Lab15 文件系统介绍

一、 实验概述

进一步了解Simple File System。

二、实验目的

1. 掌握Simple File System的具体构造
2. 简单了解操作系统对文件的读写过程

三、实验项目整体框架概述

Lab15
├── kern
│ ├── debug
│ ├── driver
│ │ ├── clock.c
│ │ ├── clock.h
│ │ ├── console.c
│ │ ├── console.h
│ │ ├── **ide.c** //设备初始化
│ │ ├── **ide.h**
│ │ ├── intr.c
│ │ ├── intr.h
│ │ ├── kbdreg.h
│ │ ├── **ramdisk.c** //磁盘初始化及相关函数
│ │ └── **ramdisk.h**
│ ├── **fs** //文件系统
│ │ ├── 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
│   ├── mm
│   ├── process
│   ├── schedule
│   ├── sync
│   ├── syscall
│   └── trap
├── libs
├── Makefile
├── tools
│   ├── function.mk
│   ├── kernel.ld
│   ├── mksfs.c //制作镜像
│   └── user.ld
└── user //用户文件
```

四、实验内容

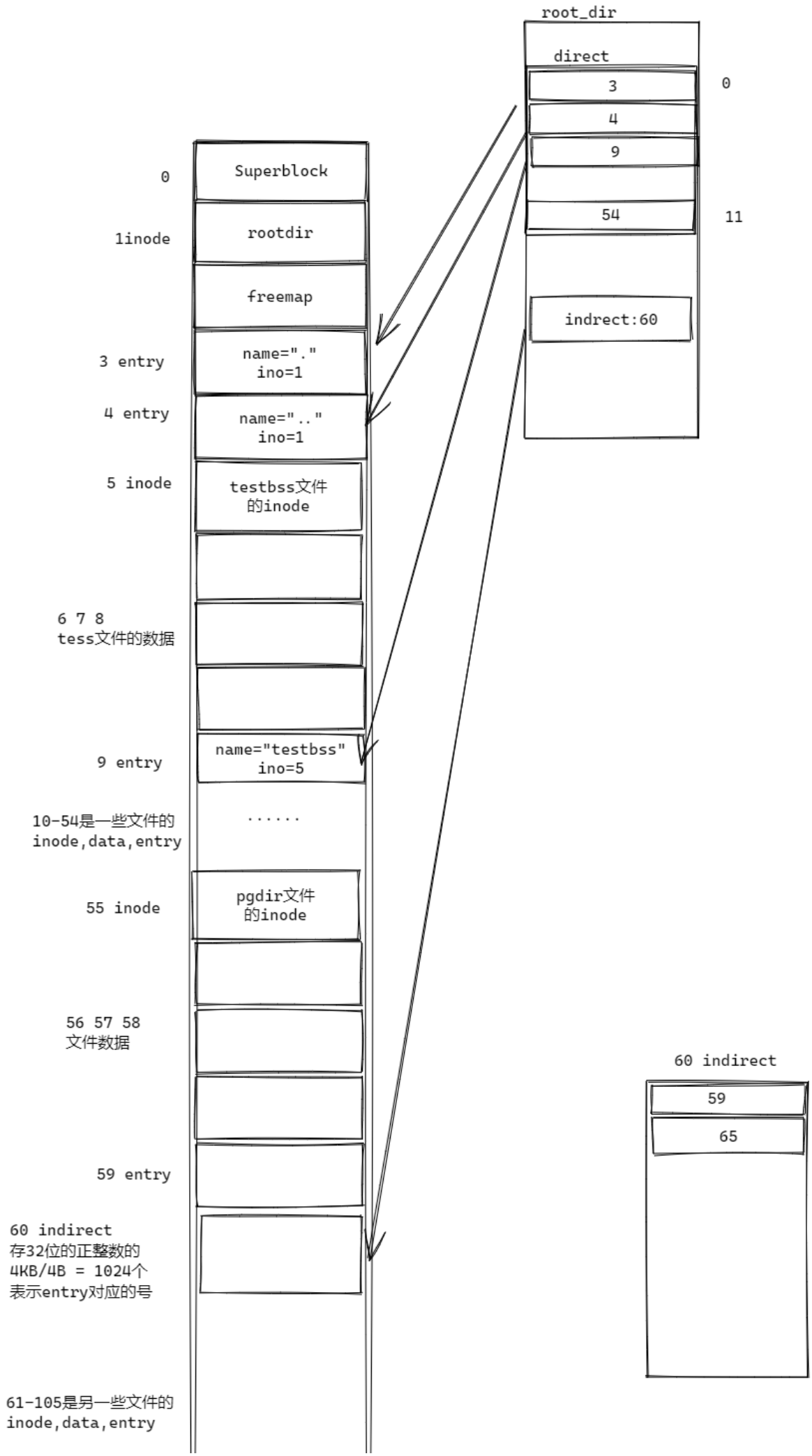
Step1. 创建一个SFS文件系统的磁盘镜像

在ucore实验包中，我们并没有一个真正的磁盘供操作系统读写，为了了解文件系统，我们首先创建一个镜像，通过将镜像加载到一段假装磁盘空间的内存空间来模拟对磁盘中文件系统的交互操作。

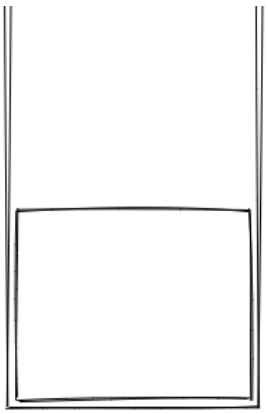
`make V=`中的指令`dd if=/dev/zero of=bin/sfs.img bs=1kB count=480`可以创建一个名为sfs.img空的镜像文件，它的内容全部是0，大小为480KB。

然后我们会把需要放入“磁盘中”的所有文件放入disk0文件夹下面，之后会通过`tools/mksfs.c`文件来构建镜像中的文件系统并把disk0文件夹下的文件放入磁盘中（在实验代码包中，我们编译产生了一些用户程序的可执行文件放入disk0文件夹）。

下图是我们的创建的磁盘镜像的整体结构。


root_dir
direct
3
4
9
54
0
11
indrect:60
0
Superblock
1inode
rootdir
freemap
3 entry
name="."
ino=1
4 entry
name=".."
ino=1
5 inode
testbss文件
的inode
6 7 8
tess文件的数据
9 entry
name="testbss"
ino=5
10-54是一些文件的
inode,data,entry
.....
55 inode
pgdir文件
的inode
56 57 58
文件数据
59 entry
60 indirect
存32位的正整数的
4KB/4B = 1024个
表示entry对应的号
61-105是另一些文件的
inode,data,entry
60 indirect
59
65

106-116为空闲块
freemap中
106-116位设置为1



Step2. sfs.img的创建过程——理解mksfs.c

注意我们的镜像创建过程是在ubuntu上进行的，也就是mksfs.c文件是在ubuntu下编译并运行的，所以这里使用的是C语言的库和ubuntu系统提供的系统调用。

准备工作

