实验八：文件系统

1. 实验目的

通过完成本次实验，希望能达到以下目标

- 了解基本的文件系统系统调用的实现方法；
- 了解一个基于索引节点组织方式的Simple FS文件系统的设计与实现；
- 了解文件系统抽象层-VFS的设计与实现；

2. 实验内容

实验七完成了在内核中的同步互斥实验。本次实验涉及的是文件系统，通过分析了解 ucore 文件系统的总体架构设计，完善读写文件操作，从新实现基于文件系统的执行程序机制（即改写 do_execve），从而可以完成执行存储在磁盘上的文件和实现文件读写等功能。

2.1 练习

练习0：填写已有实验

本实验依赖实验 1/2/3/4/5/6/7。请把你做的实验 1/2/3/4/5/6/7 的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”/“LAB5”/“LAB6” /“LAB7”的注释相应部分。并确保编译通过。注意：为了能够正确执行 lab8 的测试应用程序，可能需对已完成的实验 1/2/3/4/5/6/7 的代码进行进一步改进。

练习1 完成读文件操作的实现（需要编码）

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，编写在 sfs_inode.c 中 sfs_io_nolock 读文件中数据的实现代码。

练习2 完成基于文件系统的执行程序机制的实现（需要编码）

改写 proc.c 中的 load_icode 函数和其他相关函数，实现基于文件系统的执行程序机制。执行：make qemu。如果能看看到 sh 用户程序的执行界面，则基本成功了。如果在 sh 用户界面上可以执行”ls”,”hello”等其他放置在 sfs 文件系统中的其他执行程序，则可以认为本实验基本成功。（**使用的是 qemu-1.0.1**）。

2.2 项目组成

```
.
├── boot
├── kern
│   ├── debug
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   └── ......
│   ├── fs
│   │   ├── devs
│   │   │   ├── dev.c
│   │   │   ├── dev_disk0.c
│   │   │   ├── dev.h
│   │   │   ├── dev_stdin.c
```

```
│   │   │   └── dev_stdout.c
│   │   ├── file.c
│   │   ├── file.h
│   │   ├── fs.c
│   │   ├── fs.h
│   │   ├── iobuf.c
│   │   ├── iobuf.h
│   │   ├── sfs
│   │   │   ├── bitmap.c
│   │   │   ├── bitmap.h
│   │   │   ├── sfs.c
│   │   │   ├── sfs_fs.c
│   │   │   ├── sfs.h
│   │   │   ├── sfs_inode.c
│   │   │   ├── sfs_io.c
│   │   │   └── sfs_lock.c
│   │   ├── swap
│   │   │   ├── swapfs.c
│   │   │   └── swapfs.h
│   │   ├── sysfile.c
│   │   ├── sysfile.h
│   │   └── vfs
│   │       ├── inode.c
│   │       ├── inode.h
│   │       ├── vfs.c
│   │       ├── vfsdev.c
│   │       ├── vfsfile.c
│   │       ├── vfs.h
│   │       ├── vfslookup.c
│   │       └── vfspath.c
│   ├── init
│   ├── libs
│   │   ├── stdio.c
│   │   ├── string.c
│   │   └── ……
│   ├── mm
│   │   ├── vmm.c
│   │   └── vmm.h
│   ├── process
│   │   ├── proc.c
│   │   ├── proc.h
│   │   └── ……
│   ├── schedule
│   ├── sync
│   ├── syscall
│   │   ├── syscall.c
│   │   └── ……
│   └── trap
│       ├── trap.c
│       └── ……
├── libs
├── tools
│   ├── mksfs.c
│   └── ……
└── user
    ├── badarg.c
    ├── badsegment.c
    ├── divzero.c
```

```
├── exit.c
├── faultread.c
├── faultreadkernel.c
├── forktest.c
├── forktree.c
├── hello.c
├── libs
│   ├── dir.c
│   ├── dir.h
│   ├── file.c
│   ├── file.h
│   ├── initcode.S
│   ├── lock.h
│   ├── stdio.c
│   ├── syscall.c
│   ├── syscall.h
│   ├── ulib.c
│   ├── ulib.h
│   └── umain.c
├── ls.c
├── sh.c
└── ……
```

本次实验主要是理解kern/fs目录中的部分文件，并可用user/*.c测试所实现的Simple FS文件系统是否能够正常工作。本次实验涉及到的代码包括：

- 文件系统测试用例： user/*.c：对文件系统的实现进行测试的测试用例；
- 通用文件系统接口
  - user/libs/file.[ch]|dir.[ch]|syscall.c：与文件系统操作相关的用户库实行；
  - kern/syscall.[ch]：文件中包含文件系统相关的内核态系统调用接口
  - kern/fs/sysfile.[ch]|file.[ch]：通用文件系统接口和实行
- 文件系统抽象层-VFS
  - kern/fs/vfs/*.[ch]：虚拟文件系统接口与实现
- Simple FS文件系统
  - kern/fs/sfs/*.[ch]：SimpleFS文件系统实现
- 文件系统的硬盘IO接口
  - kern/fs/devs/dev.[ch]|dev_disk0.c：disk0硬盘设备提供给文件系统的I/O访问接口和实现
- 辅助工具
  - tools/mksfs.c：创建一个Simple FS文件系统格式的硬盘镜像。（理解此文件的实现细节对理解SFS文件系统很有帮助）
- 对内核其它模块的扩充
  - kern/process/proc.[ch]：增加成员变量 struct fs_struct *fs_struct，用于支持进程对文件的访问；重写了do_execve load_icode等函数以支持执行文件系统中的文件。
  - kern/init/init.c：增加调用初始化文件系统的函数fs_init。

3. 文件系统设计与实现

3.1 ucore文件系统总体介绍

操作系统中负责管理和存储可长期保存数据的软件功能模块称为文件系统。在本次试验中，主要侧重文件系统的设计实现和对文件系统执行流程的分析与理解。

ucore的文件系统模型源于Havard的OS161的文件系统和Linux文件系统。但其实这二者都是源于传统的UNIX文件系统设计。UNIX提出了四个文件系统抽象概念：文件(file)、目录项(dentry)、索引节点(inode)和安装点(mount point)。

- 文件：UNIX文件中的内容可理解为是一有序字节buffer，文件都有一个方便应用程序识别的文件名称（也称文件路径名）。典型的文件操作有读、写、创建和删除等。
- 目录项：目录项不是目录，而是目录的组成部分。在UNIX中目录被看作一种特定的文件，而目录项是文件路径中的一部分。如一个文件路径名是“/test/testfile”，则包含的目录项为：根目录“/”，目录“test”和文件“testfile”，这三个都是目录项。一般而言，目录项包含目录项的名字（文件名或目录名）和目录项的索引节点（见下面的描述）位置。
- 索引节点：UNIX将文件的相关元数据信息（如访问控制权限、大小、拥有者、创建时间、数据内容等等信息）存储在一个单独的数据结构中，该结构被称为索引节点。
- 安装点：在UNIX中，文件系统被安装在一个特定的文件路径位置，这个位置就是安装点。所有的已安装文件系统都作为根文件系统树中的叶子出现在系统中。

