Lab 5: 文件系统、Spawn、Shell

Lab 5 练习思路

本实验将会实现 `spawn` ，一个加载和运行磁盘可执行文件的库调用。然后，内核和库操作系统将会充分运行，以便在控制台上运行shell。上述特性首先需要一个文件系统，本实验将介绍一个简单的读写文件系统。

第一部分 文件系统预备

你将使用的文件系统比大多数真实文件系统简单得多，提供基本功能：创建，读取，写入和删除分层目录结构中组织的文件。

我们目前只开发单用户操作系统，它提供足够的保护来捕捉错误，但不能保护多个用户，因此不支持文件所有权的概念。我们的文件系统目前还不支持硬链接、符号链接、时间戳或特殊的设备文件等。

1. 磁盘文件系统结构

大多数UNIX文件系统将可用磁盘空间分为两种主要类型的区域：inode区域和数据区域。

UNIX文件系统为每个文件分配一个inode，文件的inode保存关于文件的关键性元数据，例如其 `stat` 属性和指向其数据块的指针。

数据区域被划分成更大（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。目录条目包含文件名和指向inode的指针，如果文件系统中的多个目录条目引用该文件的inode，则文件被称为硬链接。由于我们的文件系统不支持硬链接，因此可以简化为：我们的文件系统根本不会使用inode，而只是存储所有的文件（或子目录）描述该文件的（唯一）目录条目中的元数据。

文件和目录逻辑上都包含一系列数据块，这些数据块可能散布在整个磁盘上，就像环境的虚拟地址空间的页面可以分散在整个物理内存中一样。文件系统环境隐藏块布局的细节，呈现用于读取和写入文件中任意偏移量的字节序列的接口。文件系统环境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 `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），而不必依赖额外的文件系统调用。目标扫描的这种方法的缺点以及大多数现代UNIX变体取消它的原因在于，它使应用程序依赖于目录元数据的格式，使得更改文件系统的内部布局而不改变或至少重新编译应用程序很难实现。

2. 扇区和块

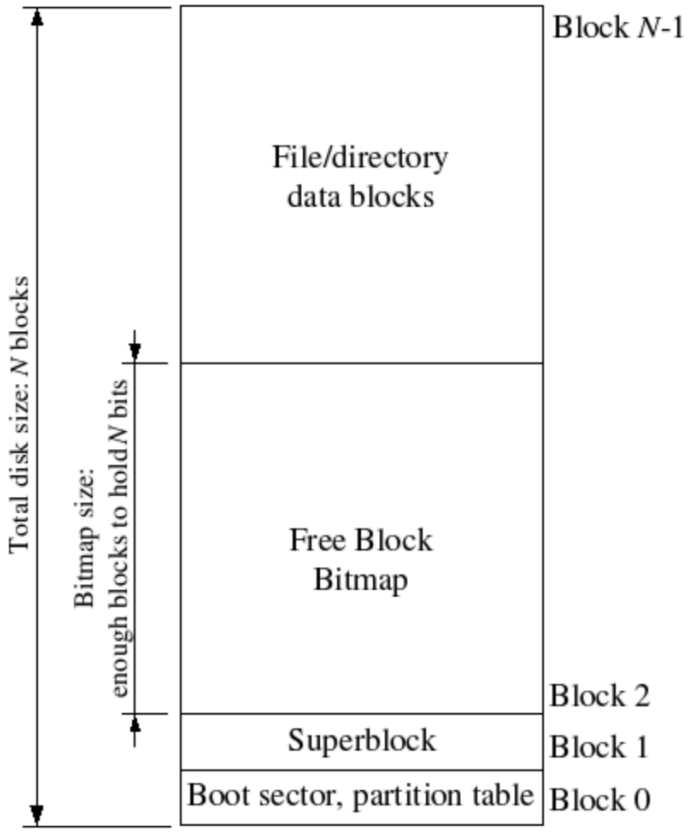
大多数磁盘不以字节为粒度执行读取和写入，而是以扇区为单位执行读取和写入操作。在JOS中，扇区为512字节。文件系统实际上以块为单位分配和使用磁盘存储。要注意两个术语之间的区别：扇区大小是磁盘的属性 硬件，而块大小是操作系统使用磁盘的一种情况。文件系统的块大小必须是底层磁盘扇区大小的倍数。

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。然而，大多数现代文件系统使用较大的块大小，因为存储空间已经变得更便宜，并且以更大的粒度来管理存储更有效。 我们的文件系统将使用4096字节的块大小，方便地匹配处理器的页面大小。

3. 超级块

文件系统通常将某些磁盘块保留在磁盘上易于查找的位置（例如最初或最后），以保存描述整个文件系统的属性的元数据，例如块大小、磁盘大小、找到根目录所需的任何元数据、文件系统上次装载的时间、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。

我们的文件系统只有一个超级块，它们将始终位于磁盘上的块1。它的布局由 `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块0。许多真实文件系统维护多个超级块，复制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发生媒体错误，则仍可以找到其他超级块，并将其用于访问文件系统。

