实验 6：文件系统

一、实验目标

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；

二、程序清单

本次实验主要是理解kern/fs目录中的部分文件，并可用user/sfs_*.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/sfs_*.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/vfs/*.[ch]：虚拟文件系统接口与实现
- Simple FS文件系统
  - kern/sfs/*.[ch]：SimpleFS文件系统实现
- 文件系统的硬盘IO接口
  - dev/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
  - mksfs.c：创建一个Simple FS文件系统格式的硬盘镜像
- 对内核其它模块的扩充
  - kern/process/proc.h：增加成员变量 struct fs_struct *fs_struct，用于支持进程对文件的访问；
  - kern/init/init.c：增加调用初始化文件系统的函数fs_init。

三、准备知识

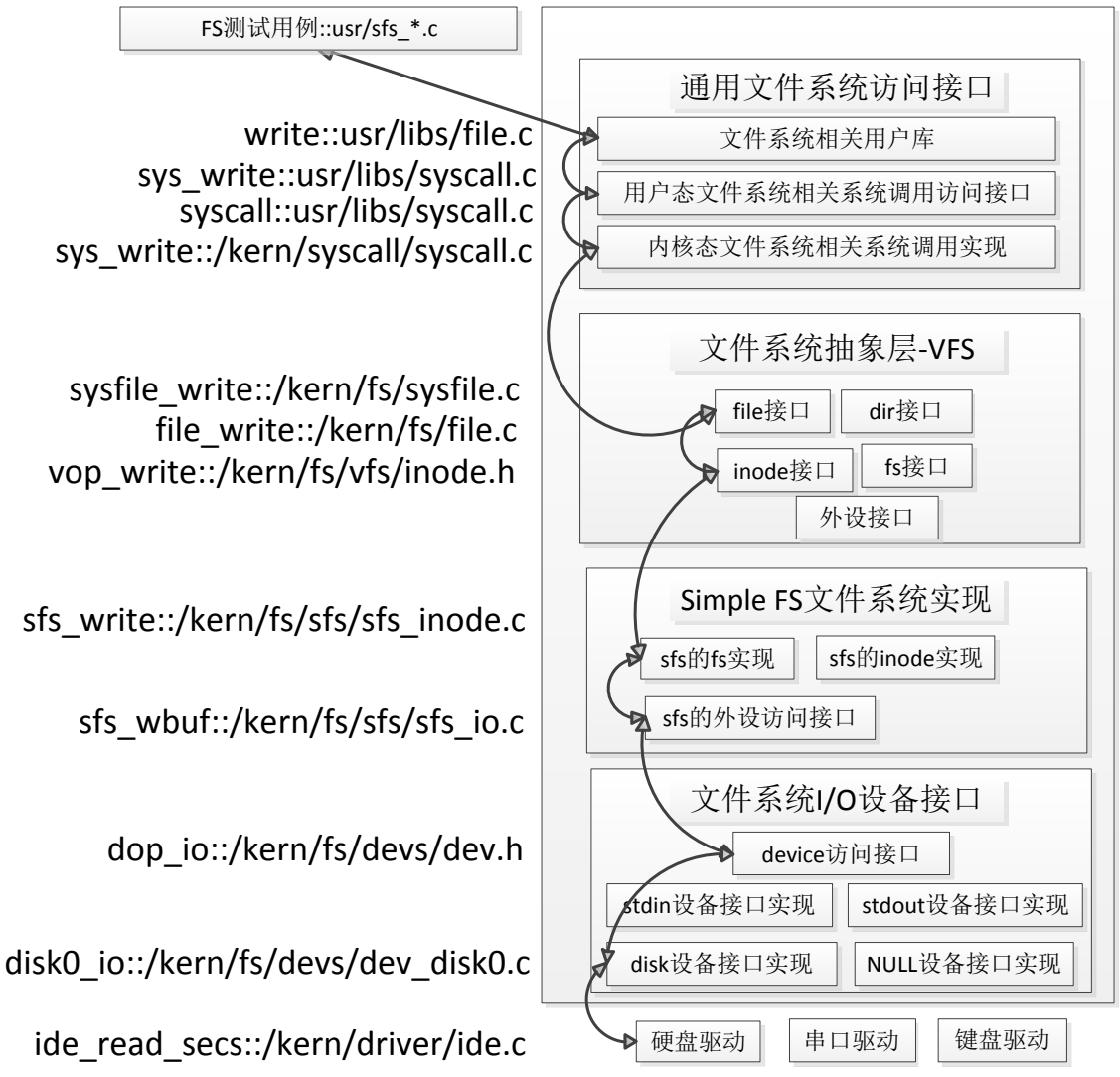
1.1 ucore 文件系统总体介绍

许多应用需要对数据进行长期保存和访问，这样就需要文件系统的支持。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。ucore文件系统主要由四部分组成：

- 通用文件系统访问接口
- 文件系统抽象层
- Simple FS文件系统
- 与硬盘驱动接口

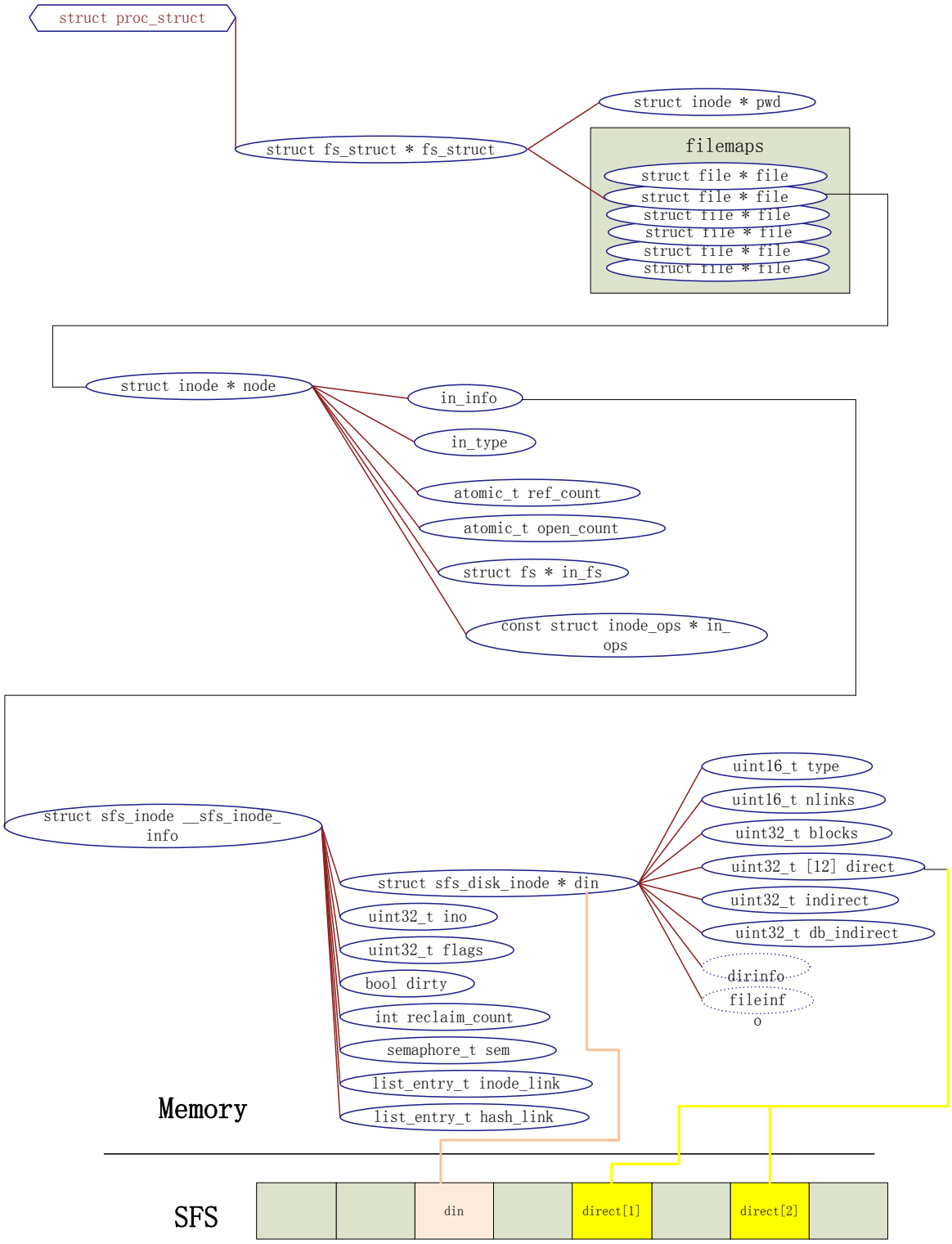
结合用户态写文件函数write的整个执行过程，我们可以比较清楚地看出ucore文件系统的层次和依赖关系。