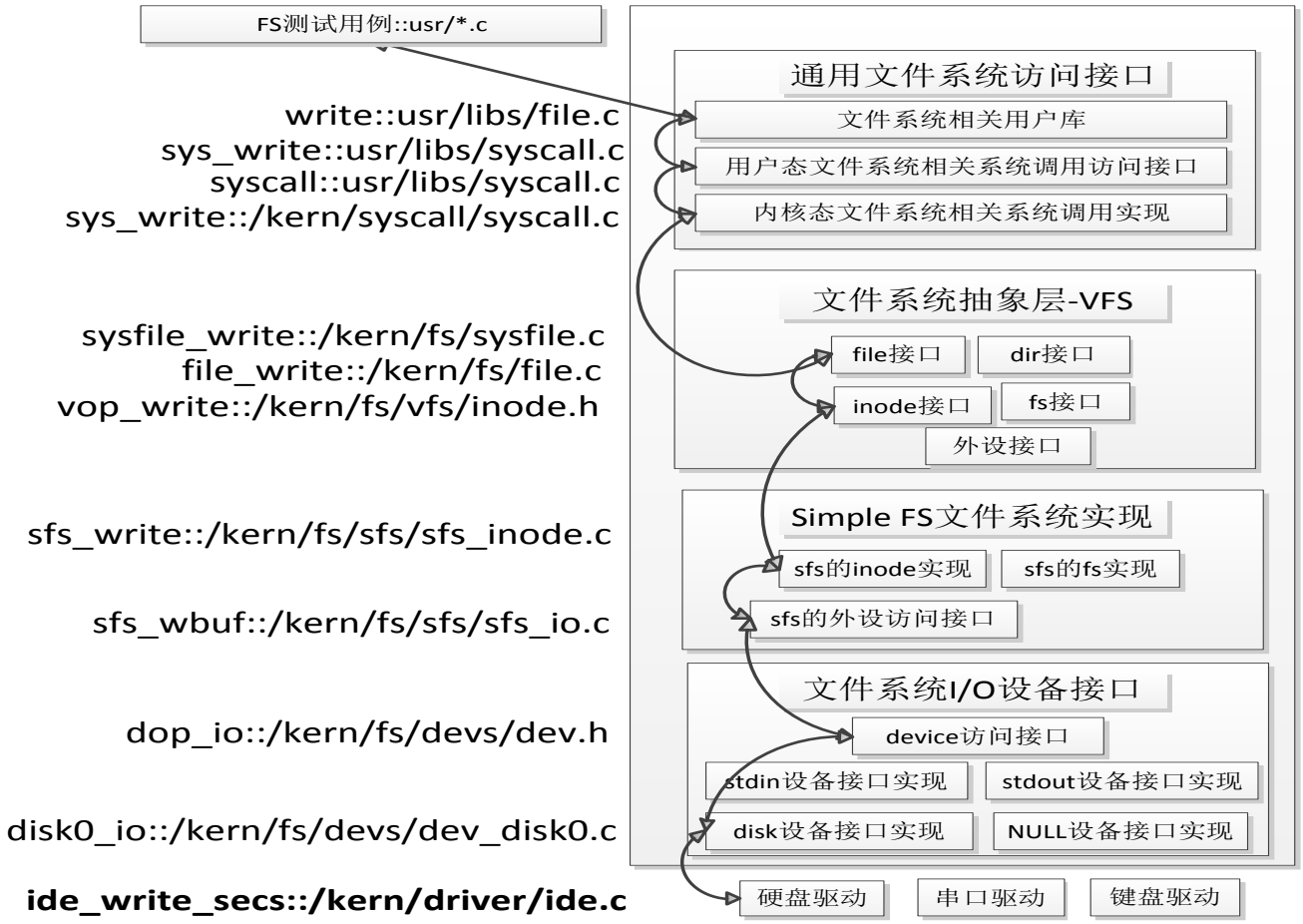
上述抽象概念形成了UNIX文件系统的逻辑数据结构，并需要通过一个具体文件系统的架构设计与实现把上述信息映射并储存到磁盘介质上。一个具体的文件系统需要在磁盘布局也实现上述抽象概念。比如文件元数据信息存储在磁盘块中的索引节点上。当文件被载如内存时，内核需要使用磁盘块中的索引点来构造内存中的索引节点。

ucore模仿了UNIX的文件系统设计，ucore的文件系统架构主要由四部分组成：

- 通用文件系统访问接口层：该层提供了一个从用户空间到文件系统的标准访问接口。这一层访问接口让应用程序能够通过一个简单的接口获得ucore内核的文件系统服务。
- 文件系统抽象层：向上提供一个一致的接口给内核其他部分（文件系统相关的系统调用实现模块和其他内核功能模块）访问。向下提供一个同样的抽象函数指针列表和数据结构屏蔽不同文件系统的实现细节。
- Simple FS文件系统层：一个基于索引方式的简单文件系统实例。向上通过各种具体函数实现以对应文件系统抽象层提出的抽象函数。向下访问外设接口
- 外设接口层：向上提供device访问接口屏蔽不同硬件细节。向下实现访问各种具体设备驱动的接口，比如disk设备接口/串口设备接口/键盘设备接口等。

对照上面的层次我们再大致介绍一下文件系统的访问处理过程，加深对文件系统的总体理解。假如应用程序操作文件（打开/创建/删除/读写），首先需要通过文件系统的通用文件系统访问接口层给用户空间提供的访问接口进入文件系统内部，接着由文件系统抽象层把访问请求转发给某一具体文件系统（比如SFS文件系统），具体文件系统（Simple FS文件系统层）把应用程序的访问请求转化为对磁盘上的block的处理请求，并通过外设接口层交给磁盘驱动例程来完成具体的磁盘操作。结合用户态写文件函数write的整个执行过程，我们可以比较清楚地看出ucore文件系统架构的层次和依赖关系。