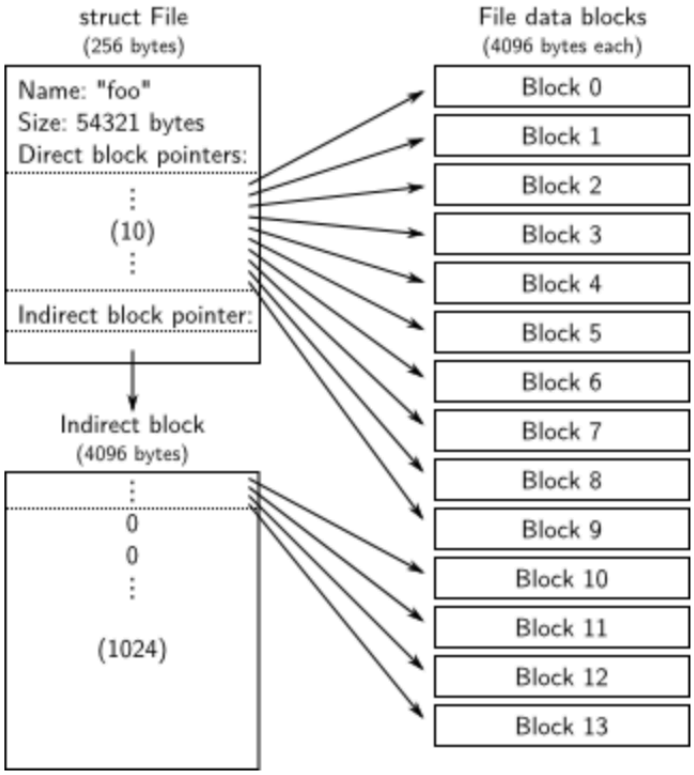


4. 文件元数据

文件系统中文件元数据的布局由 `inc/fs.h` 中的 `struct File` 描述。该元数据包括文件名称、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，所以这个元数据存储在磁盘上的目录条目中。与大多数真实文件系统不同，为简单起见，我们将使用这一个文件结构来表示文件元数据，因为它们在磁盘和内存中都有。

`struct File` 中的 `f_direct` 数组包含一部分存储文件前10（NDIRECT）个块的块号的空间，我们称

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文件结构本身。然而，对于较大的文件，我们需要一个地方来保存文件的块号的其余部分。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，以容纳最多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，或者超过4MB。为了支持更大的文件，真实文件系统通常也支持双重和三重间接块。



5. 目录与常规文件

文件系统中的文件结构可以表示常规文件或目录。这两种类型通过 `File` 中的 `type` 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，除了一点：文件系统不解释与常规文件相关联的数据块的内容，但将目录文件的内容解释为一系列 `struct File` ，描述目录中的文件和子目录。

我们的文件系统中的超级块包含一个 `struct File` （ `struct Super` 中的 `root` 字段），保存文件系统根目录的元数据。此目录文件的内容是一系列描述文件和在文件系统的根目录下目录结构的 `struct File` 。根目录中的任何子目录可能依次包含表示子子目录的更多文件结构，依此类推。

第二部分 文件系统

本次实验仅实现关键组件。例如，将块读入块Cache并将其刷新返回磁盘、分配磁盘块、将文件偏移映射到磁盘块、在IPC界面中实现读写和打开。因为并不会实现所有的文件系统，所以首先熟悉已经写好的代码和各种文件系统界面是非常重要的。

1. 磁盘访问

操作系统中的文件系统环境必须能够访问磁盘，但是我们还没有在内核中实现任何磁盘访问功能。我们需要将IDE磁盘驱动程序实现为用户级文件的一部分系统环境，而不是将常规的单片操作系统策略添加到内核中的IDE磁盘驱动程序以及必要的系统调用以允许文件系统访问它。我们仍然需要稍微修改内核，以便设置文件，使文件系统环境具有实现磁盘访问所需的权限。

只要我们依靠轮询、基于编程I/O（PIO）的磁盘访问并且不使用磁盘中断，就很容易在用户空间中实现磁盘访问。在用户模式下实现中断驱动的设备驱动（例如L3和L4内核）是可能的，但是由于内核必须检测设备中断并将其发送到正确的用户模式环境，所以比较困难。

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器件IO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IO空间中，而不是内存映射，因此为文件系统环境提供“IO权限”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位提供了内核 使用简单的“有或无”方法来控制用户模式代码是否可以访问IO空间。在我们的例子中，我们希望文件系统环境能够访问IO空间，但是我们不希望任何其他环境能够访问IO空间。

> **练习一：**
>
> `i386_init` 通过将 `ENV_TYPE_FS` 类型传递给的环境创建函数 `env_create` 识别文件系统环境。在 `env.c` 中修改 `env_create` ，以便它授予文件系统环境IO权限，但永远不会赋予任何其他环境。
>
> 确保可以启动文件环境，而不导致常规保护错误。应该 可以通过 `make grade` 中的 `fs i/o` 测试。
>
> 我的思路

2. 块Cache

我们的文件系统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Cache（块缓存）。块高速缓存的代码位于 `fs/bc.c` 中。