```
int
main(int argc, char **argv) {
    static_check();
    if (argc != 3) {
        bug("usage: <input *.img> <input dirname>\n");
    }
    const char *imgname = argv[1], *home = argv[2];
    if (create_img(open_img(imgname), home) != 0) {
        bug("create img failed.\n");
    }
    printf("create %s (%s) successfully.\n", imgname, home);
    return 0;
}
```

从`main`函数开始执行，argv[1]="bin/sfs.img"，argv[2]="disk0"。

我们会使用`open_img(imgname)`打开我们已经创建好的sys.img空文件，open_img使用了open系统调用获取了sys.img的文件描述符。接下来会调用create_sfs创建一个sfs_fs的结构体，其中包含超级块的内容。

> 在mksfs.c中，思路就是我们在内存中把我们要写入的磁盘的内容构造好，在合适的时间写入磁盘镜像。
>
> 写入的顺序为：
>
> . 文件目录
> ..文件目录
>
> 可执行文件的内容
> 可执行文件对应的entry
>
> freemap
> 超级块
> inode和block

```
struct sfs_fs {
struct {
  uint32_t magic;
```

```
  uint32_t blocks;
  uint32_t unused_blocks;
  char info[SFS_MAX_INFO_LEN + 1];
} super;//要写入磁盘的超级块

