找指定文件的数据块、扫描和管理目录以及获得文件的绝对路径。仔细阅读 fs/fs.c 中的所有代码，看明白每一个函数的作用。

作业 4

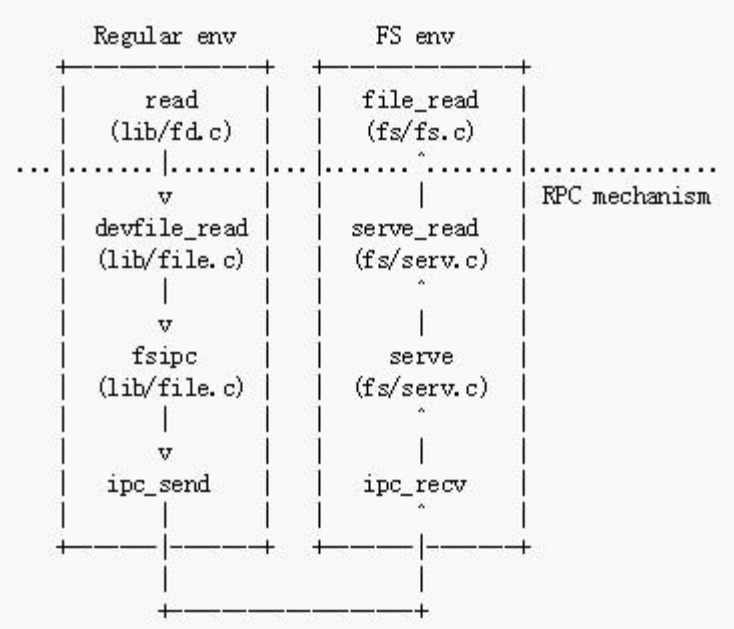
Implement file_block_walk and file_get_block. file_block_walk maps from a block offset within a file to the pointer for that block in the struct File or the indirect block, very much like what pgdir_walk did for page tables. file_get_block goes one step further and maps to the actual disk block, allocating 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".

3 C/S 模式的文件系统访问（Client/Server File System Access）

我们已经为文件系统服务器实现了必须的函数，现在必须使得其他要使用文件系统的进程也可以使用它们。这主要要修改两部分的代码：客户端和服务器端。它们要共同实现一个远过程调用(remote procedure call, RPC)，这样使得通过 IPC 通信过程的地址空间传递看上去像是位于客户端应用内部的普通 C 函数调用。

下面是一个对文件系统 server 端的调用（或者是读）：



虚线下的所有是一个简单的从普通环境读 request 到文件系统环境的机制。在最开始，在任意文件描述符上读然后简单的转换到合适的设备读函数，在这种情况下是函数 devfile_read。devfile_read 实现了磁盘文件的读。这个以及在 lib/file.c 中的其他的 devfile_*函数实现了 FS 操作的客户端，并且所有的工作都是以相同的方式，在 request 结构体中建立一个参数，调用 fsipc 来发送 IPC 请求，解析并返回结果。Fsipc 函数简单的控制了发送一个请求到 server 和接受一个应答的一般细节。

文件系统 server 端的代码可以在 fs/serv.c 中找到。他在 server 函数中循环，通过 IPC 无限接受请求，解析请求到合适的 handler 函数，并且通过 IPC 发回结果。在“读”的例子中，server 将会解析到 serve_read，它将管理 IPC 的细节，特别是像解析请求结构、调用 file_read 来实际执行读文件等读请求。

回想 jos 的 IPC 机制，让一个进程发送一个单一的 32 位的数，并且可以共享一个页。为了发送一个从 client 到 server 的请求，我们使用这个 32 位的数作为 request 的类型，并且储存一个参数到通过 IPC 共享的页的联合 Fsipc 中的请求中。在 client，我们经常在 fsipcbuf 共享页；在 server，我们映射到达的 request 页在 fsreq（0x0ffff000）。

Server 也会通过 IPC 返回应答。我们使用这个 32 位的数作为函数的返回码。对多数 RPC 而言，这是他们返回的全部。FSREQ_READ 和 FSREQ_STAT 也会返回数据，他们简单的写到 client 发送的请求所在的页中。没必要将整个页放在应答 IPC，因为 client 与 server 共享它。同时在它的应答中，FSREQ_OPEN 和 client 共享一个新的“Fd page”。

作业 5

Implement serve_read in fs/serv.c and devfile_read in lib/file.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.