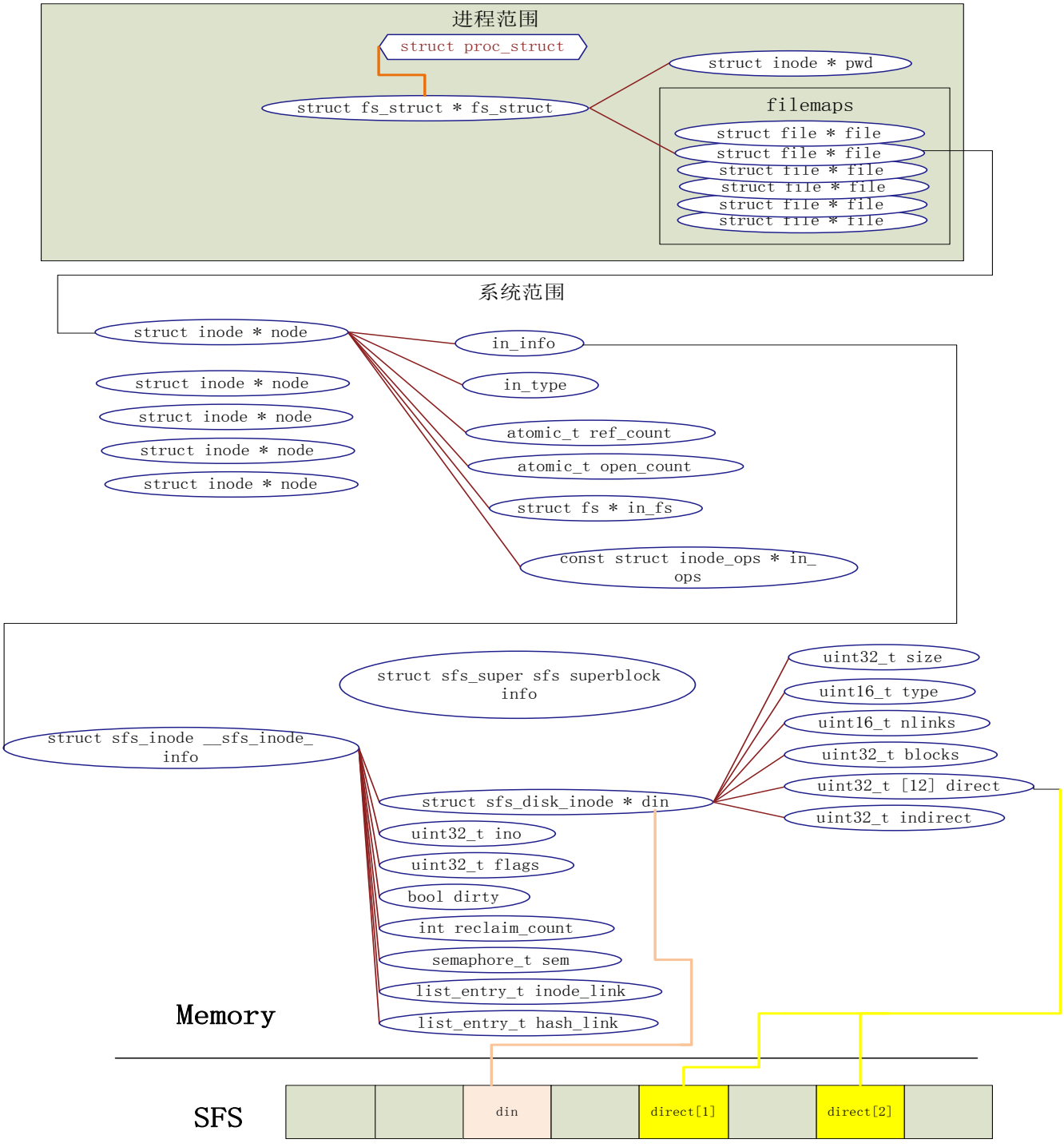
ucore文件系统总体结构



从ucore操作系统不同的角度来看，ucore中的文件系统架构包含四类主要的数据结构, 它们分别是：

- 超级块（SuperBlock），它主要从文件系统的全局角度描述特定文件系统的全局信息。它的作用范围是整个OS空间。
- 索引节点（inode）：它主要从文件系统的单个文件的角度它描述了文件的各种属性和数据所在位置。它的作用范围是整个OS空间。
- 目录项（dentry）：它主要从文件系统的文件路径的角度描述了文件路径中的特定目录。它的作用范围是整个OS空间。
- 文件（file），它主要从进程的角度描述了一个进程在访问文件时需要了解的文件标识，文件读写的位置，文件引用情况等信息。它的作用范围是某一具体进程。

如果一个用户进程打开了一个文件，那么在ucore中涉及的相关数据结构（其中相关数据结构将在下面各个小节中展开叙述）和关系如下图所示：



ucore中文件相关关键数据结构及其关系

3.2 通用文件系统访问接口

文件和目录相关用户库函数

Lab8 中部分用户库函数与文件系统有关，我们先讨论对单个文件进行操作的系统调用，然后讨论对目录和文件系统进行操作的系统调用。

在文件操作方面，最基本的相关函数是 open、close、read、write。在读写一个文件之前，首先要用 open 系统调用将其打开。open 的第一个参数指定文件的路径名，可使用绝对路径名；第二个参数指定打开的方式，可设置为 O_RDONLY、O_WRONLY、O_RDWR，分别表示只读、只写、可读

可写。在打开一个文件后，就可以使用它返回的文件描述符 fd 对文件进行相关操作。在使用完一个文件后，还要用 close 系统调用把它关闭，其参数就是文件描述符 fd。这样它的文件描述符就可以空出来，给别的文件使用。

读写文件内容的系统调用是 read 和 write。read 系统调用有三个参数：一个指定所操作的文件描述符，一个指定读取数据的存放地址，最后一个指定读多少个字节。在 C 程序中调用该系统调用的方法如下：

count = read(filehandle, buffer, nbytes);

该系统调用会把实际读到的字节数返回给 count 变量。在正常情形下这个值与 nbytes 相等，但有时可能会小一些。例如，在读文件时碰上了文件结束符，从而提前结束此次读操作。

如果由于参数无效或磁盘访问错误等原因，使得此次系统调用无法完成，则 count 被置为–1。而 write 函数的参数与之完全相同。

对于目录而言，最常用的操作是跳转到某个目录，这里对应的用户库函数是 chdir。然后就需要读目录的内容了，即列出目录中的文件或目录名，这在处理上与读文件类似，即需要通过 opendir 函数打开目录，通过 readdir 来获取目录中的文件信息，读完后还需通过 closedir 函数来关闭目录。由于在 ucore 中把目录看成是一个特殊的文件，所以 opendir 和 closedir 实际上就是调用与文件相关的 open 和 close 函数。只有 readdir 需要调用获取目录内容的特殊系统调用 sys_getdirentry。而且这里没有写目录这一操作。在目录中增加内容其实就是在此目录中创建文件，需要用到创建文件的函数。

文件和目录访问相关系统调用

与文件相关的 open、close、read、write 用户库函数对应的是 sys_open、sys_close、sys_read、sys_write 四个系统调用接口。与目录相关的 readdir 用户库函数对应的是 sys_getdirentry 系统调用。这些系统调用函数接口将通过 syscall 函数来获得 ucore 的内核服务。当到了 ucore 内核后，在调用文件系统抽象层的 file 接口和 dir 接口。

3.3 Simple FS文件系统

这里我们没有按照从上到下先讲文件系统抽象层，再讲具体的文件系统。这是由于如果能够理解 Simple FS（简称 SFS）文件系统，就可更好地分析文件系统抽象层的设计。即从具体走向抽象。ucore 内核把所有文件都看作是字节流，任何内部逻辑结构都是专用的，由应用程序负责解释。但是 ucore 区分文件的物理结构。ucore 目前支持如下几种类型的文件：

- 常规文件：文件中包括的内容信息是由应用程序输入。SFS 文件系统在普通文件上不强加任何内部结构，把其文件内容信息看作为字节。
- 目录：包含一系列的 entry，每个 entry 包含文件名和指向与之相关联的索引节点（index node）的指针。目录是按层次结构组织的。
- 链接文件：实际上一个链接文件是一个已经存在的文件的另一个可选择的文件名。
- 设备文件：不包含数据，但是提供了一个映射物理设备（如串口、键盘等）到一个文件名的机制。可通过设备文件访问外围设备。
- 管道：管道是进程间通讯的一个基础设施。管道缓存了其输入端所接受的数据，以便在管道输出端读的进程能一个先进先出的方式来接受数据。

在 lab8 中关注的主要是 SFS 支持的常规文件、目录和链接中的 hardlink 的设计实现。SFS 文件系统中目录和常规文件具有共同的属性，而这些属性保存在索引节点中。SFS 通过索引节点来管理目录和常规文件，索引节点包含操作系统所需要的关于某个文件的关键信息，比如文件的属性、访问许可权以及其它控制信息都保存在索引节点中。可以有多个文件名可指向一个索引节点。

