

国科大操作系统研讨课任务书

Project 6



版本：5.0

目录

一、实验说明.....	3
二、物理文件系统	4
2.1 superblock	4
2.2 sector/block map.....	4
2.3 inode map.....	4
2.4 inode.....	4
2.5 data.....	6
2.6 任务一：物理文件系统的实现	7
三、文件操作.....	11
3.1 文件描述符.....	11
3.2 读写文件.....	11
3.3 任务二：文件操作.....	12

一、实验说明

在之前的实验中我们实现了进程调度、例外处理、同步原语、进程通信、内存管理、网络驱动等功能，操作系统实验课已经接近了尾声，本章节也是实验课的最后的內容，在本次实验，大家将实现一个简单的物理文件系统，并实现简单的文件 I/O 访问函数。

对于本次实验，我们要求大家实现一个简单的物理文件系统，可以支持多级目录结构，并支持 `cd`、`mkdir`、`rmdir`、`ls` 等 shell 指令即可，对于文件操作，我们要求 `open`、`write`、`read`、`close` 等基本操作。对于有能力的同学，我们也非常希望大家能够将文件系统做的尽可能全面一些，因此我们会有额外的 `bonus` 供大家选择。本次实验必须实现的功能如下表所示：

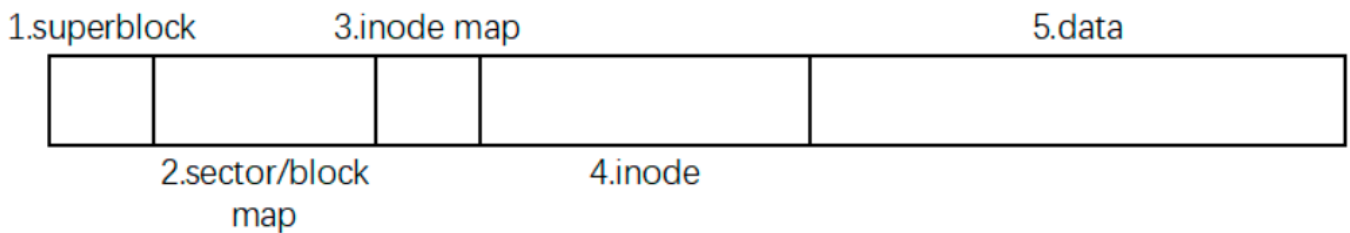
目录操作			
	函数名	shell 命令	说明
1	mkfs	mkfs	初始化文件系统
2	mkdir	mkdir	创建目录
3	rmdir	rmdir	删除目录
4	read_dir	ls	打印目录目录的内容
5	fs_info	statfs	打印文件系统信息，包括数据块的使用情况等
6	enter_fs	cd	进入目录
文件操作			
	函数名	shell 命令	说明
1	mknod	touch	建立一个文件
2	cat	cat	将文件的内容打印到屏幕
3	open		打开一个文件
4	read		读一个文件
5	write		写一个文件
6	close		关闭一个文件

二、物理文件系统

在本次实验，为了简单起见，我们只要求大家实现物理文件系统，而不必考虑现代操作系统普遍支持的虚拟文件系统。

所谓的物理文件系统的功能，大家可以通俗的理解为：描述和组织硬盘数据。对于数据，如果它只是单纯的放到硬盘上那只是数据，而当数据被有组织的存放到硬盘上，并对用户提供了可以管理数据的接口后，我们称将其为“文件系统”。

那么如何去在硬盘上组织我们的数据呢？下图为我们提供供大家参考的物理文件系统结构图：



关于图中 superblock、sector/block map、inode map、inode 都是什么，接下来我们将为大家一一讲解，当然，我们提供的思路仅供大家参考，大家如果有自己的想法，欢迎大家自己按照自己的思路去实现。

2.1 superblock

超级块——super block，占一个扇区（512 字节），它是文件系统最核心的数据结构，它里面记录了描述整个文件系统的关键数据，比如：文件系统的大小，sector/block map、inode map 等文件系统元数据的布局情况等。当内核启动的时候，从指定的硬盘数据块读取到了 superblock，识别成功后，才可以说找到了一个文件系统。并开始后续的初始化工作，可以说只有 super block 存在，那么一个文件系统才正常存在。