我们的文件系统仅限于处理大小不超过3GB的磁盘。我们保留一个固定的3GB的文件系统环境的地址空间，从0x10000000（ `DISKMAP` ）到0xD0000000（ `DISKMAP + DISKMAX` ），作为磁盘的内存映射版本。例如，磁盘块0映射到虚拟地址0x10000000，磁盘块1映射到虚拟地址0x10001000，依此类推。 `fs/bc.c` 中的 `diskaddr` 功能实现了从磁盘块号到虚拟地址的转换（以及一些合理检查）。

由于我们的文件系统环境具有独立的虚拟地址空间，与系统中所有其他环境的虚拟地址空间无关，文件系统环境所需要做的只是实现文件访问，因此保留大部分文件系统环境的这个地址空间是合理的。由于现代磁盘大于3GB，因此在32位计算机上实现真正的文件系统实现将会非常尴尬。在具有64位地址空间的机器上，这种缓冲区高速缓存管理方法可能仍然是合理的。

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请求分页的形式，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地区的页面错误。这样，我们可以假装整个磁盘都在内存中。

> **练习二：**
>
> 在 `fs/bc.c` 中实现 `bc_pgfault` 和 `flush_block` 函数。 `bc_pgfault` 是一个页面错误处理程序，就像在之前的实验中用于写入副本的代码所写的那样，除了它的任务是从磁盘加载页面以响应页面错误。
>
> 完成代码时，请记住 `addr` 可能不与块边界对齐，且 `ide_read` 以扇区为单位而不是块。
>
> 如果需要， `flush_block` 函数应该将一个块写入磁盘。如果块缓存中的块不均匀（即页面未映射），或者如果不脏，则 `flush_block` 不应该执行任何操作。我们将使用VM硬件来跟踪磁盘块是否已被上次读取或写入磁盘后被修改。要查看块是否需要写入，我们可以查看是否在 `uvpt` 条目中设置了 `PTE_D` dirty位。（ `PTE_D` 位由处理器响应于该页面的写入而设置）。将块写入磁盘后， `flush_block` 应使用 `sys_page_map` 清除 `PTE_D` 位。
>
> 使用 `make grade` 测试代码，应该可以通过 `check_bc` `check_super` 和 `check_bitmap` 。
>
> 我的思路

`fs/fs.c` 中的 `fs_init` 函数是如何使用块Cache的主要示例。在初始化块Cache之后，它将指针存储在全局变量 `super` 的磁盘映射区域中。 此后可以直接从 `super` 中读取，页面错误处理程序将根据需要从磁盘读取它们。

3. 块位图

在 `fs_init` 设置位图指针之后，我们可以将位图视为位数组，每一项都用于磁盘上的每个块。例如 `block_is_free` 只是检查位图中给定的块是否被标记为空闲。

> **练习三：**
>
> 以 `free_block` 为模型实现 `fs/fs.c` 中的 `alloc_block` ，它从位图中找到可用的磁盘块，标记它为被使用，并返回该块的编号。分配块时，应立即把 `flush_block` 更改的位图刷新到磁盘，以保持文件系统一致性。
>
> 用 `make grade` 测试代码。 现在应该可以通过 `alloc_block` 。
>
> 我的思路

4. 文件操作

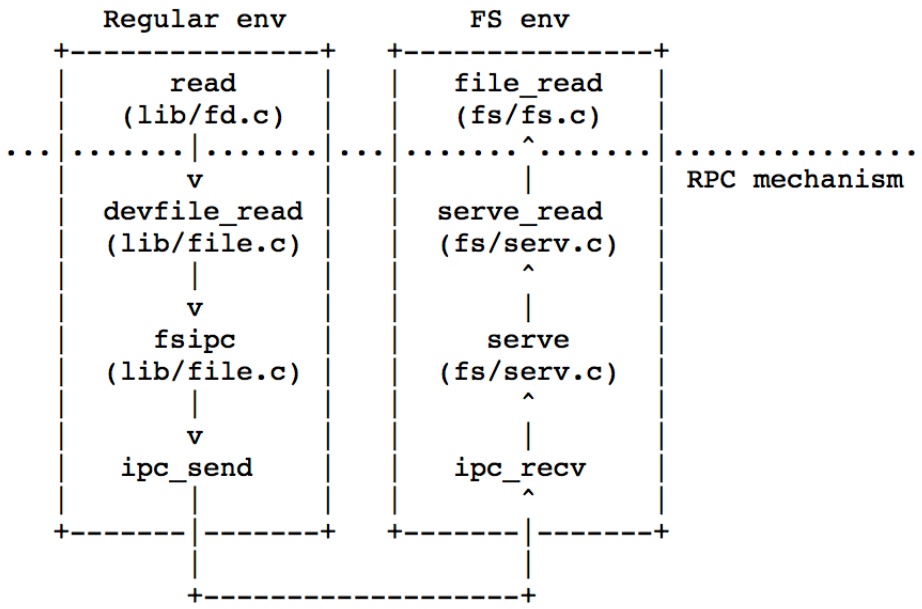
我们在 `fs/fs.c` 中提供了解释和管理文件结构、扫描和管理目录文件条目、从根目录中移走文件系统以解决绝对路径名等基本功能。阅读 `fs/fs.c` 代码，确保了解每个函数执行的操作。