ucore文件系统总体结构



如果一个用户进程打开了一个文件文件，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

Lab6 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 chdir、readdir 用户库函数对应的是 sys_chdir、sys_getdirentry 系统调用。这 6 个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，在讲具体的文件系统。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 6 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。
- 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。
- 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。

在 lab6 中关注的主要是 SFS 支持的常规文件和目录的设计实现。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，第三个磁盘用于存放一个 SFS 文件系统。SFS 文件系统的布局如下图所示。

superblock	root-dir inode	freemap	Inode/File Data/Dir Data blocks

第 0 个块是超级块（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；成员变量 blocks 记录了 SFS 中所有 block 的数量；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，一个 inode 占用一个块。

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 {
```

```
    union {
        struct {
            uint32_t size;                  如果 inode 表示常规文件，则 size 是文件大小
        } fileinfo;
        struct {
            uint32_t slots;                 如果 inode 表示目录，则 slots 表示目录下的文件项数
            uint32_t parent;                如果 inode 表示目录，则表示父目录
        } dirinfo;
    };
    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
    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
    uint32_t db_indirect;                   此 inode 的二级间接数据块索引值
};
```

通过上表可以看出，如果 inode 表示的是文件或目录，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。db_indirect 是所谓的 double indirect block，里面存放的全是 indirect block 索引。这样就可以支持很大的文件。

但需要注意，常规文件的内容数据就是实际的文件数据，但目录的内容数据是包含了目录下所有的文件名和对应的索引节点所占的数据块索引所形成的一个大的数组。每个数组元素可以用如下结构表示：

```
/* 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
};
```

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。

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 pipe_root __pipe_root_info;
        struct pipe_inode __pipe_inode_info;  //pipe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_pipe_root_info,
        inode_type_pipe_inode_info,
        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_mkdir)(struct inode *node, const char *name);
    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_unlink)(struct inode *node, const char *name);
    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 文件操作实现

3.5.1 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。

四、任务

4.1 分析关闭文件操作

参考3.5.1的分析方式，分析关闭文件操作的实现过程。

4.2 分析读写文件操作

参考3.5.1的分析方式，分析读写文件操作的实现过程。

五、扩展任务(可选)

5.1 实例分析 (有空建议尝试做做)

在lab6代码中，请扩展ucore相关函数和实现一个应用程序dump_fileinfo，应用程序dump_fileinfo的执行参数是一个路径名，比如参数为“/test/testfile”，dump_fileinfo或ucore内核的输出是各个子目录和文件的磁盘索引节点所在的磁盘块号，即一个输出的例子（注意，数字不正确，只是举例）是：

“/”：1

“test”：28

“testfile”: 38

5.2 分析硬链接的实现操作(有空建议尝试分析一下代码)

5.3 分析创建文件的实现操作(有空建议尝试分析一下代码)

六、实验要求

6.1 实验的流程和评分标准

（1）在试验报告上完成代码分析要求4.1和4.2；

（2）有空建议完成代码实现要求5.1，实现相应功能；

注意，凡是自己实现的代码都需要在提交的作业文档中清楚描述实现思路，遇到的问题等。希望同学们不要照抄或者基本照抄参考实现，这种情况将不能获得实验分数。

在试验报告中写出你练习的设计思路，如何实现，碰到的问题，如何解决，你的体会和对实验的建议等，并将你的实验报告与代码打包后一并提交。

6.2 检查你的完成情况

从网站上下载 lab6.tar.bz2 后，解压得到代码目录 lab6，完成实验。在实践中完成实验的各个练习。在报告中写出自己遇到的问题及解决方案。解压代码后，在 proj18 中执行：

make run-sfs_filetest1

可得到如下输出