有的时候，文件系统需要两个 superblock，一个用来备份，防止系统宕机或者磁盘损坏。系统启动时，对比两个 superblock 来辨别文件系统是否出现问题。根据需要从备份的 superblock 恢复文件系统信息，备份 superblock 的位置可以自己设置。应该尽可能的和文件系统首的 superblock 不要放一起，防止两个 superblock 一起坏掉。

2.2 sector/block map

sector/block map 用来记录文件系统所占据的数据块使用情况，它所占大小和文件系统大小相关，它使用位图的方法去表示一个数据块的使用情况，当某一比特（bit）为 0 的时候，代表这个数据块没有使用过，为 1 代表已经被占用。当申请一块数据块的时候，通过查找 sector/block map 寻找空闲的数据块。Sector map 使用一位表示一个扇区（512B），block map 使用一位表示一个数据块。需要注意的是，一个数据块并不一定等于一个扇区。通常一个数据块的大小是一个扇区大小的倍数，比如：4096B 或者 8192B 等。目前，我们通常使用 4KB 大小的数据块。

例如，假设文件系统大小为 1G，数据块为 4KB，那么 1G 一共为 $1024 * 256$ 个数据块。使用 block map 标记这些数据块的使用情况一共需要 $1024 * 256$ 位（bit），即 32KB，共占用 8 个数据块。

请同学们根据自己的设计来决定使用 sector map 还是 block map 表示数据块的空间占用情况。

2.3 inode map

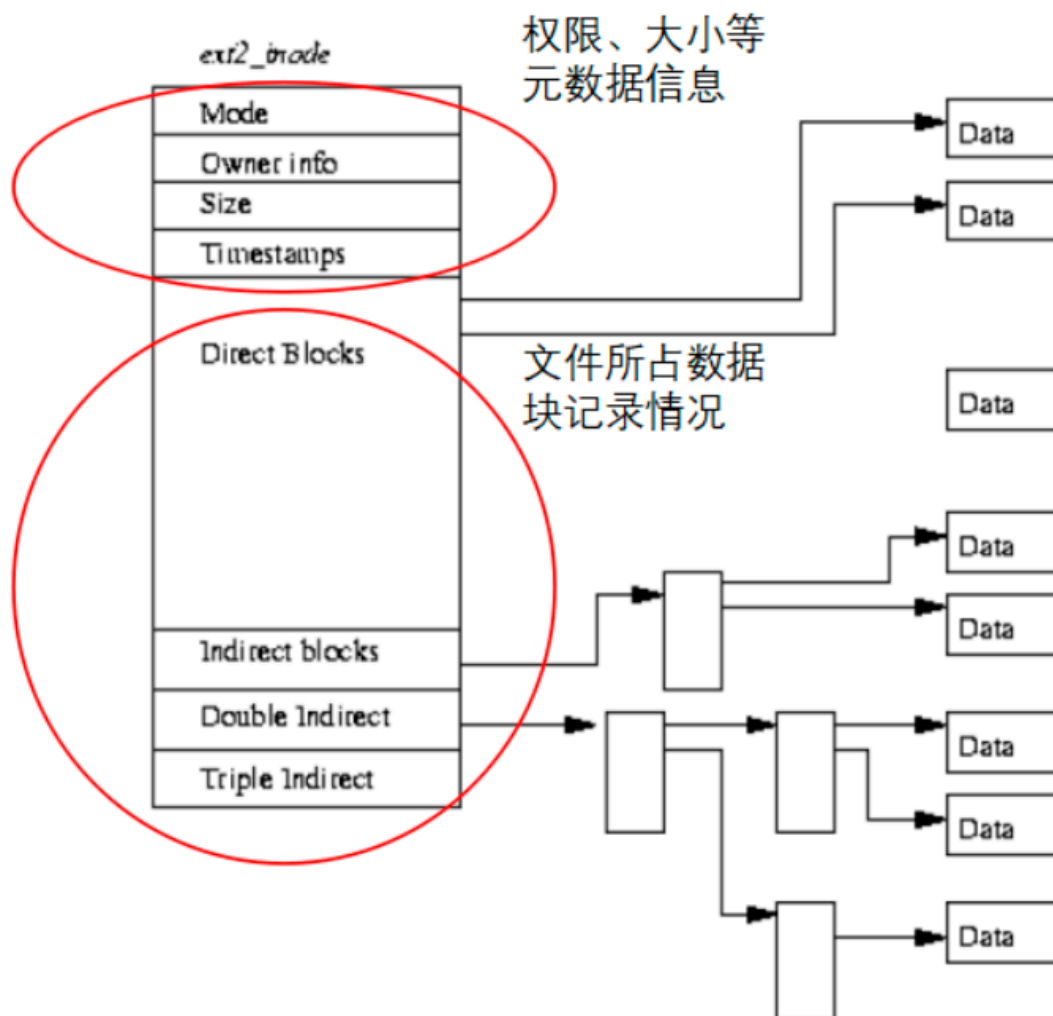
inode map，它所占大小和 inode 项数相关，同 sector/block map 类似，只不过它是用来记录 inode 的使用情况，当申请一个 inode 项的时候，通过查找 inode map 去寻找空闲的 inode。