//----------------------------
//这一部分不会写入磁盘
struct subpath {
  struct subpath *next, *prev;
  char *subname;
} __sp_nil, *sp_root, *sp_end;
int imgfd;
uint32_t ninos, next_ino;
struct cache_inode *root; //指向root-dir inode

struct cache_inode *inodes[HASH_LIST_SIZE];  //要添加的文件的inode首先会链接在这里，最后写到磁盘中去
struct cache_block *blocks[HASH_LIST_SIZE]; //这里存放indirect指向的block中
//根据一个值，进行hash操作之后，会映射到0-1023之间的一个数，然后链接到对应位置。
//root-dir 为 0
//可执行文件为实际在ubuntu上的inode号
//block为要存在到磁盘镜像的块号
//----------------------------
};
```

我们看到在这个结构体中包含了一个super结构体，这才是要我们要写入磁盘镜像第0块的内容，其他的内容不会写入磁盘，但是会在我们的程序中用到。

在create_sfs的函数中，我们会填充super结构体的内容，magic，blocks，unused_blocks(这个会随着构造的过程而变化)，以及info。我们会在最后把这一块的内容写入磁盘镜像。

然后我们会初始化sfs_fs结构体中的其他内容，img_fd是sys.img的文件描述符， ninos是可用块的总数117(0-116)，next_ino记录可用的下一块的块号。目前是3（0是超级块，1是rootdir inode，2是freemap）。

```
sfs->root = alloc_cache_inode(sfs, 0, SFS_BLKN_ROOT, SFS_TYPE_DIR);
```

在函数的最后，我们会调用alloc_cache_inode函数来分配一个cache_inode 结构体，这是我们的root-dir inode。

```
struct cache_inode {
    struct inode {
        uint32_t size;
        uint16_t type;
        uint16_t nlinks;
        uint32_t blocks;
        uint32_t direct[SFS_NDIRECT];
        uint32_t indirect;
        uint32_t db_indirect;
    } inode;  //  要写入磁盘
    //  下面的是辅助信息
    ino_t real;
    uint32_t ino;//对应的磁盘块号，inode的编号就是我们的磁盘块号
    uint32_t nblks;//这个inode指向的文件或者目录的总块数
    struct cache_block *l1, *l2;//如果用到间接索引，会进行相应的记录
    struct cache_inode *hash_next;
};
```

我们会设置ino=1，磁盘中的第1块是我们的root-dir inode，然后会设置inode结构体的类型是一个目录，目前的nblks为0，链接到sfs->inodes的0号位置。

开始创建镜像

前面是一些准备工作，我们还没有往磁盘镜像中写任何内容，都是在内存中进行操作。下面我们开始创建的我们的镜像`create_img(open_img(imgname), home)`。

首先我们会把当前目录切换到disk0，目录下面已经存好了我们的编译好的elf可执行文件。

`open_dir(sfs, sfs->root, sfs->root)`

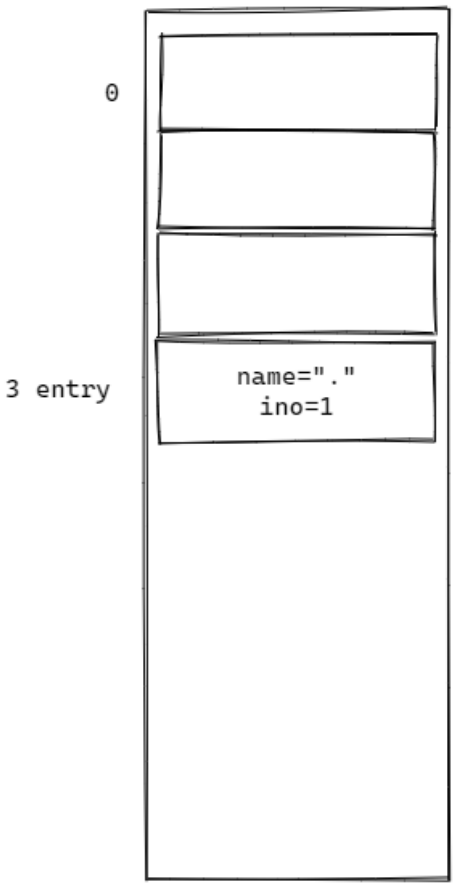
sfs是创建的struct sfs_fs 的结构体，sfs->root 是指向root dir的cache_inode。

添加. 和 .. 两个目录项

首先我们会创建两个entry项，分别是`.`和`..`。

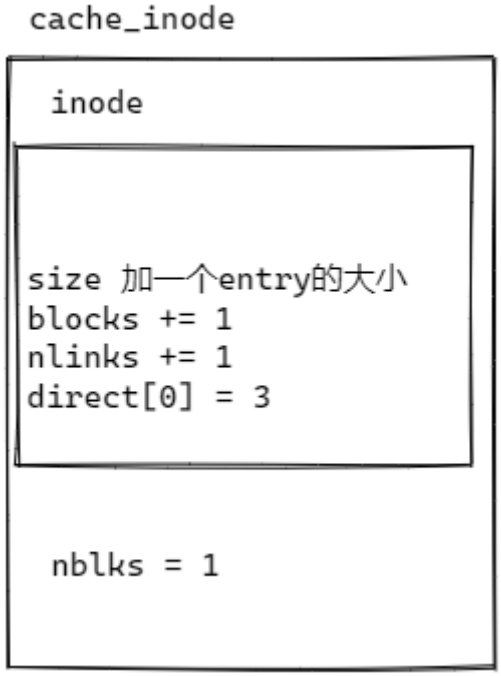
```
struct sfs_entry {
    uint32_t ino; //entry指向的inode
    char name[SFS_MAX_FNAME_LEN + 1]; entry指向的文件或者目录的名字
};
```

`.`对应的文件名`.`，inode号为root-dir inode号，这是我们要写入磁盘镜像的entry信息。但是我们应该写到那里呢，这时候就要从我们sfs_fs结构体中的next_ino获取了，目前是3，然后我们调用write_block函数把entry写到3号块对应的内容。我们的镜像里终于有第一块内容了！



当我们添加完一个entry后，需要修改root-dir inode的信息，主要是修改大小，块数，然后让direct中的某一项指向这个entry。

这里我们会调用`append_block`修改root-dir inode里的信息。



添加`..`的entry项和上面是类似的操作，由于我们这是根目录，没有上级目录，`..`对应的entry项中的inode号还是指向root-dir。我们将`..`对应的entry写入到4号块中，然后修改根目录inode结构体的相关内容。

添加可执行文件

接下来就是在我们的目录中添加可执行文件的内容以及对应可执行文件的entry。