> **练习四：**
>
> 实现 `file_block_walk` 和 `file_get_block` 。 `file_block_walk` 从文件中的块偏移量映射到 `struct File` 或间接块中的该块的指针，类似 `pgdir_walk` 对页表执行的操作。 `file_get_block` 进一步映射到实际的磁盘块，如有必要，分配一个新的磁盘块。
>
> 用 `make grade` 测试代码。应该可以通过 `file_open` `file_get_block` `file_flush/file_truncated/file_rewrite` 和 `testfile` 。
>
> 我的思路

`file_block_walk` 和 `file_get_block` 是文件系统的主角。例如， `file_read` 和 `file_write` 比在分散块和顺序缓冲区之间复制字节所需的 `file_get_block` 顶部的簿记更多。

5. 文件系统界面

文件系统环境本身已经具有了基本功能，我们必须使其他希望使用文件系统的环境也可以访问。由于其他环境不能直接调用文件系统环境中的函数，我们将通过远程过程调用或JOS IPC机制上构建的RPC抽象来公开对文件系统环境的访问。以下是对文件系统服务器的调用（如read）的图形表示：



虚线以下的所有内容都只是从常规环境到文件系统环境的读取请求的机制。最开始， `read` （我们提供的）可以在任何文件描述符上工作，并且简单地调度到适当的设备读取功能，在JOS下为 `devfile_read` （我们可以有更多的设备类型，如管道）。 `devfile_read` 专门为磁盘文件进行读取。 `lib/file.c` 中的这个和其他 `devfile_*` 函数实现了FS操作的客户端，并且所有这些都以大致相同的方式工作，在请求结构中绑定参数，调用 `fsipc` 发送IPC请求，并解包和返回结果。 `fsipc` 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。它在服务功能中循环，无休止地接收到IPC的请求，将该请求发送到适当的处理程序功能，并通过IPC发送结果。在阅读示例中，服务将发送到 `serve_read` ，这将照顾IPC详细信息来读取请求，如解包请求结构，最后调用 `file_read` 来实际执行文件读取。

回想一下，JOS的IPC机制允许环境发送一个32位数字，并且可选地共享一个页面。要将请求从客户端发送到服务器，我们使用32位数字作为请求类型（文件系统服务器RPC编号，就像系统调用编号一样），并将参数存储在 `union Fsipc` 中通过IPC共享页面。在客户端，我们总是在 `fsipcbuf` 共享页面;在服务器端，我们将传入请求页面映射到 `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。 `FSREQ_READ` 和 `FSREQ_STAT` 也返回数据，它们只是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它。此外，在其回复中， `FSREQ_OPEN` 与客户共享一个新的“Fd页面”。我们将很快返回到文件描述符页面。

> **练习五：**
>
> 在 `fs/serv.c` 中实现 `serve_read` 。
>
> 通过 `fs/fs.c` 中已经实现的 `file_read` （其实只是一系列对 `file_get_block` 的调用）， `serve_read` 的完成比较简单。 `serve_read` 只需提供RPC接口进行文件读取。查阅 `serve_set_size` 中的注释和代码，以了解如何构建服务器功能的一般概念。
>
> 用 `make grade` 测试代码。 应该可以通过 `serve_open/file_stat/file_close` 和 `file_read` ，获得70分。
>
> 我的思路

> **练习六：**
>
> 实现 `fs/serv.c` 中的 `serve_write` 和 `lib/file.c` 中的 `devfile_write` 。
>
> 用 `make grade` 测试代码。 应该可以通过 `file_write` `file_read after file_write` `open` 和 `large file` ，获得90分。
>
> 我的思路

第三部分 Spawning过程

我们已经给你提供了 `spawn` 代码（参阅 `lib/spawn.c` ），这部分代码生成一个新环境、将文件系统中的程序映像加载到其中、之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` 函数在UNIX中像一个 `fork` 一样有效，后面紧跟着子进程中的一个 `exec` 。

我们实现了 `spawn` 而不是一个UNIX风格的 `exec` ，因为没有内核的特殊帮助下 `spawn` 更容易从用户空间实现。思考在实现 `exec` 时必须做什么，并确保你明白为什么这么做难度更大。

> **练习七：**
>
> `spawn` 依赖于新的系统调用 `sys_env_set_trapframe` 来初始化新创建的环境的状态。在 `kern/syscall.c` 中实现 `sys_env_set_trapframe` （不要忘记在 `syscall()` 中作处理）。
>
> 通过在 `kern/init.c` 中运行 `user/spawnhello` 程序来测试代码，该程序将尝试从文件系统中生成 `/hello` 。
>
> 用 `make grade` 测试代码。
>
> 我的思路

fork和spawn共享库状态

UNIX文件描述符是一个普遍的概念，也包括管道，控制台IO等。在JOS中，这些设备类型中的每一个都具有相应的 `struct Dev` ，为该设备类型指向实现读写等功能的函数。 `lib/fd.c` 实现了一般的类似UNIX的文件描述符接口。每个 `struct Fd` 表示其设备类型，并且 `lib/fd.c` 中的大多数函数只是将操作分配给适当的 `struct Dev` 。