2.4 inode

inode 用来描述一个文件或目录的数据结构，如果说 super block 是用来表示一个文件系统的关键数据结构，那么

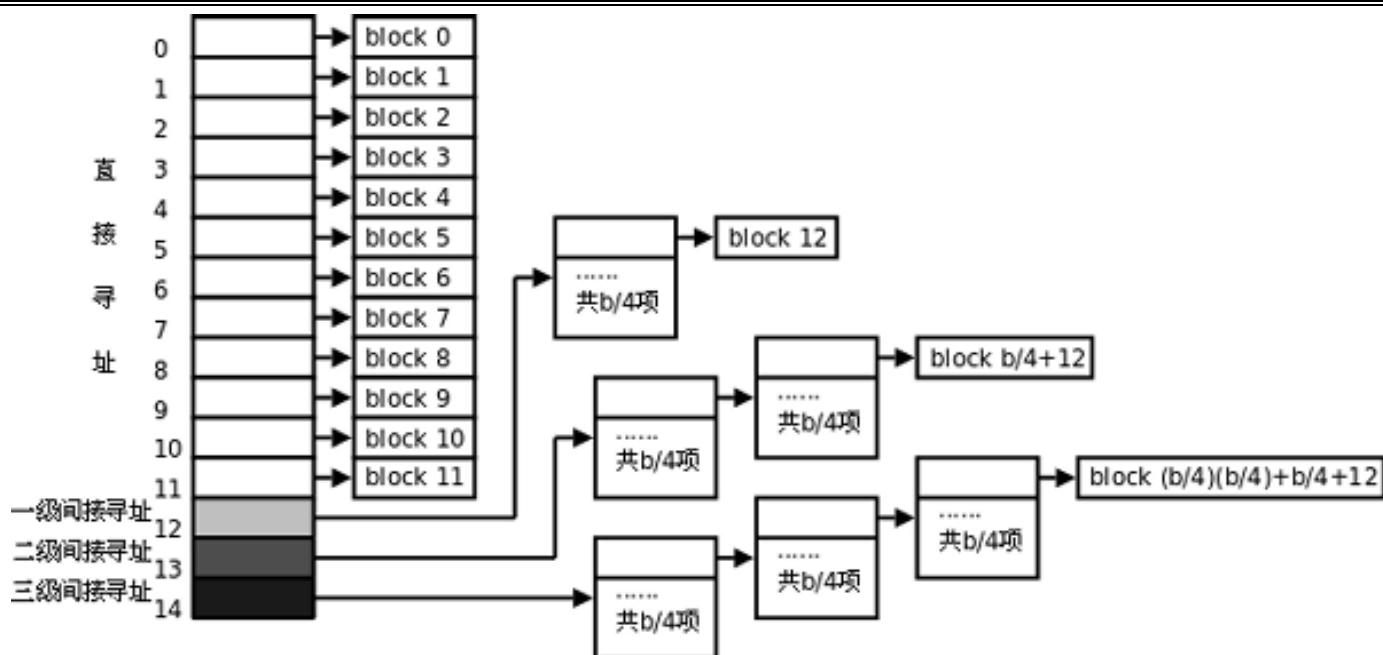
inode 就是用来表示一个文件或目录的关键数据结构。inode 里面存储了文件的元数据，比如：大小、类型、所占数据块号等。当你打开一个文件的时候，需要首先搜索当前目录的目录项，找到指定文件的目录项（directory entry）后找到该文件的 inode，然后通过 inode 的内容才知道文件的大小、权限、数据具体在哪些数据块保存等信息，然后找到对应的数据块，进行数据读写。