3.3.1文件系统的布局

文件系统通常保存在磁盘上。在本实验中，第三个磁盘（即 disk0，前两个磁盘分别是 ucore.img 和 swap.img）用于存放一个 SFS 文件系统（Simple Filesystem）。通常文件系统中，磁盘的使用是以扇区（Sector）为单位的，但是为了实现简便，SFS 中以 block （4K，与内存 page 大小相等）为基

本单位。

SFS 文件系统的布局如下图所示。

第 0 个块（4K）是超级块（superblock），它包含了关于文件系统的所有关键参数，当计算机被启动或文件系统被首次接触时，超级块的内容就会被装入内存。其定义如下：

```
struct sfs_super {
    uint32_t magic;                                 /* magic number, should be SFS_MAGIC */
    uint32_t blocks;                                /* # of blocks in fs */
    uint32_t unused_blocks;                         /* # of unused blocks in fs */
    char info[SFS_MAX_INFO_LEN + 1];                /* infomation for sfs  */
};
```

可以看到，包含一个成员变量魔数 magic，其值为 0x2f8dbe2a，内核通过它来检查磁盘镜像是否是合法的 SFS img；成员变量 blocks 记录了 SFS 中所有 block 的数量，即 img 的大小；成员变量 unused_block 记录了 SFS 中还没有被使用的 block 的数量；成员变量 info 包含了字符串"simple file system"。

第 1 个块放了一个 root-dir 的 inode，用来记录根目录的相关信息。有关 inode 还将在后续部分介绍。这里只要理解 root-dir 是 SFS 文件系统的根结点，通过这个 root-dir 的 inode 信息就可以定位并查找到根目录下的所有文件信息。

从第 2 个块开始，根据 SFS 中所有块的数量，用 1 个 bit 来表示一个块的占用和未被占用的情况。这个区域称为 SFS 的 freemap 区域，这将占用若干个块空间。为了更好地记录和管理 freemap 区域，专门提供了两个文件 kern/fs/sfs/bitmap.[ch]来完成根据一个块号查找或设置对应的 bit 位的值。

最后在剩余的磁盘空间中，存放了所有其他目录和文件的 inode 信息和内容数据信息。需要注意的是虽然 inode 的大小小于一个块的大小（4096B），但为了实现简单，每个 inode 都占用一个完整的 block。

在 sfs_fs.c 文件中的 sfs_do_mount 函数中，完成了加载位于硬盘上的 SFS 文件系统的超级块 superblock 和 freemap 的工作。这样，在内存中就有了 SFS 文件系统的全局信息。

3.3.2索引节点

磁盘索引节点

SFS 中的磁盘索引节点代表了一个实际位于磁盘上的文件。首先我们看看在硬盘上的索引节点的内容：

```
struct sfs_disk_inode {
    uint32_t size;                  如果 inode 表示常规文件，则 size 是文件大小
    uint16_t type;                      inode 的文件类型
    uint16_t nlinks;                    此 inode 的硬链接数
    uint32_t blocks;                    此 inode 的数据块数的个数
    uint32_t direct[SFS_NDIRECT];  此 inode 的直接数据块索引值（有 SFS_NDIRECT 个）
    uint32_t indirect;                  此 inode 的一级间接数据块索引值
};
```

通过上表可以看出，如果 inode 表示的是文件，则成员变量 direct[]直接指向了保存文件内容数据的数据块索引值。indirect 间接指向了保存文件内容数据的数据块，indirect 指向的是间接数据块（indirect block），此数据块实际存放的全部是数据块索引，这些数据块索引指向的数据块才被用来存放文件内容数据。

默认的，ucore 里 SFS_NDIRECT 是 12，即直接索引的数据页大小为 12 * 4k = 48k；当使用一级间接数据块索引时，ucore 支持最大的文件大小为 12 * 4k + 1024 * 4k = 48k + 4m。数据索引表内，0 表示一个无效的索引，inode 里 blocks 表示该文件或者目录占用的磁盘的 block 的个数。indiret 为 0 时，表示不使用一级索引块。（因为 block 0 用来保存 super block，它不可能被其他任何文件或目录使用，所以这么设计也是合理的）。

对于普通文件，索引值指向的 block 中保存的是文件中的数据。而对于目录，索引值指向的数据保存的是目录下所有的文件名以及对应的索引节点所在的索引块（磁盘块）所形成的数组。数据

结构如下：

```
/* file entry (on disk) */
struct sfs_disk_entry {
    uint32_t ino;                                   索引节点所占数据块索引值
    char name[SFS_MAX_FNAME_LEN + 1];               文件名
};
```

操作系统中，每个文件系统下的 inode 都应该分配唯一的 inode 编号。SFS 下，为了实现的简便（偷懒），每个 inode 直接用他所在的磁盘 block 的编号作为 inode 编号。比如，root block 的 inode 编号为 1；每个 sfs_disk_entry 数据结构中，name 表示目录下文件或文件夹的名称，ino 表示磁盘 block 编号，通过读取该 block 的数据，能够得到相应的文件或文件夹的 inode。ino 为 0 时，表示一个无效的 entry。

此外，和 inode 相似，每个 sfs_dirent_entry 也占用一个 block。

内存中的索引节点

```
/* inode for sfs */
struct sfs_inode {
    struct sfs_disk_inode *din;             /* on-disk inode */
    uint32_t ino;                           /* inode number */
    uint32_t flags;                         /* inode flags */
    bool dirty;                             /* true if inode modified */
    int reclaim_count;                      /* kill inode if it hits zero */
    semaphore_t sem;                        /* semaphore for din */
    list_entry_t inode_link;                /* entry for linked-list in sfs_fs */
    list_entry_t hash_link;                 /* entry for hash linked-list in sfs_fs */
};
```

可以看到 SFS 中的内存 inode 包含了 SFS 的硬盘 inode 信息，而且还增加了其他一些信息，这属于是便于进行是判断否改写、互斥操作、回收和快速地定位等作用。需要注意，一个内存 inode 是在打开一个文件后才创建的，如果关机则相关信息都会消失。而硬盘 inode 的内容是保存在硬盘中的，只是在进程需要时才被读入到内存中，用于访问文件或目录的具体内容数据

为了方便实现上面提到的多级数据的访问以及目录中 entry 的操作，对 inode SFS 实现了一些辅助的函数：

1. sfs_bmap_load_nolock：将对应 sfs_inode 的第 index 个索引指向的 block 的索引值取出存到相应的指针指向的单元（ino_store）。该函数只接受 index <= inode->blocks 的参数。当 index == inode->blocks 时，该函数理解为需要为 inode 增长一个 block。并标记 inode 为 dirty（所有对 inode 数据的修改都要做这样的操作，这样，当 inode 不再使用的时候，sfs 能够保证 inode 数据能够被写回到磁盘）。sfs_bmap_load_nolock 调用的 sfs_bmap_get_nolock 来完成相应的操作，阅读 sfs_bmap_get_nolock，了解他是如何工作的。（sfs_bmap_get_nolock 只由 sfs_bmap_load_nolock 调用）
2. sfs_bmap_truncate_nolock：将多级数据索引表的最后一个 entry 释放掉。他可以认为是 sfs_bmap_load_nolock 中，index == inode->blocks 的逆操作。当一个文件或目录被删除时，sfs 会循环调用该函数直到 inode->blocks 减为 0，释放所有的数据页。函数通过 sfs_bmap_free_nolock 来实现，他应该是 sfs_bmap_get_nolock 的逆操作。和 sfs_bmap_get_nolock 一样，调用 sfs_bmap_free_nolock 也要格外小心。
3. sfs_dirent_read_nolock：将目录的第 slot 个 entry 读取到指定的内存空间。他通过上面提到的函数来完成。
4. sfs_dirent_write_nolock：用指定的 entry 来替换某个目录下的第 slot 个 entry。他通过调用 sfs_bmap_load_nolock 保证，当第 slot 个 entry 不存在时（slot == inode->blocks），SFS 会分配一个新的 entry，即在目录尾添加了一个 entry。
5. sfs_dirent_search_nolock：是常用的查找函数。他在目录下查找 name，并且返回相应的搜索结果（文件或文件夹）的 inode 的编号（也是磁盘编号），和相应的 entry 在该目录