```
while ((direntp = readdir(dir)) != NULL)
```

```
const char *name = direntp->d_name;
```

```
struct stat *stat = safe_lstat(name);
```

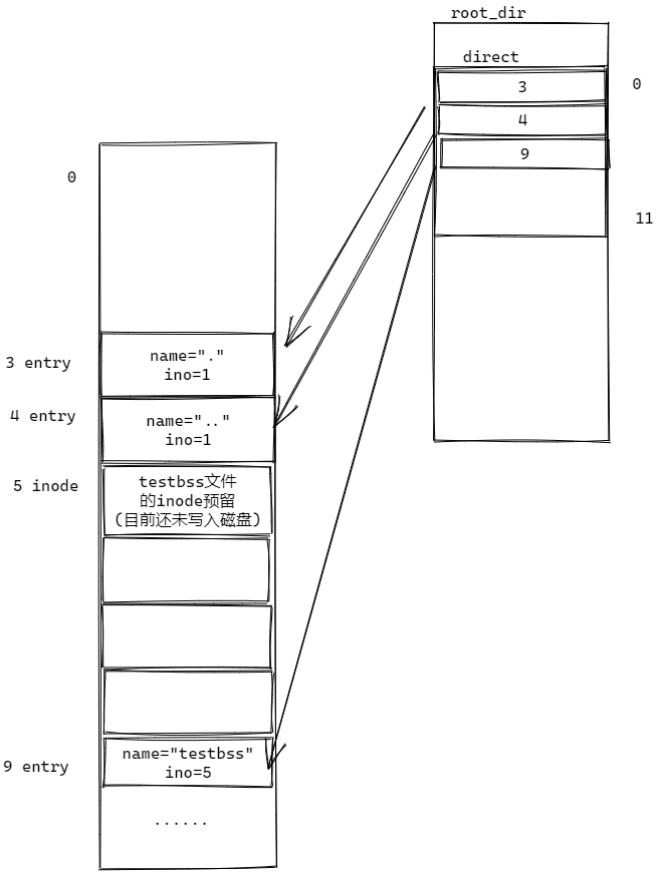
通过这三行代码，可以依次读取到disk0目录下的文件，获取到文件名以及文件的stat结构体。首先我们会打开文件`fd = open(name, O_RDONLY)`，接着调用`add_file(sfs, current, name, fd, stat->st_ino);`在我们的磁盘镜像中添加相关内容。

我们以添加的第一个文件testbss为例，来讲解这个过程。

首先我们会查找这个这个文件的cache_inode是否在已经存在，具体方法是取这个文件存放在ubuntu上的inode号(ubuntu上的inode号)进行hash操作，映射到0-1023之间，然后再sfs_fs结构体cache_inodes数组的对应位置查找是否存在。我们这是第一次查找，是不存在的。然后我们就要创建一个新的cache_inode，分配到的inode号是5，然后设置cache_inode结构体的属性，添加到sfs_fs的inodes链表中，设置cache_inode中inode中的type为文件，然后连接到对应的sfs_fs结构体中的inodes对应hash值的位置。

open_file函数会把testbss文件的具体内容写到磁盘镜像中去，并且会修改testbss对应的cache_inode结构体，testbss文件会占用三个磁盘块，分别写到了磁盘镜像的6 7 8号块中。

最后`add_entry(sfs, current, file, filename)`会新创建一个entry项，`name="testbass" ,ino=5`,放在一个新的磁盘块中，根据next_ino，值是9，然后让root-dit inode 中direct[2] = 9，并修改root-dit inode的其他内容。



接下来会重复添加testbss文件的操作。

文件divzero的inode对应位置10，文件数据11、12、13，entry14。

然后依次是priority，spin，softint，matrix，sleep，forktree，badarg，yield。

文件pgdir的inode对应位置55，文件数据56、57、58，entry59。

direct数组不够用了

当我们把pgdir的entry写入磁盘块59，修改root-dir的inode时(发生在append_block中)，应该写到direct[12]中，但是已经越界了，这时候我们的indirect就派上用场了。