inode 的结构大体如下图所示，除了图中的表示信息外，同学们可以参考理论课上讲授的 inode 包含内容进行设计：



ext2 文件系统的 inode 结构

需要注意的是，对于数据块的索引，不同的文件系统做法不同。最直接的方法是在 inode 中开一个数组存放文件所占数据块的块号，这种方法我们称之为“直接索引”，这种做法的好处是简单，但是对于可能占据成百上千数据块的大文件而言，inode 里面显然不太合适存放上千个数据块的块号（可以是可以，但是会消耗大量空间，而且不一定每个文件都是大文件），因此我们可以采用“间接寻址”的方法去查找数据块，如下图：



简单来说，就是使用专门的数据块去记录文件所用数据块的块号，然后在 `inode` 中则记录上述这些专门的数据块的块号（请同学们仔细想一想间接寻址的方式，需要理解其含义），通过这种方式，为我们支持大文件提供了比较好的解决方法。

2.5 data

数据区，用来保存文件、目录的数据。对于文件的数据，那就只是单纯的数据，写进去就读出来什么，而至于目录，它占的数据块实际上存的是一个一个目录项。对于目录项，里面通常只保存文件的部分信息，比如：名称、`inode` 号。当需要打开一个文件或者目录的时候，通过读取目录项的 `inode` 号，找到 `inode` 项即可完成文件操作。

2.6 任务一：物理文件系统的实现

2.6.1 实验要求

设计并实现具有多级目录结构（**至少二级目录**）的物理文件系统，要求其实现以下功能，对于下述的功能，由于都属于内核功能因此我们要为其封装系统调用，请同学们注意：

ID	功能	对应 Shell 命令	说明
1	初始化文件系统	mkfs	建立文件系统
2	打印文件系统信息	statfs	打印文件系统的信息，包括但不限于：文件系统大小、文件系统数据块使用情况、文件系统 inode 使用情况。
3	进入一个目录	cd [filename]	进入一个目录
4	建立目录	mkdir [filename]	创建目录
5	删除目录	rmdir [filename]	删除目录
6	打印目录目录项	ls	打印出目录的目录项

2.6.2 实验步骤

（1）mkfs 如下图所示，要求初始化文件系统时打印初始化的信息。可以参考下图，在初始化的时候打印出了要初始化文件系统的信息，例如文件系统大小，起始扇区，inode map 偏移，sector/block map 偏移，inode 大小，dentry 大小，inode 数据区的偏移，文件数据区的偏移等信息：

```
----- COMMAND -----
[FS] Start initialize filesystem!
[FS] Setting superblock...
    magic : 0x66666666
    num sector : 1048576, start sector : 1048576
    inode map offset : 1 (1)
    sector map offset : 2 (256)
    inode offset : 258 (512)
    data offset : 770 (1047807)
    inode entry size : 64B, dir entry size : 32B
[FS] Setting inode-map...
[FS] Setting sector-map...
[FS] Setting inode...
[FS] Initialize filesystem finished!
```