的 index 编号以及目录下的数据页是否有空闲的 entry。(SFS 实现里文件的数据页是连续的，不存在任何空洞；而对于目录，数据页不是连续的，当某个 entry 删除的时候，SFS 通过设置 entry->ino 为 0 将该 entry 所在的 block 标记为 free，在需要添加新 entry 的时候，SFS 优先使用这些 free 的 entry，其次才会去在数据页尾追加新的 entry。

注意，这些后缀为 nolock 的函数，只能在已经获得相应 inode 的 semaphore 才能调用。

Inode 的文件操作函数

```
static const struct inode_ops sfs_node_fileops = {
    .vop_magic                  = VOP_MAGIC,
    .vop_open                   = sfs_openfile,
    .vop_close                  = sfs_close,
    .vop_read                   = sfs_read,
    .vop_write                  = sfs_write,
    ……
};
```

上述 sfs_openfile、sfs_close、sfs_read 和 sfs_write 分别对应用户进程发出的 open、close、read、write 操作。其中 sfs_openfile 不用做什么事；sfs_close 需要把对文件的修改内容写回到硬盘上，这样确保硬盘上的文件内容数据是最新的；sfs_read 和 sfs_write 函数都调用了一个函数 sfs_io，并最终通过访问硬盘驱动来完成对文件内容数据的读写。

Inode 的目录操作函数

```
static const struct inode_ops sfs_node_dirops = {
    .vop_magic                  = VOP_MAGIC,
    .vop_open                   = sfs_opendir,
    .vop_close                  = sfs_close,
    .vop_getdirentry            = sfs_getdirentry,
    .vop_lookup                 = sfs_lookup,
    ……
};
```

对于目录操作而言，由于目录也是一种文件，所以 sfs_opendir、sys_close 对应户进程发出的 open、close 函数。相对于 sfs_open，sfs_opendir 只是完成一些 open 函数传递的参数判断，没做其他更多的事情。目录的 close 操作与文件的 close 操作完全一致。由于目录的内容数据与文件的内容数据不同，所以读出目录的内容数据的函数是 sfs_getdirentry，其主要工作是获取目录下的文件 inode 信息。

3.4 文件系统抽象层-VFS

文件系统抽象层是把不同文件系统的对外共性接口提取出来，形成一个函数指针数组，这样，通用文件系统访问接口层只需访问文件系统抽象层，而不需关心具体文件系统的实现细节和接口。

3.4.1 file&dir接口

file&dir接口层定义了进程在内核中直接访问的文件相关信息，这定义在file数据结构中，具体描述如下：

```
struct file {
    enum {
        FD_NONE, FD_INIT, FD_OPENED, FD_CLOSED,
    } status;                   //访问文件的执行状态
    bool readable;                  //文件是否可读
    bool writable;                  //文件是否可写
    int fd;                     //文件在filemap中的索引值
    off_t pos;                  //访问文件的当前位置
    struct inode *node;             //该文件对应的内存inode指针
    atomic_t open_count;            //打开此文件的次数
};
```

而在kern/process/proc.h中的proc_struct结构中描述了进程访问文件的数据接口 fs_struct，其数据结构定义如下：

```
struct fs_struct {
   struct inode *pwd;            //进程当前执行目录的内存inode指针
   struct file *filemap;         //进程打开文件的数组
   atomic_t fs_count;            //访问此文件的线程个数？？
   semaphore_t fs_sem;           //确保对进程控制块中fs_struct的互斥访问
};
```

当创建一个进程后，该进程的fs_struct将会被初始化或复制父进程的fs_struct。当用户进程打开一个文件时，将从filemap数组中取得一个空闲file项，然后会把此file的成员变量node指针指向一个代表此文件的inode的起始地址。

3.4.2 inode接口

index node是位于内存的索引节点，它是VFS结构中的重要数据结构，因为它实际负责把不同文件系统的特定索引节点信息（甚至不能算是一个索引节点）统一封装起来，避免了进程直接访问具体文件系统。其定义如下：

```
struct inode {
   union {                       //包含不同文件系统特定inode信息的union域
      struct device __device_info;       //设备文件系统内存inode信息
      struct sfs_inode __sfs_inode_info;  //SFS文件系统内存inode信息
   } in_info;
   enum {
      inode_type_device_info = 0x1234,
      inode_type_sfs_inode_info,
   } in_type;                    //此inode所属文件系统类型
   atomic_t ref_count;           //此inode的引用计数
   atomic_t open_count;          //打开此inode对应文件的个数
   struct fs *in_fs;             //抽象的文件系统，包含访问文件系统的函数指针
   const struct inode_ops *in_ops;   //抽象的inode操作，包含访问inode的函数指针
};
```

在inode中，有一成员变量为in_ops，这是对此inode的操作函数指针列表，其数据结构定义如下：

```
struct inode_ops {
   unsigned long vop_magic;
   int (*vop_open)(struct inode *node, uint32_t open_flags);
   int (*vop_close)(struct inode *node);
   int (*vop_read)(struct inode *node, struct iobuf *iob);
   int (*vop_write)(struct inode *node, struct iobuf *iob);
   int (*vop_getdirentry)(struct inode *node, struct iobuf *iob);
   int (*vop_create)(struct inode *node, const char *name, bool excl, struct inode **node_store);
   int (*vop_lookup)(struct inode *node, char *path, struct inode **node_store);
      ……
};
```

参照上面对SFS中的索引节点操作函数的说明，可以看出inode_ops是对常规文件、目录、设备文件所有操作的一个抽象函数表示。对于某一具体的文件系统中的文件或目录，只需实现相关的函数，就可以被用户进程访问具体的文件了，且用户进程无需了解具体文件系统的实现细节。

3.5 设备层文件IO层

在本实验中，为了统一地访问设备，我们可以把一个设备看成一个文件，通过访问文件的接口来访问设备。目前实现了 stdin 设备文件文件、stdout 设备文件、disk0 设备。stdin 设备就是键盘，stdout 设备就是 CONSOLE（串口、并口和文本显示器），而 disk0 设备是承载 SFS 文件系统的磁盘设备。下面我们逐一分析 ucore 是如何让用户把设备看成文件来访问。