```
static void
append_block(struct sfs_fs *sfs, struct cache_inode *file, size_t size, uint32_t
ino, const char *filename) {
    //...
    if (nblks < SFS_L0_NBLKS) {
        inode->direct[nblks] = ino;
    }
    else if (nblks < SFS_L1_NBLKS) {  //nblks = 12
        nblks -= SFS_L0_NBLKS;
        update_cache(sfs, &(file->l1), &(inode->indirect));
        uint32_t *data = file->l1->cache;
        data[nblks] = ino;
    }
    //...
}
```

这时候我们会分配一个block结构体，链接在我们的sfs_fs的blocks数组中，ino号是我们新分配的磁盘块号60，cache是要写入磁盘块的内容。让sfs_fs的l1指针指向这个结构体，indirect的值设置为60。block中cache的第一项设置为59，也是第一个间接索引。

```
struct cache_block {
    uint32_t ino;  //indirect 对应的磁盘块号
    struct cache_block *hash_next;
    void *cache;  //要写入磁盘块的内容，大小是4096B，相当于一个int的数组，依次存放entry的
块号
};
```

接下来是添加其他的文件，forktest，faultreadkernel，exit，faultread，sh，sleepkill，waitkill，badsegment，hello，最后使用的块号是105块，所以空闲块是106-116。

这时候我们所有inode，超级块以及freemap还没有写入磁盘，这些操作在`close_sfs(sfs)`来完成。

收尾工作

在`close_sfs(sfs)`中，我们首先会设置一个4096bits 的buffer，将106-116位设置为1，然后写入到磁盘中。随后将超级块写入到磁盘中，最后将所有inode和blocks写到我们的硬盘中去。

Step3. 简单访问文件系统

这一步，我们在disk0中放入一个文本文件test，里面写入sustech。然后该文件会被mksfs.c放入文件系统内，之后我们可以在ucore中打开并读取这个文件。

首先我们执行`make qemu`指令，会产生disk0文件夹，然后再用文本编辑器产生test文件放入disk0。再执行`make qemu`指令就可以看到运行结果。（注意`make clean`会删除整个disk0文件夹）

在本次实验代码中，操作系统会产生进程执行hello可执行文件，hello可执行文件的源代码为：

```
int fd = open("test",O_RDONLY);
char buffer[1000];
read(fd,buffer,1000);
cprintf("%s\n", buffer);
return 0;
```

open

用户程序中执行的open函数`fd=open("test", O_RDONLY)`，会执行以下函数流程： 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。

sysfile_open会调用file_open函数，这个函数在kern/fs/file.c目录下。fd_array_alloc会在进程的打开文件表中找到一个未使用的file结构体项来填入文件的信息，这个file结构体在打开文件表对应的下标就是文件描述符fd的值。

> **file结构体**
>
> file结构体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：

```
// kern/fs/file.h
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
 int 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files_struct，其数据结构定义如下：