（2）statfs 如下图所示，读取 superblock 后，尽可能详细的打印出文件系统的元数据信息。可以参考下图所示，该图打印出了文件系统所占的扇区数、扇区使用情况、inode 项数、inode 项使用情况等信息：

```
> root@UCAS_OS: statfs
magic : 0x66666666 (KFS)
used sector : 780/1048576, start sector : 1048576 (0x20000000)
inode map offset : 1, occupied sector : 1, used : 4/4096
sector map offset : 2, occupied sector : 256
inode offset : 258, occupied sector : 512
data offset : 770, occupied sector : 1047807
inode entry size : 64B, dir entry size : 32B
```

(3) 对于 mkdir 的操作，步骤如下：

- 查找到父目录，然后查找该目录是否存在，如果存在则返回错误并结束。
- 扫描 inode table 的 bitmap，查找空闲 inode，初始化该 inode。
- 在 inode 中初始化文件类型，大小，数据块索引等信息。
- 初始化 Inode 的 inode number，一个文件系统中每个文件的 inode number 都是唯一的。
- 初始化目录（和创建文件不同）：在新创建的目录中增加两个目录项(dentry)，即常见的 “.” 和 “..”，如下图所示。这两个 dentry 的类型都为目录，一个 dentry 的文件名为“.”，inode number 为新创建目录的父目录值。一个 dentry 的文件名为“..”，inode number 为新创建目录的 inode number 值。注意，根目录中的 “..” 目录项指向根目录本身。

```
drwxr-xr-x 1 parallels parallels 680 12月 3 12:43 .
drwxrwxr-x 1 parallels parallels 238 12月 3 12:40 ..
drwxr-xr-x 1 parallels parallels 102 8月 15 09:57 arch
-rwxrwxr-x 1 parallels parallels 1385 12月 3 12:43 bootblock
-rwxrwxr-x 1 parallels parallels 13872 12月 3 12:43 createimage
drwxr-xr-x 1 parallels parallels 136 10月 22 13:30 drivers
-rw-r--r-- 1 parallels parallels 10244 11月 30 17:36 .DS_Store
drwxr-xr-x 1 parallels parallels 136 11月 25 15:24 fs
-rw-rw-r-- 1 parallels parallels 77312 12月 3 12:43 image
drwxrwxr-x 1 parallels parallels 272 8月 15 10:11 include
drwxr-xr-x 1 parallels parallels 102 8月 15 10:05 init
drwxr-xr-x 1 parallels parallels 238 8月 15 10:04 kernel
-rw-rw-r-- 1 parallels parallels 409588 12月 3 12:43 kernel.txt
-rw-r--r-- 1 parallels parallels 1715 7月 26 10:27 ld.script
drwxr-xr-x 1 parallels parallels 272 11月 29 16:39 libs
-rwxrwxr-x 1 parallels parallels 135457 12月 3 12:43 main
-rw-r--r-- 1 parallels parallels 2390 12月 1 22:46 Makefile
drwxr-xr-x 1 parallels parallels 306 11月 29 16:39 QEMUloongson
drwxr-xr-x 1 parallels parallels 272 12月 1 20:04 test
drwxr-xr-x 1 parallels parallels 170 10月 28 17:22 tools
```

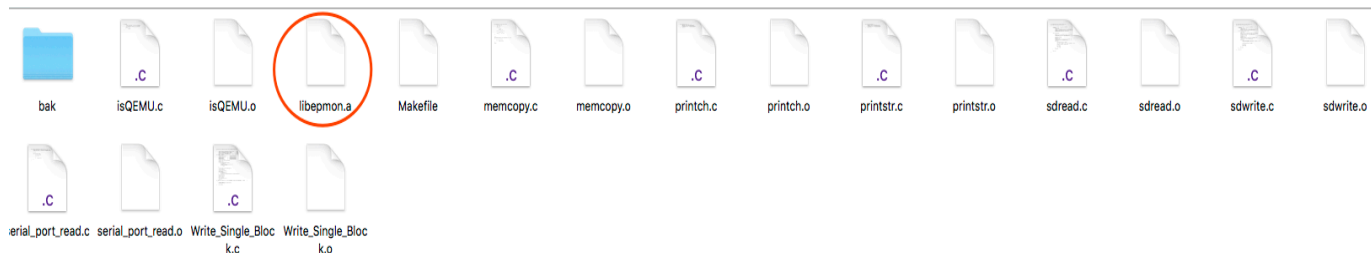
- 将 inode bitmap 中相应位标记为有效。
- 在父目录的数据块中分配 dentry 空间，初始化 dentry，在该 dentry 中记录所创建目录的 inode number 和名称，标记 dentry 的类型为目录。
- 更新父目录的修改时间，增加父目录的硬链接数（新建的目录有一个“.”指向父目录）。