3.5.1 关键数据结构

为了表示一个设备，需要有对应的数据结构，ucore 为此定义了 struct device，其描述如下：

```
struct device {
   size_t d_blocks;   //设备占用的数据块个数
   size_t d_blocksize; //数据块的大小
   int (*d_open)(struct device *dev, uint32_t open_flags); //打开设备的函数指针
   int (*d_close)(struct device *dev); //关闭设备的函数指针
   int (*d_io)(struct device *dev, struct iobuf *iob, bool write); //读写设备的函数指针
   int (*d_ioctl)(struct device *dev, int op, void *data); //用 ioctl 方式控制设备的函数指针
};
```

这个数据结构能够支持对块设备（比如磁盘）、字符设备（比如键盘、串口）的表示，完成对设备的基本操作。ucore 虚拟文件系统为了把这些设备链接在一起，还定义了一个设备链表，即双向链表 vdev_list，这样通过访问此链表，可以找到 ucore 能够访问的所有设备文件。

但这个设备描述没有与文件系统以及表示一个文件的 inode 数据结构建立关系，为此，还需要另外一个数据结构把 device 和 inode 联通起来，这就是 vfs_dev_t 数据结构：

```
// device info entry in vdev_list
typedef struct {
    const char *devname;
    struct inode *devnode;
    struct fs *fs;
    bool mountable;
    list_entry_t vdev_link;
} vfs_dev_t;
```

利用 vfs_dev_t 数据结构，就可以让文件系统通过一个链接 vfs_dev_t 结构的双向链表找到 device 对应的 inode 数据结构，一个 inode 节点的成员变量 in_type 的值是 0x1234，则此 inode 的成员变量 in_info 将成为一个 device 结构。这样 inode 就和一个设备建立了联系，这个 inode 就是一个设备文件。

3.5.2stdout设备文件

初始化

既然 stdout 设备是设备文件系统的文件，自然有自己的 inode 结构。在系统初始化时，即只需如下处理过程

kern_init-->fs_init-->dev_init-->dev_init_stdout --> dev_create_inode
--> stdout_device_init
--> vfs_add_dev

在 dev_init_stdout 中完成了对 stdout 设备文件的初始化。即首先创建了一个 inode，然后通过 stdout_device_init 完成对 inode 中的成员变量 inode->__device_info 进行初始：

这里的 stdout 设备文件实际上就是指的 console 外设（它其实是串口、并口和 CGA 的组合型外设）。这个设备文件是一个只写设备，如果读这个设备，就会出错。接下来我们看看 stdout 设备的相关处理过程。

初始化

stdout 设备文件的初始化过程主要由 stdout_device_init 完成，其具体实现如下：

```
static void
stdout_device_init(struct device *dev) {
    dev->d_blocks = 0;
    dev->d_blocksize = 1;
    dev->d_open = stdout_open;
    dev->d_close = stdout_close;
```

```
    dev->d_io = stdout_io;
    dev->d_ioctl = stdout_ioctl;
}
```

可以看到，stdout_open 函数完成设备文件打开工作，如果发现用户进程调用 open 函数的参数 flags 不是只写（O_WRONLY），则会报错。

访问操作实现

stdout_io 函数完成设备的写操作工作，具体实现如下：

```
static int
stdout_io(struct device *dev, struct iobuf *iob, bool write) {
    if (write) {
        char *data = iob->io_base;
        for (; iob->io_resid != 0; iob->io_resid --) {
            cputchar(*data ++);
        }
        return 0;
    }
    return -E_INVAL;
}
```

可以看到，要写的数据放在 iob->io_base 所指的内存区域，一直写到 iob->io_resid 的值为 0 为止。每次写操作都是通过 cputchar 来完成的，此函数最终将通过 console 外设驱动来完成把数据输出到串口、并口和 CGA 显示器上过程。另外，也可以注意到，如果用户想执行读操作，则 stdout_io 函数直接返回错误值-E_INVAL。

3.5.3 stdin设备文件

这里的 stdin 设备文件实际上就是指的键盘。这个设备文件是一个只读设备，如果写这个设备，就会出错。接下来我们看看 stdin 设备的相关处理过程。

初始化

stdin 设备文件的初始化过程主要由 stdin_device_init 完成了主要的初始化工作，具体实现如下：

```
static void
stdin_device_init(struct device *dev) {
    dev->d_blocks = 0;
    dev->d_blocksize = 1;
    dev->d_open = stdin_open;
    dev->d_close = stdin_close;
    dev->d_io = stdin_io;
    dev->d_ioctl = stdin_ioctl;

    p_rpos = p_wpos = 0;
    wait_queue_init(wait_queue);
}
```

相对于 stdout 的初始化过程，stdin 的初始化相对复杂一些，多了一个 stdin_buffer 缓冲区，描述缓冲区读写位置的变量 p_rpos、p_wpos 以及用于等待缓冲区的等待队列 wait_queue。在 stdin_device_init 函数的初始化中，也完成了对 p_rpos、p_wpos 和 wait_queue 的初始化。

访问操作实现

stdin_io 函数负责完成设备的读操作工作，具体实现如下：

```
static int
stdin_io(struct device *dev, struct iobuf *iob, bool write) {
    if (!write) {
        int ret;
        if ((ret = dev_stdin_read(iob->io_base, iob->io_resid)) > 0) {
            iob->io_resid -= ret;
        }
        return ret;
    }
    return -E_INVAL;
```

```
}
```

可以看到，如果是写操作，则 stdin_io 函数直接报错返回。所以这也进一步说明了此设备文件是只读文件。如果此读操作，则此函数进一步调用 dev_stdin_read 函数完成对键盘设备的读入操作。dev_stdin_read 函数的实现相对复杂一些，主要的流程如下：

```
static int
dev_stdin_read(char *buf, size_t len) {
    int ret = 0;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        for (; ret < len; ret ++, p_rpos ++) {
        try_again:
            if (p_rpos < p_wpos) {
                *buf ++ = stdin_buffer[p_rpos % stdin_BUFSIZE];
            }
            else {
                wait_t __wait, *wait = &__wait;
                wait_current_set(wait_queue, wait, WT_KBD);
                local_intr_restore(intr_flag);

                schedule();

                local_intr_save(intr_flag);
                wait_current_del(wait_queue, wait);
                if (wait->wakeup_flags == WT_KBD) {
                    goto try_again;
                }
                break;
            }
        }
    }
    local_intr_restore(intr_flag);
    return ret;
}
```