`lib/fd.c` 还在每个应用程序环境的地址空间中维护一个文件描述符表区域，地址始于 `FDTABLE` 。该区域为应用程序可以一次打开的最多 `MAXFD` （当前为32）个文件描述符保留一个页（4KB）的地址空间。在任何给定的时间，当且仅当相应的文件描述符被使用时，特定的文件描述符表页才会被映射。
在 `FILEDATA` 开始的区域中，每个文件描述符还有一个可选的数据页，如果选择了，设备就可以使用。

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。现在在 `fork` 上，内存将被标记为COW，所以状态将被复制而不是共享，这意味着环境将无法寻找没有将自己打开的文件，所以管道不能在 `fork` 中不能正常运行。而在 `spawn` 中，内存将被忽略，不会被复制。（有效地，产生的环境始于没有打开的文件描述符。）

我们将更改 `fork` ，以了解某些区域的内存是由“库操作系统”使用，应始终共享。而不是硬编码某个部分的列表，我们将在页表条目中设置一个未使用的位（就像我们在 `fork` 中使用 `PTE_COW` 位一样）。

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“可用于软件使用”的三个PTE位之一。我们将建立约定，如果一个页表条目设置了这个位，那么PTE应该在 `fork` 和 `spawn` 中直接从父环境复制到子环境。需要注意的是，这与将它标记为COW是不同的：如第一段所述，我们要确保页面更新也是共享的。

练习八：

在 `lib/fork.c` 中更改 `duppage` 以遵循新的约定。如果页表项已经设置了 `PTE_SHARE` 位，则只需直接复制映射。（您应该使用 `PTE_SYSCALL` 而不是 `0xfff` 来屏蔽页表条目中的相关位， `0xfff` 也会拾取已访问的为和脏位）。

同样，在 `lib/spawn.c` 中实现 `copy_shared_pages` 。该函数循环遍历当前进程中的所有页表项（类似 `fork` ），将具有 `PTE_SHARE` 位的任何页面映射复制到子进程中。

我的思路

第四部分 键盘界面

为了使shell工作，我们需要一种方法来键入它。QEMU一直在显示我们写入CGA显示和串行端口的输出，但到目前为止，我们只在内核监视器中输入了。在QEMU中，在图形窗口中键入的输入显示为从键盘输入到JOS，而输入到控制台的输入显示为串行端口上的字符。 `kern/console.c` 已经包含自Lab 1以来内核监视器使用的键盘和串行驱动程序，但现在需要将它们附加到系统的其余部分。

练习九：

在你的 `kern/trap.c` 中，调用 `kbd_intr` 来处理 `IRQ_OFFSET + IRQ_KBD` 陷入，调用 `serial_intr` 来处理 `IRQ_OFFSET + IRQ_SERIAL` 陷入。

我的思路

`lib/console.c` 中已经实现了控制台输入/输出文件类型。当控制台文件类型排除缓冲区时， `kbd_intr` 和 `serial_intr` 会填充一个带最近读取的输入缓冲区（默认情况下，控制台文件类型用于 `stdin/stdout` ，除非用户重定向它们）。

通过运行 `make run-testkbd` 并输入几行来测试代码。完成它们后，系统应该回应你的行。尝试在控制台和图形窗口中键入。

第五部分 Shell

运行 `make run-icode` 运行内核并启动 `user/icode` 。 `icode` 执行 `init` ，将控制台设置为文件描述符0和1（即标准输入和标准输出）。然后它会生成 `sh` ，即shell。应该可以运行以下命令：

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

需要注意的是，用户库例程 `cprintf` 直接打印到控制台，而不使用文件描述符代码。这对于调试很有帮助，但对于其他程序的管道来说不是很好。要将输出打印到特定文件描述符（例如1，标准输出），应当使用 `fprintf(1, .., ..)` 。 `printf(.., ..)` 则是打印到FD 1的快捷方式。有关示例，参阅 `user/lsfd.c` 。

> **练习十：**
>
> shell不支持IO重定向。运行 `sh <script` 而不是手动输入脚本中的所有命令比较合理。在 `user/sh.c` 中为 `<` 添加IO重定向。
>
> 通过在你的shell中输入 `sh <script` 来测试。
>
> 运行 `make run-testshell` 测试shell。 `testhell` 只是将上述命令（可以在 `fs/testshell.sh` 中找到）输入到shell中，然后检查输出是否匹配 `fs/testshell.key` 。
>
> 我的思路

练习思路

练习一：

这个练习比较简单，只要在 `env_create` 里赋予文件系统环境权限即可，添加代码比较简单，如下：

```
454     if (e->env_type == ENV_TYPE_FS){
455         e->env_tf.tf_eflags |= FL_IOPL_MASK;
456     }
```