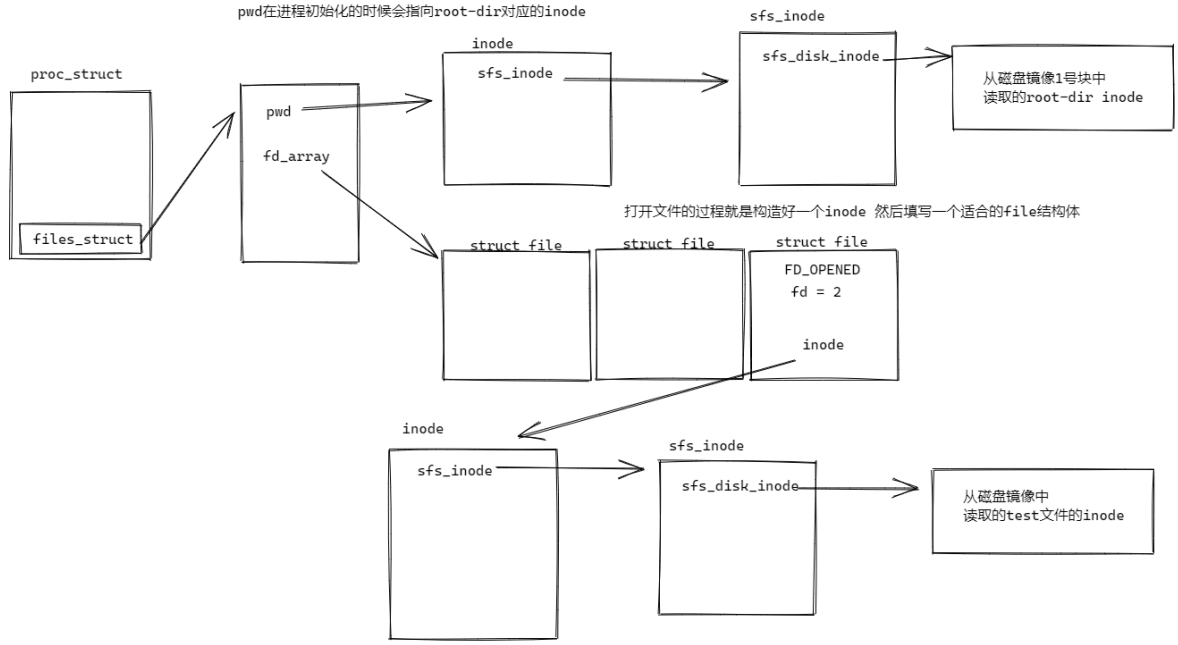
```
struct files_struct {
 struct inode *pwd;       // 进程当前执行目录的内存inode指针
 struct file *fd_array;   // opened files array
 int files_count;         // the number of opened files
 semaphore_t files_sem;   // 确保对进程控制块中fs_struct的互斥访问
};
```

当创建一个进程后，该进程的files_struct将会被初始化或复制父进程的files_struct。当用户进程打开一个文件时，将从fd_array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。

然后sysfile_open函数会调用vfs_open函数进行进一步的处理。而在vfs_open中会调用vfs_lookup来获取到文件对应的inode。

> 在进程初始化init_main的时候，会执行`vfs_set_bootfs("disk0:")`，这个函数会将磁盘里的root-dir inode读出来，然后构造进程可见的inode后，并存放在filesp->pwd项中。

vfs_lookup有两步，第一步是根据使用get_device获取到当前的打开目录，这个目录存放在filesp->pwd中。第二步是vop_lookup(node, path, node_store)，由于我们当前的inode是一个dir，这个函数会对应到dir的lookup操作，也就是sfs_inode里面的sfs_lookup函数。在这个函数中，我们构造test文件对应的inode，然后把test文件在磁盘镜像中对应的inode读取进来放在inode指向的sfs_inode里。过程如下：



需要注意的是，这里的inode不是sfs中的inode，而是VFS接口层包含磁盘inode信息的索引节点。

> **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：
>
> ```
> // kern/vfs/inode.h
> 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。

read

open之后，文件的相关信息被保存在了文件描述符fd对应的file结构体中，之后我们就可以使用read系统调用来读取文件的内容。同open，read会到内核的sysfile_read函数进程处理。

sysfile_read会调用file_read函数。首先会根据文件描述符找到文件的file结构体得到文件的inode。然后调用vop_read函数，对应于sfs_inode里的文件的sfs_read函数，sfs_read会通过底层操作读取文件的内容。

这一部分供大家了解，不做深入介绍。

六、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. SFS文件系统的结构
2. 读写SFS中文件的大致流程

七、下一实验简单介绍

下一次实验是复习课。