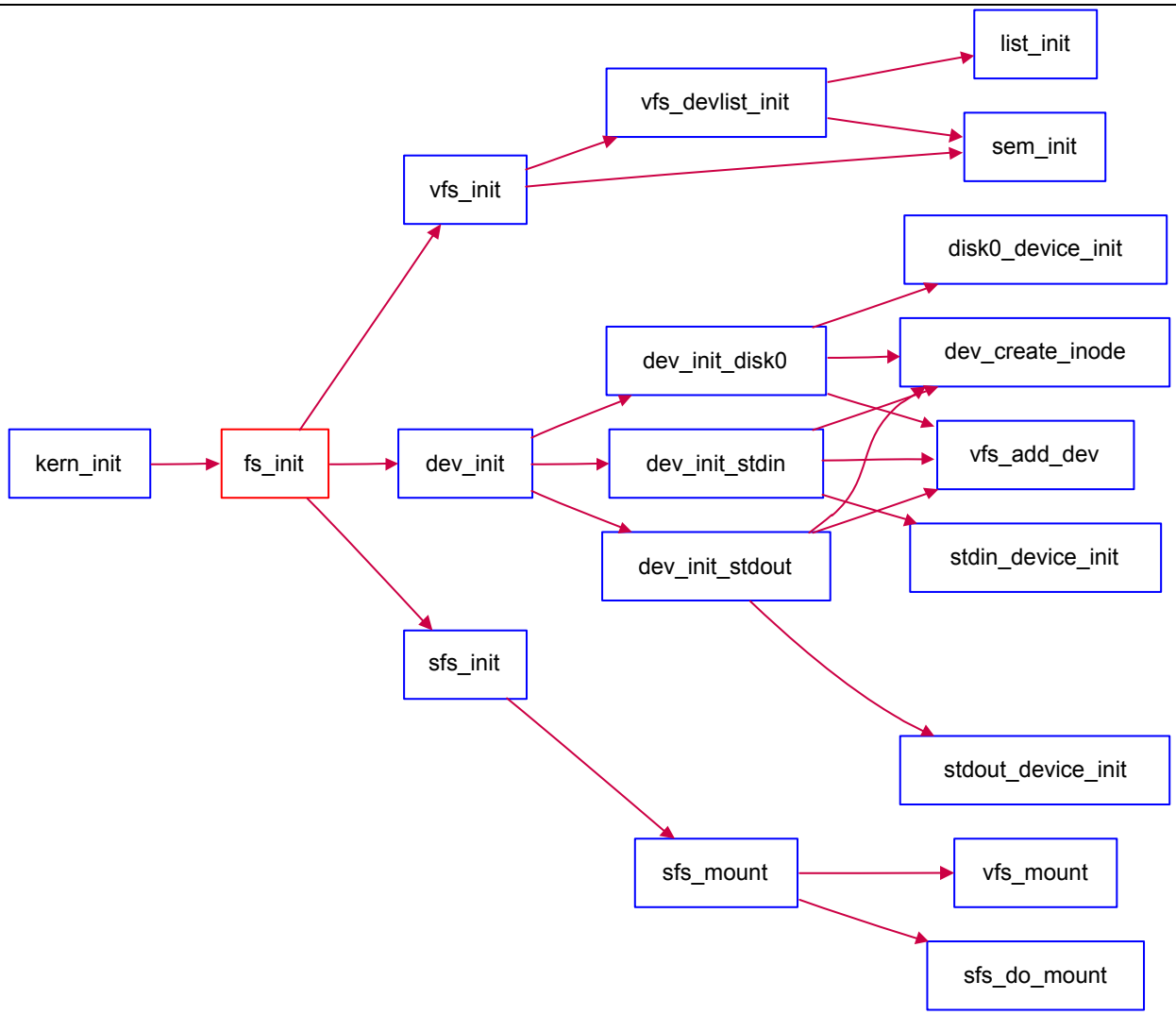
在上述函数中可以看出，如果 p_rpos < p_wpos，则表示有键盘输入的新字符在 stdin_buffer 中，于是就从 stdin_buffer 中取出新字符放到 iobuf 指向的缓冲区中；如果 p_rpos >=p_wpos，则表明没有新字符，这样调用 read 用户态库函数的用户进程就需要采用等待队列的睡眠操作进入睡眠状态，等待键盘输入字符的产生。

键盘输入字符后，如何唤醒等待键盘输入的用户进程呢？回顾 lab1 中的外设中断处理，可以了解到，当用户敲击键盘时，会产生键盘中断，在 trap_dispatch 函数中，当识别出中断是键盘中断（中断号为 IRQ_OFFSET + IRQ_KBD）时，会调用 dev_stdin_write 函数，来把字符写入到 stdin_buffer 中，且会通过等待队列的唤醒操作唤醒正在等待键盘输入的用户进程。

3.6 实验执行流程概述

与实验七相比，实验八增加了文件系统，并因此实现了通过文件系统来加载可执行文件到内存中运行的功能，导致对进程管理相关的实现比较大的调整。我们来简单看看文件系统是如何初始化并能在 ucore 的管理下正常工作的。

首先看看 kern_init 函数，可以发现与 lab7 相比增加了对 fs_init 函数的调用。fs_init 函数就是文件系统初始化的总控函数，它进一步调用了虚拟文件系统初始化函数 vfs_init，与文件相关的设备初始化函数 dev_init 和 Simple FS 文件系统的初始化函数 sfs_init。这三个初始化函数联合在一起，协同完成了整个虚拟文件系统、SFS 文件系统和文件系统对应的设备（键盘、串口、磁盘）的初始化工作。其函数调用关系图如下所示：



文件系统初始化调用关系图

参考上图，并结合源码分析，可大致了解到文件系统的整个初始化流程。vfs_init 主要建立了一个 device list 双向链表 vdev_list，为后续具体设备（键盘、串口、磁盘）以文件的形式呈现建立查找访问通道。dev_init 函数通过进一步调用 disk0/stdin/stdout_device_init 完成对具体设备的初始化，把它们抽象成一个设备文件，并建立对应的 inode 数据结构，最后把它们链入到 vdev_list 中。这样通过虚拟文件系统就可以方便地以文件的形式访问这些设备了。sfs_init 是完成对 Simple FS 的初始化工作，并把此实例文件系统挂在虚拟文件系统中，从而让 ucore 的其他部分能够通过访问虚拟文件系统的接口来进一步访问到 SFS 实例文件系统。

3.7文件操作实现

3.7.1 打开文件

有了上述分析后，我们可以看看如果一个用户进程打开文件会做哪些事情？首先假定用户进程需要打开的文件已经存在在硬盘上。以 user/sfs_filetest1.c 为例，首先用户进程会调用在 main 函数中的如下语句：

```
int fd1 = safe_open("/test/testfile", O_RDWR | O_TRUNC);
```

从字面上可以看出，如果 ucore 能够正常查找到这个文件，就会返回一个代表文件的文件描述符 fd1，这样在接下来的读写文件过程中，就直接用这样 fd1 来代表就可以了。那这个打开文件的过程是如何一步一步实现的呢？

通用文件访问接口层的处理流程

首先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数： open->sys_open->syscall， 从而引起系统调用进入到内核态。到了内核态后，通过中断处理例程，会调用到

sys_open 内核函数，并进一步调用 sysfile_open 内核函数。到了这里，需要把位于用户空间的字符串"/test/testfile"拷贝到内核空间中的字符串 path 中，并进入到文件系统抽象层的处理流程完成进一步的打开文件操作中。

文件系统抽象层的处理流程

1. 分配一个空闲的 file 数据结构变量 file

在文件系统抽象层的处理中，首先调用的是 file_open 函数，它要给这个即将打开的文件分配一个 file 数据结构的变量，这个变量其实是当前进程的打开文件数组 current->fs_struct->filemap[]中的一个空闲元素（即还没用于一个打开的文件），而这个元素的索引值就是最终要返回到用户进程并赋值给变量 fd1。到了这一步还仅仅是给当前用户进程分配了一个 file 数据结构的变量，还没有找到对应的文件索引节点。

为此需要进一步调用 vfs_open 函数来找到 path 指出的文件所对应的基于 inode 数据结构的 VFS 索引节点 node。vfs_open 函数需要完成两件事情：通过 vfs_lookup 找到 path 对应文件的 inode；调用 vop_open 函数打开文件。