Likewise, devfile_read should pack its arguments into fsipcbuf for serve_read, call fsipc, and handle the result.

Use make grade to test your code. Your code should pass "lib/file.c" and "file_read".

作业 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" and "file_read after file_write".

3.1 客户进程的文件系统访问（Client-Side File Operations）

在 lib/file.c 中的函数是针对 on-disk 的文件的，但是 UNIX 文件描述符是一个围绕 pipe, console I/O 等的更一般的概念。在 JOS 中，每个设备类型有一个对应的结构 Dev，含有指向函数的指针，这些为这个设备类型函数实现了 read/write 等。lib/fd.c 实现了一般的 UNIX-like 的文件描述符接口。每个结构 Fd 代表了它的设备类型，在 lib/fd.c 中的许多函数仅简单地为函数解析一些操作。

lib/fd.c 也在应用的进程地址空间中维护了 file descriptor table 区域，从 FSTABLE 开始。这个区域保存了地址空间一页的内容（4kb），对每个可达 MAXFD 地址空间（目前为 32）的应用程序可以同时打开的文件描述符。在任意给定时间，一个特别的 file descriptor table page 被映射，当且仅当相应的文件描述符在使用。每个文件描述符在从 FILEDATA 开始的区域也有一个可选的“data page”，尽管我们在本实验中不会用到。

对几乎所有文件的交互，用户代码都需要阅读 lib/fd.c 中的函数。Lib.file.c 中的一个 public 的函数是 open 的，它通过打开一个所谓的 on-disk file 构建一个新的文件描述符。

作业 7

Implement open. The open function must find an unused file descriptor using the fd_alloc() function we have provided, make an IPC request to the file system environment to open the file, and return the number of the allocated file descriptor. Be sure your code fails gracefully if the maximum number of files are already open, or if any of the IPC requests to the file system environment fail.

Use make grade to test your code. Your code should pass "open", "large file", "motd display", and "motd change".

3.2 Spawning Processes

已经提供给你了 spawn 的代码，它创建了一个新的进程，从文件系统加载一个程序镜像，然后启动子进程来运行这个程序。然后父进程继续独立于子进程运行。Spawn 函数的角色就像 UNIX 中的一个 folk，它可以在子进程中立刻退出。

我们实现 spawn 而不是一个 UNIX 风格的 exec 因为 spawn 更容易实现，不需要内核的特殊帮助。考虑你将如何来实现用户空间的 exec。

作业 8

spawn relies on the new syscall sys_env_set_trapframe to initialize the state of the newly created environment. Implement sys_env_set_trapframe. Test your code by running the user/icode program from kern/init.c, which will attempt to spawn /init from the file system. Use make grade to test your code.

4 作业要求

4.1 提交内容

- ✧ 修改后的源码
- ✧ PPT 展示：ppt 内容与本门课程相关并且不限于本次作业（秀出你的风采！）展示时长为 5 分钟

4.2 提交方式

- ✧ 当面提交，提交地点：计控学院楼 427
- ✧ 提交要求：和以往提交方式相同
- ✧ 作业说明
  - ◆ 本次作业分数总分为 10 分，直接加在总成绩上（所有 make grade 通过才视为有效，否则直接为 0 分）
  - ◆ PPT 展示
  - ◆ 本次作业为 challenge，不提供指导

4.3 提交时间

本次作业于 2015 年 12 月 30 日发布，放假（2016、1、18）前的周六、日当面提交。