(4) 关于 ls 指令，同学们可以自由发挥，基本要求是能查看目录下所有文件，建议同学们将其实现的更完善，例如可以支持 ls -al 等复杂的操作。

(5) **注意：**对于本次任务的检查，不设置 test task，在检查的时候助教们会现场让同学运行相关 shell 指令。

2.6.3 注意事项

(1) 本次实验需要使用对 SD 卡读、写的函数，我们已经提供好现成的库给大家了，在 libepmon.fixed.tgz 文件中，同学们解压即可。里面有的源代码和编译好的库文件，如下图：



对于有的想了解库实现的同学可以看一下源代码，如果不需要，只需要把 `libepmon.a` 文件拷贝到大家的 `lib` 目录下，然后在 `makefile` 的 `main` 指令中添加：`-L libs/ -lepmon`，如下图：

```

27 main : $(SRC_ARCH) $(SRC_DRIVER) $(SRC_INIT) $(SRC_INT) $(SRC_LO
28         ${CC} -g -G 0 -O0 -Iinclude -Itest/test_fs -Ifs -Ilibs -Iarc
29         -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=
30         $(SRC_TEST_FS) $(SRC_ARCH) $(SRC_FS) $(SRC_DRIVER) $(SRC_INI
31         -L libs/ -lepmon

```

如果同学们想在 `qemu` 中使用该库，需要将 `gzram` 文件拷贝到 `qemu` 文件夹的 `bios` 中，覆盖掉中间的 `gzram` 之后，可以直接在代码中调用 `sdread`、`sdwrite` 函数，对于函数的说明如下：

函数名申明	说明
<code>sdread(char *buff, uint32_t offset, uint32_t size);</code>	其中 <code>buff</code> 为要保存数据的缓冲区地址， <code>offset</code> 为要读数据在 SD 卡的偏移位置， <code>size</code> 为要读取数据大小。
<code>sdwrite(char *buff, uint32_t offset, uint32_t size);</code>	其中 <code>buff</code> 为要写数据的缓冲区地址， <code>offset</code> 为要读数据在 SD 卡的偏移位置， <code>size</code> 为要读取数据大小。