练习二：

这个练习也比较简单，题目中已经清楚地给出代码补全的步骤：

- `bc_pgfault` ：先把 `addr` 与 `PGSIZE` 对齐，然后先分配一页，然后从磁盘中读取，增加代码如

下：

```
45      // Allocate a page in the disk map region, read the contents
46      // of the block from the disk into that page.
47      // Hint: first round addr to page boundary. fs/ide.c has code to read
48      // the disk.
49      //
50      // LAB 5: you code here:
51      void *raddr = ROUNDDOWN(addr, PGSIZE);
52      envid_t curenvid = sys_getenvid();
53      uint32_t secno = ((uint32_t)raddr - DISKMAP) / SECTSIZE;
54
55      sys_page_alloc(curenvid, raddr, PTE_SYSCALL);
56      ide_read(secno, raddr, BLKSECTS);
```

- `flush_block`：先把 `addr` 与 `PGSIZE` 对齐，然后如果页面未映射或者不脏则不做操作，否则写入磁盘，最后根据提示用 `PTE_SYSCALL` 清除 `PTE_D` 位，增加代码如下：

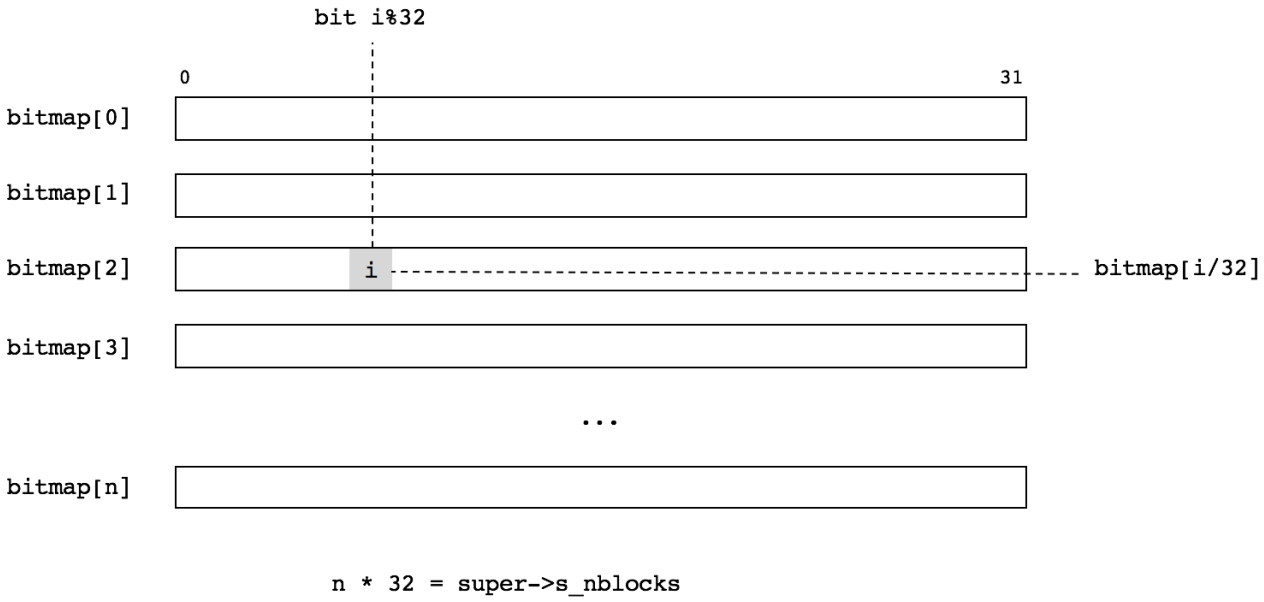
```
85      // LAB 5: Your code here.
86      void *raddr = ROUNDDOWN(addr, PGSIZE);
87      envid_t curenvid = sys_getenvid();
88      uint32_t secno = ((uint32_t)raddr - DISKMAP) / SECTSIZE;
89
90      if (!(!va_is_mapped(addr) || !va_is_dirty(addr))) {
91          ide_write(secno, addr, BLKSECTS);
92          sys_page_map(curenvid, addr, curenvid, addr, PTE_SYSCALL);
93      }
```

`ide_read()` 和 `ide_write()` 的用法可以参阅 `ide.c`，参数传递在注释中也有提示。

练习三：

这一练习思路比较简单，但是实现上还是得稍微思考一下。首先题目或者注释中已经很详细地说明了函数的过程：它从位图中找到可用的磁盘块，标记它为被使用，并返回该块的编号，最后立即 `flush_block` 刷新。找空闲磁盘块可以用已经写好的 `block_is_free` 判断，外边嵌套一个循环体即可。关键性问题就在于如何把位图中对应的这一项修改为占用，我对此理解如下：

根据注释提示可以参照一下 `free_block` 中对位图的处理，看过以后我认为位图 `bitmap` 可以看作是若干个32位字组成的一个数组。如下图所示，当遍历到第 `i` 块时，对应的位图项为 `bitmap[i/32]`，并且是项中的第 `i%32` 位，所以参照 `free_block` 处理方法，只要先把 `1` 左移 `32-i` 位，然后与位图项按位与即可。