```
touch -c kern/process/proc.c
make      --quiet      --no-print-directory      "DEFS+=-DTEST=sfs_filetest1      -
DTESTSTART=_binary_obj___user_sfs_filetest1_out_start                              -
DTESTSIZE=_binary_obj___user_sfs_filetest1_out_size"
+ cc kern/process/proc.c
+ ld bin/kernel
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.0364926 s, 140 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 6.6244e-05 s, 7.7 MB/s
8098+1 records in
8098+1 records out
4146474 bytes (4.1 MB) copied, 0.0226681 s, 183 MB/s
qemu        -parallel        stdio        -hda        bin/ucore.img        -drive
file=bin/swap.img,media=disk,cache=writeback                                       -drive
file=bin/fs.img,media=disk,cache=writeback -serial null
(THU.CST) os is loading ...
```

```
Special kernel symbols:
  entry  0xc010002c (phys)
  etext  0xc0125798 (phys)
  edata  0xc04e8876 (phys)
  end    0xc04eeeec (phys)
Kernel executable memory footprint: 4028KB
memory management: buddy_pmm_manager
e820map:
  memory: 0009fc00, [00000000, 0009fbff], type = 1.
  memory: 00000400, [0009fc00, 0009ffff], type = 2.
  memory: 00010000, [000f0000, 000fffff], type = 2.
  memory: 07efd000, [00100000, 07ffcfff], type = 1.
  memory: 00003000, [07ffd000, 07ffffff], type = 2.
  memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_slab() succeeded!
check_vma_struct() succeeded!
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: MLFQ_scheduler
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
ide 2:     262144(sectors), 'QEMU HARDDISK'.
check_swap() succeeded.
check_mm_swap: step1, mm_map ok.
check_mm_swap: step2, mm_unmap ok.
check_mm_swap: step3, exit_mmap ok.
check_mm_swap: step4, dup_mmap ok.
check_mm_swap() succeeded.
check_mm_shm_swap: step1, share memory ok.
check_mm_shm_swap: step2, dup_mmap ok.
check_mm_shm_swap() succeeded.
sfs: mount: 'simple file system' (201/32567/32768)
vfs: mount disk0.
++ setup timer interrupts
kernel_execve: pid = 3, name = "sfs_filetest1".
init_data ok.
random_test ok.
sfs_filetest1 pass.
sfs: cleanup: 'simple file system' (201/32567/32768)
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:580:
    initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

如果你打算做 5.1，可参考 sfs_filetest1.c 的写法完成 dump_fileinfo.c。在 dump_fileinfo.c 中指定一个字符串 path 代表文件路径（可在 disk0 中找一个存在的文件路径），然后执行新的系统调用 sys_dumpfileinfo，参数就是 path，然后 ucore 就可以打印出此路径中各个目录或文件的磁盘 inode 所在的块号。

实验报告文档命名为 lab-学生 ID.txt。推荐用 txt 格式，即基本文本格式，保存时请用 UTF-8 编码。对于 proj 18 函数编写任务，完成编写之后，在对应 proj 目录下执行 make handin 任务，即会自动生成 proj18-handin.tar.gz 文件。建立一个目录,名字为 lab6 result，将实验报告文档和之前生成的 handin 文件放在该目录下。然后用 tar 软件压缩打包此目录，并命名为 lab6-学生 ID.tar.bz2（在 lab6_result 的上层目录下执行“tar -cjvf lab6-学生 ID.tar.bz2 lab6_result”即可，请不要用其他软件例如 WinRAR 压缩）。最后请一定提前或按时提交到网络学堂上。

文件内容如下所示，请注意区分大小写

```
lab6-2007011350.tar.bz2
+-- lab6_result/
|-- lab6-2007011350.txt
+-- proj18-handin.tar.gz
```