对于调用，直接在代码里通过函数名调用即可，如下图所示：

```

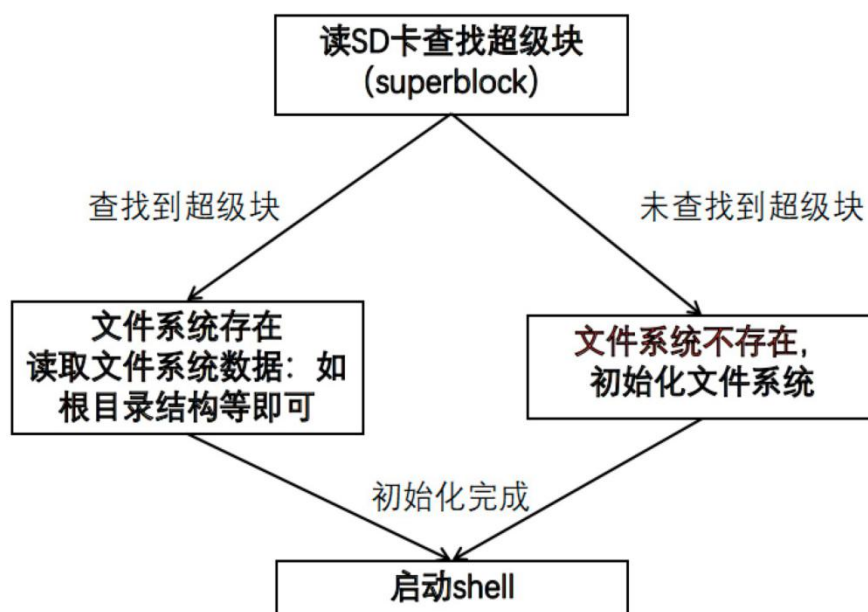
41 // use libepmon.a to read SD card
42 // sdread(char *buff, int offset, int size)
43 void sd_card_read(void *dest, uint32_t offset, uint32_t size)
44 {
45     sdread((char*)dest, offset, size);
46 }
47
48 // use libepmon.a to write SD card
49 // sdwrite(char *buff, int offset, int size)
50 void sd_card_write(void *dest, uint32_t offset, uint32_t size)
51 {
52     // printf("write:0x%x, size:%d\n", offset, size);
53     sdwrite((char*)dest, offset, size);
54 }
55

```

注意：对于选择 RISC-V 实验的同学，请大家直接使用 SBI 接口来读写 SD 卡。无需添加任何额外的库。需要注意的是，`sbi_sd_read` 和 `sbi_sd_write` 是通过 `ecall` 调用 `machine` 态的接口实现的，而 `machine` 态运行在物理地址。因此，传

递给 `sbi_sd_read` 和 `sdi_sd_write` 的地址必须是物理地址。由于文件系统的相关功能整个做在内核态，所以内核态虚地址可以通过简单地减去固定偏移得到物理地址（这是由于我们映射内核的时候是线性映射的，每个内核虚地址映射到了相应的物理地址上）。

(2) 请建立至少为 512MB 大小的物理文件系统。此外，对于文件系统的初始化，我们一方面要求可以通过在 shell 中执行 `mkfs` 命令手动初始化文件系统，在测试时我们可能会现场 `mkfs` 进行新建文件系统。另一方面，在内核启动的过程中，我们要求内核在初始化时去磁盘上查找 `superblock`，如果找到了，则表明磁盘上已经建立了文件系统，不必再进行初始化；若没有找到，则在内核启动的过程中让内核自动初始化文件系统，确保在 shell 启动后有一个文件系统可供使用，具体逻辑如下图所示：



内核启动时的文件系统初始化流程

(3) 物理文件系统在 SD 卡上的位置请大家在初始化文件系统时自行决定，注意不要覆盖生成的内核。例如，可以在 SD 卡的 512MB 处开始建立文件系统。

(4) 在 `mkdir/rmdir` 一个目录的时候，只用考虑单级目录，不考虑目录中还有子目录需要递归处理的情况。

(5) `cd` 和 `ls` 指令需要支持多级目录的寻址（至少两级），比如 `cd dir1/dir2/dir3`

三、文件操作

在上一个任务我们实现了物理文件系统，实现了多级目录结构，在本次任务中，大家将实现内核中对文件的访问，并制定文件的 I/O 函数。

3.1 文件描述符

如果要访问一个文件，首先需要对文件打开，其实所谓的打开可以分为两步：从文件系统查找文件、为查找到的文件建立文件描述符。将文件描述符返回给用户，之后用户对这个文件的访问将通过这个文件描述符进行。文件描述符的内容包括：要访问文件的 inode 号，打开的权限（可读、可写、读写），读写指针等。

注意：读写指针指的是在进行 read、write 的时候，读写的文件内部偏移位置 pos，例如当在文件偏移位置 pos 写入一个 size 大小的数据后，下次数据写入则是在 pos + size 的文件偏移处继续写入，这种定位是由调用函数通过读写指针去控制读写位置 pos 完成的。