2. 找到文件设备的根目录“/”的索引节点

需要注意，这里的 vfs_lookup 函数是一个针对目录的操作函数，它会调用 vop_lookup 函数来找到 SFS 文件系统中的“/test”目录下的“testfile”文件。为此，vfs_lookup 函数首先调用 get_device 函数，并进一步调用 vfs_get_bootfs 函数（其实调用了）来找到根目录“/”对应的 inode。这个 inode 就是位于 vfs.c 中的 inode 变量 bootfs_node。这个变量在 init_main 函数（位于 kern/process/proc.c）执行时获得了赋值。

3. 找到根目录“/”下的“test”子目录对应的索引节点

在找到根目录对应的 inode 后，通过调用 vop_lookup 函数来查找“/”和“test”这两层目录下的文件“testfile”所对应的索引节点，如果找到就返回此索引节点。

4. 把 file 和 node 建立联系

完成第 3 步后，将返回到 file_open 函数中，通过执行语句“file->node=node;”，就把当前进程的 current->fs_struct->filemap[fd]（即 file 所指变量）的成员变量 node 指针指向了代表“/test/testfile”文件的索引节点 node。这时返回 fd。经过重重回退，通过系统调用返回，用户态的 syscall->sys_open->open->safe_open 等用户函数的层层函数返回，最终把把 fd 赋值给 fd1。自此完成了打开文件操作。但这里我们还没有分析第 2 和第 3 步是如何进一步调用 SFS 文件系统提供的函数找位于 SFS 文件系统上的“/test/testfile”所对应的 sfs 磁盘 inode 的过程。下面需要进一步对此进行分析。

SFS 文件系统层的处理流程

这里需要分析文件系统抽象层中没有彻底分析的 vop_lookup 函数到底做了啥。下面我们来看看。在 sfs_inode.c 中的 sfs_node_dirops 变量定义了“.vop_lookup = sfs_lookup”，所以我们重点分析 sfs_lookup 的实现。

sfs_lookup 有三个参数：node，path，node_store。其中 node 是根目录“/”所对应的 inode 节点；path 是文件“testfile”的绝对路径“/test/testfile”，而 node_store 是经过查找获得的“testfile”所对应的 inode 节点。

Sfs_lookup 函数以“/”为分割符，从左至右逐一分解 path 获得各个子目录和最终文件对应的 inode 节点。在本例中是分解出“test”子目录，并调用 sfs_lookup_once 函数获得“test”子目录对应的 inode 节点 subnode，然后循环进一步调用 sfs_lookup_once 查找以“test”子目录下的文件“testfile1”所对应的 inode 节点。当无法分解 path 后，就意味着找到了 testfile1 对应的 inode 节点，就可顺利返回了。

当然这里讲得还比较简单，sfs_lookup_once 将调用 sfs_dirent_search_nolock 函数来查找与路径名匹配的目录项，如果找到目录项，则根据目录项中记录的 inode 所处的数据块索引值找到路径名

对应的 SFS 磁盘 inode，并读入 SFS 磁盘 inode 对的内容，创建 SFS 内存 inode。

3.7.2 读文件

读文件其实就是读出目录中的目录项，首先假定文件在磁盘上且已经打开。用户进程有如下语句：

```
read(fd, data, len);
```

即读取 fd 对应文件，读取长度为 len，存入 data 中。下面来分析一下读文件的实现。

通用文件访问接口层的处理流程

先进入通用文件访问接口层的处理流程，即进一步调用如下用户态函数：read->sys_read->syscall，从而引起系统调用进入到内核态。到了内核态以后，通过中断处理例程，会调用到 sys_read 内核函数，并进一步调用 sysfile_read 内核函数，进入到文件系统抽象层处理流程完成进一步读文件的操作。

文件系统抽象层的处理流程

1) 检查错误，即检查读取长度是否为 0 和文件是否可读。
2) 分配 buffer 空间，即调用 kmalloc 函数分配 4096 字节的 buffer 空间。
3) 读文件过程
   [1] 实际读文件
   循环读取文件，每次读取 buffer 大小。每次循环中，先检查剩余部分大小，若其小于 4096 字节，则只读取剩余部分的大小。然后调用 file_read 函数（详细分析见后）将文件内容读取到 buffer 中，alen 为实际大小。调用 copy_to_user 函数将读到的内容拷贝到用户的内存空间中，调整各变量以进行下一次循环读取，直至指定长度读取完成。最后函数调用层层返回至用户程序，用户程序收到了读到的文件内容。
   [2] file_read 函数
   这个函数是读文件的核心函数。函数有 4 个参数，fd 是文件描述符，base 是缓存的基地址，len 是要读取的长度，copied_store 存放实际读取的长度。函数首先调用 fd2file 函数找到对应的 file 结构，并检查是否可读。调用 filemap_acquire 函数使打开这个文件的计数加 1。调用 vop_read 函数将文件内容读到 iob 中（详细分析见后）。调整文件指针偏移量 pos 的值，使其向后移动实际读到的字节数 iobuf_used(iob)。最后调用 filemap_release 函数使打开这个文件的计数减 1，若打开计数为 0，则释放 file。

SFS 文件系统层的处理流程

vop_read 函数实际上是对 sfs_read 的包装。在 sfs_inode.c 中 sfs_node_fileops 变量定义了.vop_read = sfs_read，所以下面来分析 sfs_read 函数的实现。

sfs_read 函数调用 sfs_io 函数。它有三个参数，node 是对应文件的 inode，iob 是缓存，write 表示是读还是写的布尔值（0 表示读，1 表示写），这里是 0。函数先找到 inode 对应 sfs 和 sin，然后调用 sfs_io_nolock 函数进行读取文件操作，最后调用 iobuf_skip 函数调整 iobuf 的指针。

在 sfs_io_nolock 函数中，先计算一些辅助变量，并处理一些特殊情况（比如越界），然后有 sfs_buf_op = sfs_rbuf,sfs_block_op = sfs_rblock，设置读取的函数操作。接着进行实际操作，先处理起始的没有对齐到块的部分，再以块为单位循环处理中间的部分，最后处理末尾剩余的部分。每部分中都调用 sfs_bmap_load_nolock 函数得到 blkno 对应的 inode 编号，并调用 sfs_rbuf 或 sfs_rblock 函数读取数据（中间部分调用 sfs_rblock，起始和末尾部分调用 sfs_rbuf），调整相关变量。完成后如果 offset + alen > din->fileinfo.size（写文件时会出现这种情况，读文件时不会出现这种情况，alen 为实际读写的长度），则调整文件大小为 offset + alen 并设置 dirty 变量。

sfs_bmap_load_nolock 函数将对应 sfs_inode 的第 index 个索引指向的 block 的索引值取出存到相应的指针指向的单元（ino_store）。它调用 sfs_bmap_get_nolock 来完成相应的操作。sfs_rbuf 和 sfs_rblock 函数最终都调用 sfs_rwblock_nolock 函数完成操作，而 sfs_rwblock_nolock 函数调用 dop_io->disk0_io->disk0_read_blks_nolock->ide_read_secs 完成对磁盘的操作。

4. 实验报告要求

从网站上下载 lab8.zip 后，解压得到本文档和代码目录 lab8，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab8-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有“LAB8”的注释，这是需要主要修改的内容。代码中所有需要完成的地方 challenge 除外）都有“LAB8”和“YOUR CODE”的注释，请在提交时特别注意保持注释，并将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。