最后将块号 `i` 传递给 `flush_block` 刷新磁盘块。代码如下：

```
57 int
58 alloc_block(void)
59 {
60     // The bitmap consists of one or more blocks.  A single bitmap block
61     // contains the in-use bits for BLKBITSIZE blocks.  There are
62     // super->s_nblocks blocks in the disk altogether.
63
64     // LAB 5: Your code here.
65     for (int i = 0; i < super->s_nblocks; i++) {
66         if (block_is_free(i)) {
67             bitmap[i / 32] &= (1 << ((32 - i) % 32));
68             flush_block(diskaddr(i % 32));
69             return i;
70         }
71     }
72     return -E_NO_DISK;
73 }
```

练习四：

这个练习也比较容易，按照注释中所给的多种情况进行一一列举，然后做相应的处理就好了， `struct File` 的各个域的含义以及为0的意思在 `inc/fs.h` 中描述比较详细。 `fs.c` 中两个函数代码部分如下：

```
140 static int
141 file_block_walk(struct File *f, uint32_t filebno, uint32_t **ppdiskbno, bool alloc)
142 {
143     // LAB 5: Your code here.
144     int ret = 0;
145
146     if (filebno >= NDIRECT + NINDIRECT)
147         return -E_INVAL;
148
149     if (filebno >= NDIRECT) {
150         if (f->f_indirect == 0) {
151             if (!alloc)
152                 return -E_NOT_FOUND;
153             else {
154                 ret = alloc_block();
155                 if (ret < 0)
156                     return -E_NO_DISK;
157
158                 f->f_indirect = ret;
159                 memset(diskaddr(ret), 0, BLKSIZE);
160             }
161         }
162         *ppdiskbno = &((uintptr_t *) diskaddr(f->f_indirect))[filebno - NDIRECT];
163     } else
164         *ppdiskbno = &f->f_direct[filebno];
165
166     return ret;
167 }
```

```
177 int
178 file_get_block(struct File *f, uint32_t filebno, char **blk)
179 {
180     // LAB 5: Your code here.
181     int ret, bno;
182     uint32_t *ppdiskbno;
183
184     ret = file_block_walk(f, filebno, &ppdiskbno, true);
185     if (ret < 0)
186         return ret;
187     else {
188         if (*ppdiskbno == 0) {
189             bno = alloc_block();
190             if (bno < 0)
191                 return -E_NO_DISK;
192
193             *ppdiskbno = bno;
194         }
195
196         *blk = (char *)diskaddr(*ppdiskbno);
197     }
198
199     return 0;
200 }
```

练习五：

这一练习的代码补充也比较简单，调用函数在题目和 `serve_set_status` 中都有提示，先读入文件，然后读取其中的字节。涉及到的结构体主要是 `OpenFile` 和 `Fsreq_read` `Fsret_read`，分别可以在 `fs/serv.c` 和 `inc/fs.h` 中找到定义。只要按照函数的参数含义去传递合理的参数即可。代码如下：

```
207 int
208 serve_read(envid_t envid, union Fsipc *ipc)
209 {
210     struct Fsreq_read *req = &ipc->read;
211     struct Fsret_read *ret = &ipc->readRet;
212     int r;
213     struct OpenFile *of;
214
215     if (debug)
216         cprintf("serve_read %08x %08x %08x\n", envid, req->req_fileid, req->req_n);
217
218     // Lab 5: Your code here:
219     r = openfile_lookup(envid, req->req_fileid, &of);
220     if (r < 0)
221         return r;
222
223     r = file_read(of->o_file,
224                   ret->ret_buf,
225                   req->req_n,
226                   of->o_fd->fd_offset);
227     if (r >= 0)
228         of->o_fd->fd_offset += r;
229     return r;
230 }
```

练习六：

第一个函数 `serve_write` 和上一练习的 `serve_read` 道理完全一样，按照函数 `file_write` 作修改即可。代码如下：

```
237 int
238 serve_write(envid_t envid, struct Fsreq_write *req)
239 {
240     if (debug)
241         cprintf("serve_write %08x %08x %08x\n", envid, req->req_fileid, req->req_n);
242
243     // LAB 5: Your code here.
244     int r;
245     struct OpenFile *of;
246
247     r = openfile_lookup(envid, req->req_fileid, &of);
248     if (r < 0)
249         return r;
250
251     r = file_write(of->o_file,
252                    req->req_buf,
253                    req->req_n,
254                    of->o_fd->fd_offset);
255     if (r >= 0) {
256         of->o_fd->fd_offset += r;
257     }
258     return r;
259 }
```

第二个函数 `devfile_write` 是具体的实现版本（目前没搞懂和 `file_write` 的差异，对流程不太熟悉，后期再做深入分析），因为实际要从缓冲读取的的不一定正好是 `n` 个字节，所以要先判断二者的大小，选择小的做处理。这部分代码直接看了结果，比较容易理解，代码如下：

```
136 static ssize_t
137 devfile_write(struct Fd *fd, const void *buf, size_t n)
138 {
139     // Make an FSREQ_WRITE request to the file system server.  Be
140     // careful: fsipcbuf.write.req_buf is only so large, but
141     // remember that write is always allowed to write *fewer*
142     // bytes than requested.
143     // LAB 5: Your code here
144
145     int r;
146
147     r = MIN(n, sizeof(fsipcbuf.write.req_buf));
148     fsipcbuf.write.req_fileid = fd->fd_file.id;
149     fsipcbuf.write.req_n = r;
150     memmove(fsipcbuf.write.req_buf, buf, r);
151
152     return fsipc(FSREQ_WRITE, NULL);
153 }
```