比如使用 open 打开文件，首先通过路径信息，找到要打开文件的 inode 号，将 inode 号和权限等信息保存在文件描述符（fd）数组中，返回数组的下标，如下图：



3.2 读写文件

对于文件的读写，就是通过 fd 找到要读写的文件，进一步找到要读写文件的数据数据块进行读写，在这里我们为了简单起见，不要求大家实现数据 cache 等功能，大家在实验的过程中只要运行我们给定的测试用例，将数据最终写到存储介质上即可。

3.3 任务二：文件操作

3.3.1 实验要求

实现文件系统文件操作，具体要求如下，对于下述的功能，由于都属于内核功能因此我们要为其封装系统调用，请同学们注意：

ID	功能	对应 shell 命令	说明
1	建立一个空文件	touch [filename]	建立一个文件
2	打印文件内容	cat [filename]	打印一个文件到 shell
ID	功能	函数实现	说明
1	打开文件	int open(char*name, int access)	打开一个文件，传入的参数为文件名，打开权限（可读、可写、读写），返回文件描述符下标。
2	读文件	int read(int fd, char *buff, int size)	读一个文件，传入的参数为文件描述符下标，读取出来的数据要存放的缓冲区地址，要读数据的大小。返回实际写入的数据大小。
3	写文件	int write(int fd, char *buff, int size)	写一个文件，传入的参数为文件描述符下标，写入的数据缓冲区地址，要写数据的大小。返回实际写入的数据大小。
4	关闭文件	void close(int fd)	关闭一个文件，释放文件描述符。

3.3.2 实验步骤

（1）使用 touch 指令建立一个目录，如下图：

```

----- COMMAND -----
> root@UCAS_OS: touch 1.txt
> root@UCAS_OS: ls
.  ..  dev  tmp  mnt  1.txt
> root@UCAS_OS:

```

（2）运行我们给出的 test_fs.c 中的任务，该任务的内容为：打开 1.txt 文件，写入 10 句 “hello world”，读取 1.txt 文件的内容，打印出来，关闭文件，正确运行结果如下图：

```
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!

----- COMMAND -----
num sector : 1048576, start sector : 1048576
inode map offset : 1 (1)
sector map offset : 2 (256)
inode offset : 258 (512)
data offset : 770 (1047807)
inode entry size : 64B, dir entry size : 32B
[FS] Setting inode-map...
[FS] Setting sector-map...
[FS] Setting inode...
[FS] Initialize filesystem finished!
> root@UCAS_OS: touch 1.txt
> root@UCAS_OS: exec 0
exec task0
> root@UCAS_OS:
```

(3) 重启开发板，使用 `cat` 指令打印出 `1.txt` 中的内容，如下图：

```
----- COMMAND -----
> root@UCAS_OS: cat 1.txt
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
```

3.3.3 注意事项

- (1) 在写文件的时候确保每一步操作都持久化到存储介质上。
- (2) 在实验的过程中，助教可能会在一步操作后重启操作系统（比如 `touch` 后重启再运行实例），确保每一步操作是持久化到存储介质的。

3.3.4 Bonus (2 分)

在实现了基本的目录和文件操作后，同学们实现以下内容我们将给予加分：

- (1) 额外功能的实现：

ID	功能	对应 Shell 命令	说明
1	查找文件/目录	find	find [path] [name] 递归的查找目标路径下是否有对应名称的文件或目录，这里的 name 不要求实现正则语法匹配。
2	重命名文件/目录	rename	rename [old_name] [new_name]，修改目标文件/目录的名称
3	硬链接	ln	建立一个文件的硬连接，具体实现参考 Linux 指令 ln
4	软连接	ln -s	建立一个文件的软连接，具体实现参考 Linux 指令 ln -s

(2) 大文件的支持

在之前的 `inode` 中，我们没有对大家提出“间接索引”的要求，因此如果同学们只使用了“直接索引”方法，那么可以支持的文件大小有限，因此在本次实验中，我们要求大家使用“间接索引”对数据块进行索引，实现支持单文件大小至少为 256MB 的文件，能进行文件创建和读写。