# 9.Lab8实验八：文件系统

## 9.1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；

## 9.2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。

### 9.2.1练习

对实验报告的要求：

* 基于markdown格式来完成，以文本方式为主
* 填写各个基本练习中要求完成的报告内容
* 完成实验后，请分析ucore\_lab中提供的参考答案，并请在实验报告中说明你的实现与参考答案的区别
* 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
* 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

#### 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。

请在实验报告中给出设计实现”UNIX的PIPE机制“的概要设方案，鼓励给出详细设计方案

#### 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。

请在实验报告中给出设计实现基于”UNIX的硬链接和软链接机制“的概要设方案，鼓励给出详细设计方案

**祝贺你通过自己的努力，完成了ucore OS lab1-lab8!**

### 9.2.2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  
  n user/libs/file.[ch]|dir.[ch]|syscall.c：与文件系统操作相关的用户库实行；  
  n kern/syscall.[ch]：文件中包含文件系统相关的内核态系统调用接口  
  n 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  
  n kern/fs/vfs/\*.[ch]：虚拟文件系统接口与实现
* Simple FS文件系统  
  n kern/fs/sfs/\*.[ch]：SimpleFS文件系统实现
* 文件系统的硬盘IO接口  
  n 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  
  n 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  
  n 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。  
  n kern/init/init.c：增加调用初始化文件系统的函数fs\_init。

## 9.3文件系统设计与实现

### 9.3.1 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)。

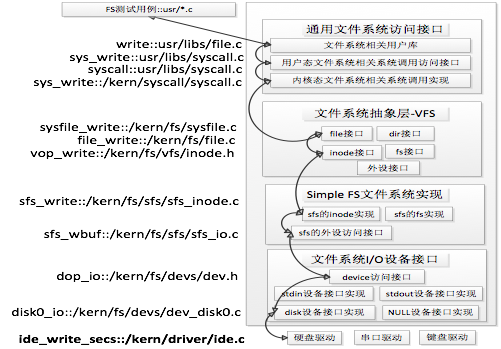
* 文件：UNIX文件中的内容可理解为是一有序字节buffer，文件都有一个方便应用程序识别的文件名称（也称文件路径名）。典型的文件操作有读、写、创建和删除等。
* 目录项：目录项不是目录，而是目录的组成部分。在UNIX中目录被看作一种特定的文件，而目录项是文件路径中的一部分。如一个文件路径名是“/test/testfile”，则包含的目录项为：根目录“/”，目录“test”和文件“testfile”，这三个都是目录项。一般而言，目录项包含目录项的名字（文件名或目录名）和目录项的索引节点（见下面的描述）位置。
* 索引节点：UNIX将文件的相关元数据信息（如访问控制权限、大小、拥有者、创建时间、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。

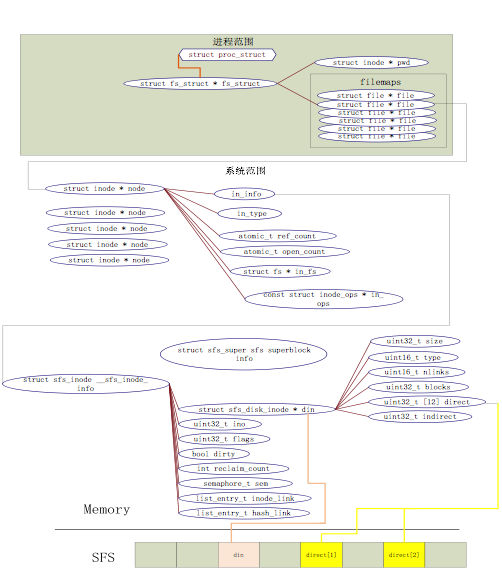


**ucore文件系统总体结构**

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**

### 9.3.2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。

### 9.3.3 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。

#### 9.3.3.1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。

image

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。

#### 9.3.3.2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。

### 9.3.4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。

#### 9.3.4.1 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。

#### 9.3.4.2 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成员变量

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。

### 9.4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。

#### 9.4.1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。

#### 9.4.2 stdout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。

#### 9.4.3 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;

}

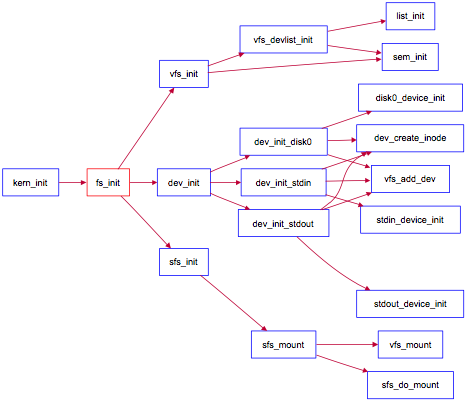
在上述函数中可以看出，如果p\_rpos < p\_wpos，则表示有键盘输入的新字符在stdin\_buffer中，于是就从stdin\_buffer中取出新字符放到iobuf指向的缓冲区中；如果p\_rpos >=p\_wpos，则表明没有新字符，这样调用read用户态库函数的用户进程就需要采用等待队列的睡眠操作进入睡眠状态，等待键盘输入字符的产生。

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。

### 9.5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。

### 9.6文件操作实现

#### 9.6.1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。

1. 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。
2. 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。
3. 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。

#### 9.6.2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。

## 9.7实验报告要求

从git server网站上取得ucore\_lab后，进入目录labcodes/lab8，完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录labcodes/lab8下存放实验报告，实验报告文档命名为lab8-学堂在线ID.md。推荐用**markdown**格式。对于lab8中编程任务，完成编写之后，再通过git push命令把代码同步回git server网站。最后请一定提前或按时提交到git server网站。

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。