练习七：

这个练习比较简单，按照注释和题目要求对 `syscall.c` 进行修改即可，除 `syscall()` 中再添加一组 `case` 外，函数 `sys_env_set_trapframe` 部分代码如下：

```
141 static int
142 sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
143 {
144     // LAB 5: Your code here.
145     // Remember to check whether the user has supplied us with a good
146     // address!
147     struct Env *env_store;
148
149     envid2env(envid, &env_store, 1);
150     if (env_store == NULL)
151         return -E_BAD_ENV;
152
153     env_store->env_tf = *tf;
154     return 0;
155
156 }
```

练习八：

根据题意，这一问需要在两个部分进行修改，一是 `fork.c` 的 `duppage` ，另一个是 `lib/spawn.c` 的 `copy_shared_pages` 。

对于前者，只需按照题意对其进行修改即可：如果页表项已经设置了 `PTE_SHARE` 位，则只需直接通过 `sys_page_map` 复制映射，无需进行多余的步骤，同时要设置 `PTE_SYSCALL` 位屏蔽相关位（这里不太理解）。添加之后整体代码如下：

```
60 static int
61 duppage(envid_t envid, unsigned pn)
62 {
63     int r;
64     int perm = PTE_U | PTE_P;
65     envid_t curenvid = sys_getenvid();
66
67     // LAB 4: Your code here.
68     if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW))
69         perm |= PTE_COW ;
70
71     if (uvpt[pn] & PTE_SHARE)
72         perm |= PTE_SYSCALL;
73
74     r = sys_page_map(curenvid, (void *)(pn * PGSIZE),
75                 envid, (void *)(pn * PGSIZE), perm);
76     if (r < 0)
77         panic("duppage error: %e", r);
78
79     if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)) {
80         r = sys_page_map(curenvid, (void *)(pn * PGSIZE),
81                         curenvid, (void *)(pn * PGSIZE), perm);
82         if (r < 0)
83             panic("duppage error: %e", r);
84     }
85
86     return 0;
87 }
```

对于后者，题目已经提示到，与 `fork` 类似地去遍历所有页表项，然后把设置为 `PTE_SHARE` 的页面映射到子进程。这里可以对比一下 `fork` 中的判断：

```
122         for (uint32_t addr = UTEXT; addr < USTACKTOP; addr += PGSIZE) {
123             if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
124                 duppage(new_envid, PGNUM(addr));
125         }
```

`fork` 中进行的操作是将所有页面复制到子环境，而 `spawn` 是将设置为 `PTE_SHARE` 的页面映射到子环境，所以一方面判断上要添加一个条件，另一方面是通过 `sys_page_map` 映射环境。所以最终代码如下：

```
300 static int
301 copy_shared_pages(envid_t child)
302 {
303     // LAB 5: Your code here.
304     envid_t curenvid = sys_getenvid();
305
306     for (int i = 0; i < UTOP; i += PGSIZE) {
307         if ((uvpd[PDX(i)] & PTE_P) &&
308             (uvpt[PGNUM(i)] & PTE_P) &&
309             (uvpt[PGNUM(i)] & PTE_SHARE))
310             sys_page_map(curenvid, (void *)i,
311                         child, (void *)i, PTE_U | PTE_P | PTE_SYSCALL);
312     }
313
314     return 0;
315 }
```

练习九：

这个练习比较只需要在 `trap_dispatch()` 里添加题目中两个陷入对应的情况，分别调用已经写好的函数

就行了。添加部分代码如下：

```
284     // Handle keyboard and serial interrupts.
285     // LAB 5: Your code here.
286     if (tf->tf_trapno == IRQ_OFFSET + IRQ_KBD) {
287         kbd_intr();
288         return;
289     }
290
291     if (tf->tf_trapno == IRQ_OFFSET + IRQ_SERIAL) {
292         serial_intr();
293         return;
294     }
```

练习十：

最后一个练习是在 `user/sh.c` 中给 `<` 添加重定向，对shell不太了解，但是根据注释和已经写好的 `>` 的重定向，还是可以把代码补全的。根据注释提示：先把文件 `t` 打开为 `fd` ，然后判断，如果 `fd` 不为零则依次调用 `dup` 和 `close` 。唯一需要注意的部分应该是 `open` 的第二个参数，与 `>` 的重定向对比，并参考 `inc/lib.h` 中相关宏定义，可以猜出选择 `O_RDONLY` 。最终代码如下：

```
56              if ((fd = open(t, O_RDONLY)) < 0) {
57                  cprintf("open %s for read: %e", t, fd);
58                  exit();
59              }
60              if (fd != 0) {
61                  dup(fd, 0);
62                  close(fd);
63              }
64              break;
```